Chapter 7
Predefined Data Modules

Maude has a standard library of predefined modules that, by default, are entered into the system at the beginning of each session, so that any of these predefined modules can be imported by any other module defined by the user. Also, by default, the predefined functional module `BOOL` is automatically imported (in `including` mode) as a submodule of any user-defined module, unless such importation is explicitly disabled. These modules can be found in the file `prelude.maude` that is part of the Maude distribution.

We describe below those predefined modules that provide commonly used data types, including Booleans, numbers, strings, and quoted identifiers. The relationships among these modules are shown in the importation graph in Figure 7.1, where all the importations are in `protecting` mode.

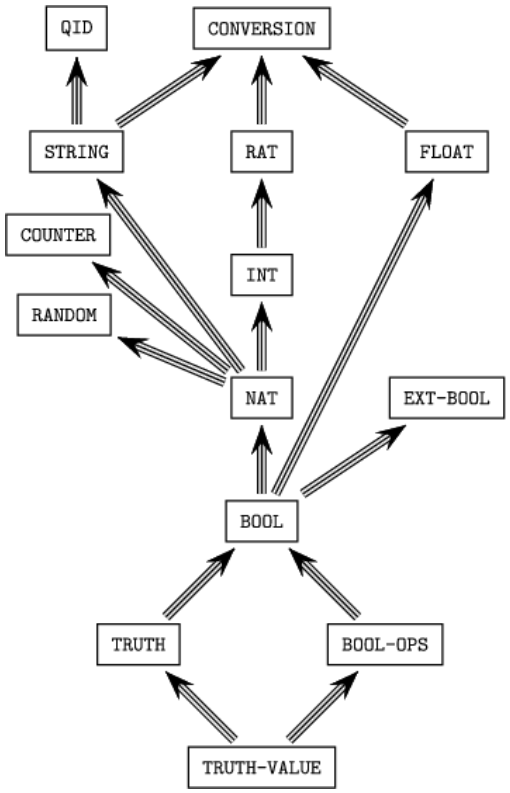

Figure 7.1: Importation (protecting) graph of predefined modules

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 11, the `LOOP-MODE` module in Section 13.1, and the `CONFIGURATION` module in Sections 8.1 and 8.4.

Furthermore, this chapter also describes a predefined module `MACHINE-INT` for machine integers, which is obtained from the module `INT` of (arbitrary size) integers, but is distributed in a separate file `machine-int.maude`.

As explained in Section 4.4.10, many operators in predefined modules are declared with the `special` attribute, so that they are to be treated as *built-in* operators associated with appropriate C++ code by "hooks" specified after the `special` attribute. In what follows, to lighten the exposition, we will omit the details about such hooks in special operators, writing `special (...)` instead. The full definitions can be found in the file `prelude.maude`.

Most built-in data types are algebraically constructed, that is, they are built out of constants and constructor operators; however, floating point numbers (floats), strings, and quoted identifiers (qids) are treated as countable sets of constants and are represented by "special" operators `<Floats>`, `<Strings>`, and `<Qids>`, respectively. These operators are used in specifying the hooks mentioned above, but they cannot be used explicitly in terms.

7.1 Boolean values

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

```
fmod TRUTH-VALUE is
  sort Bool .
  op true : -> Bool [ctor special (...)] .
  op false : -> Bool [ctor special (...)] .
endfm
```

This module just declares the two Boolean values `true` and `false` as constants of sort `Bool`. The key thing to note is the `special` attribute associated with each of the operator declarations for these constants. In the case of Boolean values this is especially important, because certain basic constructs of the language such as conditions in a conditional equation, membership axiom, or rule, and also sort predicates associated with membership assertions evaluate to these built-in truth values.

The module `TRUTH` adds three important operators to `TRUTH-VALUE`.

```
fmod TRUTH is
  protecting TRUTH-VALUE .
  op if_then_else_fi : Bool Universal Universal -> Universal
       [poly (2 3 0) special (...)] .
  op _==_ : Universal Universal -> Bool
       [poly (1 2) prec 51  special (...)] .
  op _=/=_ : Universal Universal -> Bool
       [poly (1 2) prec 51  special (...)] .
endfm
```

The operators are, respectively, `if_then_else_fi`, and the built-in operators for equality and inequality predicates.[1] These operators are special in a number of ways. Firstly, they are, by default, automatically added to every module (see Section 3.9.3). Secondly, they are *polymorphic*, so that, for each module, they can be considered to be normal operators that are ad-hoc overloaded for each connected component in the module. This is done by means of the `polymorphic` (or `poly`) attribute, as discussed in Section 4.4.4, and the symbol `Universal`, that should not be considered a common sort, as explained at the end of this section. For example, in the declaration of the `if_then_else_fi` operator, the attribute `poly (2 3 0)` means that `if_then_else_fi` is polymorphic in its second and third arguments

as well as in its result.

The `if_then_else_fi` operator first rewrites its first argument, the test. If the result is of sort `Bool`, the *then* or *else* argument is selected, according to whether the test evaluated to `true` or `false`, and rewritten. If the test result is not of sort `Bool` the *then* and *else* arguments are rewritten. For example, working in the `INT` module (see Section 7.4) we get the following reductions:

```
Maude> red in INT : if 4 - 2  == 2 then 0 else 1 fi .
result Zero: 0

Maude> red if 4 - 2  =/= 2 then 0 else 1 fi .
result NzNat: 1
```

The built-in Boolean predicates `_==_` and `_=/=_` have a straightforward operational meaning: given an expression `u == v` with `u` and `v` ground terms (i.e., terms without variables), then both `u` and `v` are simplified by the equations in the module (which are assumed to be Church-Rosser and terminating) to their canonical forms (perhaps modulo some axioms such as `assoc`, etc., see Section 4.4.1) and these canonical forms are compared for equality. If they are equal, the value of `u == v` is `true`; if they are different, it is `false`. The predicate `u =/= v` is just the negation of `u == v`.

Similar in spirit to the built-in operators for equality predicates, there are built-in operators for membership predicates: `_:: S` for each sort `S`. These do not need to be explicitly declared, because they are part of the extended signature associated with each module, as described in Section 3.9.3. The operational meaning for membership operators is analogous to that of the equality operators. Namely, given a term `u` and a sort `S` in its kind, the built-in predicate `u :: S` is evaluated by reducing `u` to its canonical form, computing its *least sort* (under the preregularity, Church-Rosser, and termination assumptions), and checking that it is smaller than or equal to `S`.

But what about the *mathematical* meaning of these built-in predicates? That is, do they really correspond to ordinary equations (and not to negations or other Boolean combinations of such equations)? The point is that these built-in and efficiently implemented equality and inequality predicates could in principle have been defined in a more cumbersome and inefficient way by the user. In fact, assuming that the equations and membership axioms in the user's module are Church-Rosser and terminating modulo the equational axioms in the operator attributes (see Section 4.4.1) and that the operators satisfy the preregularity requirement, the corresponding initial algebra is a *computable* algebraic data type, for which equality, inequality, and membership in a sort are also computable functions. Therefore, by a well-known theorem of Bergstra and Tucker [8], such predicates can themselves be equationally defined by Church-Rosser and terminating equations. It is of course very convenient, and much more efficient, to unburden the user from having to give those explicit equational definitions of the equality, inequality, and membership predicates by providing them in a built-in way.

Note also that, by the above meta-argument, the use of inequality predicates, negations of membership predicates, or any Boolean combination of such predicates in abbreviated Boolean conditions does not involve any real introduction of *negation* (or other Boolean connectives) in the underlying membership equational logic, which remains a Horn logic. What we are really doing is adding more Boolean-valued functions to the module, but such functions, although provided in a built-in way for convenience and efficiency, could have been equationally defined by the user without any use of negation.

The module `BOOL-OPS` imports `TRUTH-VALUE` and adds the usual conjunction, disjunction, exclusive or, negation, and implication operators.[2] These operators are defined entirely equationally.

```
fmod BOOL-OPS is
  protecting TRUTH-VALUE .
  op _and_ : Bool Bool -> Bool [assoc comm prec 55] .
  op _or_ : Bool Bool -> Bool [assoc comm prec 59] .
  op _xor_ : Bool Bool -> Bool [assoc comm prec 57] .
  op not_ : Bool -> Bool [prec 53] .
  op _implies_ : Bool Bool -> Bool [gather (e E) prec 61] .
  vars A B C : Bool .
  eq true and A = A .
  eq false and A = false .
  eq A and A = A .
  eq false xor A = A .
  eq A xor A = false .
  eq A and (B xor C) = A and B xor A and C .
  eq not A = A xor true .
  eq A or B = A and B xor A xor B .
  eq A implies B = not(A xor A and B) .
endfm
```

Finally, the module `BOOL` puts together all the operators in `TRUTH` and in `BOOL-OPS`.

```
fmod BOOL is
  protecting BOOL-OPS .
  protecting TRUTH .
endfm
```

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

```
set include BOOL on .
```

that appears in the standard library file `prelude.maude`. The `set include` command can mention any module we wish to import—in this case `BOOL`. However, this default importation can be disabled. For example, if the user wished to have the polymorphic equality, inequality and `if_then_else_fi` operators automatically added to modules, but wanted to exclude the usual Boolean connectives for the built-in truth values, this could be accomplished by writing

```
set include BOOL off .
set include TRUTH on .
```

Similar commands are available for enabling and disabling implicit importation in `extending` and `protecting` modes (see Section 18.13). For example, the `BOOL` module can be protected by default instead of included by writing

```
set include BOOL off .
set protect BOOL on .
```

The module `EXT-BOOL` declares short-circuit versions of the conjunction and disjunction operators such as those found in LISP and other programming languages. Like the operators declared in `BOOL`, these operators are defined entirely equationally. The short-circuit behavior is the result of the strategy attributes declared for the operators as discussed in Section 4.4.7.

```
fmod EXT-BOOL is
  protecting BOOL .
  op _and-then_ : Bool Bool -> Bool
      [strat (1 0) gather (e E) prec 55] .
  op _or-else_ : Bool Bool -> Bool
```

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

When the module `BOOL` is imported, the system automatically adds to the module an operator to test for membership, `_:: S`, for each sort `S`, as mentioned above (see also Section 3.9.3). Again, working in the `NUMBERS` module we have the following examples:

```
Maude> red in NUMBERS : sd(zero, zero) :: Zero .
result Bool: true

Maude> red sd(zero, zero) :: NzNat .
result Bool: false

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

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

The term `sd(zero, zero)` reduces to `zero`, which has least sort `Zero` but also its supersort `Nat`. The term `zero nil` has also sort `Zero` because, using the equational axioms for the `assoc` and `id: nil` attributes, `zero nil` is the same as `zero`, which has sort `Zero`.

Note that these membership predicates are polymorphic on sorts, not on kinds. This is because to be syntactically well-formed the argument term must be of the right kind, namely the connected component containing the sort being tested. Thus a membership at the kind level is either trivially true or a syntactic error. Also, the presence of the system truth values is required for the predicates to be meaningful, so they are only added to modules that import the module `TRUTH-VALUE` (which is included by default, as part of `BOOL`, unless the user specifies otherwise).

Advisory. In fact, the symbol `Universal` does not denote a real sort: it is instead a place holder for parsing purposes that is given an interpretation by the `polymorphic` attribute (see Section 4.4.4). The concrete effect of the interpretation of `Universal` is the instantiation in each connected component of the operators with one or more `Universal` arguments.

7.2 Natural numbers

The natural numbers module `NAT` provides a Peano-like specification of the natural numbers with an explicit successor function, while at the same time providing efficient built-in operators thanks to the `iter` theory (see Section 4.4.2) and an efficient binary representation of unbounded natural numbers arithmetic using the GNU GMP library.

The natural numbers sort hierarchy has top sort `Nat` and (disjoint) subsorts `Zero` and `NzNat`. The sort `Nat` is generated from the constant `0` (of sort `Zero`) and the successor operator `s_`.

```
fmod NAT is
  protecting BOOL .
  sorts Zero NzNat Nat .
  subsort Zero NzNat < Nat .
  *** constructors
```

```
  op 0 : -> Zero [ctor] .
  op s_ : Nat -> NzNat [ctor iter special (...)] .
```

Having 0 and successor as constructors means that you can define functions on the natural numbers by matching into the successor notation; for example:

```
fmod FACTORIAL is
  protecting NAT .
  op _! : Nat -> NzNat .
  var N : Nat .
  eq 0 ! = 1 .
  eq (s N) ! = (s N) * N ! .
endfm
```

Try entering this module into Maude and then entering the commands

```
Maude> red 100 ! .
Maude> red 1000 ! .
```

(The results are omitted; the first has 158 digits and the second 2568 digits.)

Natural numbers can be input, and by default will be output, in normal decimal notation; however `42` is just syntactic sugar for `s_^42(0)`. The command `set print number on/off` controls whether or not decimal notation is used by the pretty printer. Thus executing the command `set print number off` will cause numbers to be printed using iteration notation.

Most of the usual arithmetic operators are provided in `NAT`. They are not defined algebraically but could be given an algebraic definition by the user if desired, for example for theorem proving purposes.

```
  *** ARITHMETIC OPERATIONS
  *** addition
  op _+_ : NzNat Nat -> NzNat [assoc comm prec 33 special (...)] .
  op _+_ : Nat Nat -> Nat [ditto] .
  *** symmetric difference
  op sd : Nat Nat -> Nat [comm special (...)] .
  *** multiplication
  op _*_ : NzNat NzNat -> NzNat [assoc comm prec 31 special (...)] .
  op _*_ : Nat Nat -> Nat [ditto] .
  *** quotient
  op _quo_ : Nat NzNat -> Nat [prec 31 gather (E e) special (...)] .
  *** remainder
  op _rem_ : Nat NzNat -> Nat [prec 31 gather (E e) special (...)] .
  *** exponential n^m = n * .... * n (m times)
  op _^_ : Nat Nat -> Nat [prec 29  gather (E e) special (...)] .
  op _^_ : NzNat Nat -> NzNat [ditto] .
  *** exponential modulo  modExp(n,m,p) = n^m mod p
  op modExp : Nat Nat NzNat ~> Nat [special (...)] .
  *** greatest common divisor
  op gcd : NzNat NzNat -> NzNat [assoc comm special (...)] .
  op gcd : Nat Nat -> Nat [ditto] .
  *** least common multiple
  op lcm : NzNat NzNat -> NzNat [assoc comm special (...)] .
  op lcm : Nat Nat -> Nat [ditto] .
  *** minimum
  op min : NzNat NzNat -> NzNat [assoc comm special (...)] .
  op min : Nat Nat -> Nat [ditto] .
  *** maximum
  op max : NzNat Nat -> NzNat [assoc comm special (...)] .
```

```
op max : Nat Nat -> Nat [ditto] .
```

The operators _+_ and _*_ compute the usual addition and multiplication operations and _^_ is exponentiation.

The *symmetric difference* operator, `sd`, subtracts the smaller of its arguments from the larger. Thus, for example,

```
Maude> red in NAT : sd(4, 9) .
result NzNat: 5

Maude> red sd(9, 4) .
result NzNat: 5
```

The quotient and remainder operators, denoted `_quo_` and `_rem_`, satisfy the equation

$$((\texttt{i quo j}) * \texttt{j}) + (\texttt{i rem j}) = \texttt{i},$$

for natural numbers `i` and `j`. For example,

```
Maude> red in NAT : 11 quo 4 .
result NzNat: 2

Maude> red 11 rem 4 .
result NzNat: 3
```

The operator `modExp` computes modular exponentiation, with the third argument being the modulus. For example,

```
Maude> red in NAT : modExp(7, 1234, 2) .
result NzNat: 1

Maude> red modExp(8, 1234, 2) .
result Zero: 0
```

The operators `gcd`, `lcm`, `min`, and `max` compute the greatest common divisor, the least common multiple, the minimum and the maximum, respectively. Since these operators are associative and commutative, they can be used with any number (at least two) of arguments. For example,

```
Maude> red in NAT : gcd(6, 15, 21) .
result NzNat: 3

Maude> red lcm(6, 15, 21) .
result NzNat: 210

Maude> red min(6, 15, 21) .
result NzNat: 6

Maude> red max(6, 15, 21) .
result NzNat: 21

Maude> red gcd(0, 0) .
result Zero: 0
```

Operators that act on the binary representation of natural numbers interpreted as bit strings are:

- bitwise exclusive or (`_xor_`);

- bitwise and (_&_);
- bitwise or (_|_);
- rightshift—quotient by a power of 2 (_>>_); and
- leftshift—multiplication by a power of 2 (_<<_).

```
*** BITSTRING MANIPULATION
*** bitwise exclusive or
op _xor_ : Nat Nat -> Nat [assoc comm prec 55 special (...)] .
*** bitwise and
op _&_ : Nat Nat -> Nat [assoc comm prec 53 special (...)] .
*** bitwise or
op _|_ : NzNat Nat -> NzNat [assoc comm prec 57 special (...)] .
op _|_ : Nat Nat -> Nat [ditto] .
*** right shift -- quotient by power of 2
op _>>_ : Nat Nat -> Nat [prec 35 gather (E e) special (...)] .
*** left shift --  multiplication by power of 2
op _<<_ : Nat Nat -> Nat [prec 35 gather (E e) special (...)] .
```

Here are some examples using the bitwise operators.

```
Maude> red in NAT : 5 xor 7 .
result NzNat: 2

Maude> red 5 xor 2 .
result NzNat: 7

Maude> red 5 xor 5 .
result Zero: 0

Maude> red 5 & 7 .
result NzNat: 5

Maude> red 5 & 2 .
result Zero: 0

Maude> red 5 | 7 .
result NzNat: 7

Maude> red 5 | 2 .
result NzNat: 7

Maude> red 5 >> 2 .
result NzNat: 1

Maude> red 5 << 2 .
result NzNat: 20
```

The operators _<_, _<=_, _>_, and _>=_ denote the usual operations for comparing numbers: less than, less than or equal, greater than, and greater than or equal, respectively. The operator `divides` returns true if and only if its second argument is a multiple of the first one.

```
*** TESTS
*** n less than m
op _<_ : Nat Nat -> Bool [prec 37 special (...)] .
*** n less than or equal to m
op _<=_ : Nat Nat -> Bool [prec 37 special (...)] .
*** n greater than m
op _>_ : Nat Nat -> Bool [prec 37 special (...)] .
```

```
    *** n greater than or equal to m
    op _>=_ : Nat Nat -> Bool [prec 37 special (...)] .
    *** n divides m
    op _divides_ : NzNat Nat -> Bool [prec 51 special (...)] .
  endfm
```

Note that, to avoid producing negative numbers, subtraction and bitwise not are not provided. The symmetric difference can be used in place of subtraction.

The operational semantics for most of the built-in operators is such that you only get built-in behavior when all the arguments are actually natural numbers. The exception is associative and commutative built-in operators which will compute as much as possible on natural number arguments and leave the remaining arguments unchanged; for example,

```
Maude> red in NAT : gcd(gcd(12, X:Nat), 15) .
result Nat: gcd(X:Nat, 3)
```

If the built-in operator does not disappear using the built-in semantics, then user equations are tried.

Advisory. It is easy to overload your machine's memory by generating huge natural numbers. There is a limit on exponentiation in that built-in behavior will not occur if the first argument is greater than 1 and the second argument is too large. Similarly, leftshift does not work if the first argument is greater than or equal to 1 and the second argument is too large. Currently "too large" means greater than 1000000 but this may change. Modular exponentiation has no such limits as its built-in semantics takes advantage of the fact that the result cannot be larger than the modulus. This is likely to be useful for cryptographic algorithms.

7.3 Random numbers and counters

The functional module `RANDOM` adds to `NAT` a pseudo-random number generator:

```
fmod RANDOM is
  protecting NAT .
  op random : Nat -> Nat [special (...)] .
endfm
```

The function `random` is the mapping from `Nat` into the range of natural numbers $[0,2^{32} - 1]$ computed by successive calls to the Mersenne Twister Random Number Generator.[3] For example,

```
Maude> red in RANDOM : random(17) .
result NzNat: 1171049868
```

Although `random` is purely functional, it caches the state of the random number generator so that evaluating `random(0)` is always fast, as is evaluating `random(n+1)` if `random(n)` was the previous call to the operator `random`. In general, after generating `random(n)`, both `random(n)` and `random(n+1)` are computed efficiently because `random(n)` is a look up, while `random(n+k)` takes `k` steps of the twister or O(`k`) time.

By default the seed 0 is used, but a different seed, giving rise to a different function, may be specified by the command line option `-random-seed=`*n*, where *n* is a natural number in the range $[0,2^{32} - 1]$. For example, if we invoke the Maude interpreter with the option `-random-seed=42` and run the previous example again we get

```
Maude> red in RANDOM : random(17) .
```

```
result NzNat: 613608295
```

The predefined system module `COUNTER` adds a "counter" that can be used to generate new names and new random numbers in Maude programs that do not want to explicitly maintain this state.

```
mod COUNTER is
  protecting NAT .
  op counter : -> [Nat] [special (...)] .
endm
```

For the `rewrite` and `frewrite` commands (see Sections 5.4 and 18.2), 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. This module also imports the predefined module `CONVERSION`, described later in Section 7.9, which includes conversion functions between different types of numbers.

```
mod SAMPLER is
  pr RANDOM .
  pr COUNTER .
  pr CONVERSION .
  op rand : -> [Float] .
  op sampleBernoulli : Float -> [Bool] .
  rl rand => float(random(counter) / 4294967295) .
  rl sampleBernoulli(P:Float) => rand < P:Float .
endm
```

The first rule rewrites the constant `rand` to a floating point number between 0 and 1 pseudo-randomly chosen according to the uniform distribution. This floating point number is obtained by converting the rational number `random(counter) / 4294967295` into a floating point number, where `4294967295`= 2^{32} - 1 is the maximum value that the `random` function can attain. We can then use the uniform sampling of a number between 0 and 1 to sample the tossing of a coin with a given bias `P:Float` between 0 and 1. This is accomplished by the second rewrite rule in `SAMPLER`.

Sampling capabilities defined in this style for different probability distributions can then be used to specify *probabilistic models* in Maude. We can give a flavor for how such models can be simulated in Maude by means of a simple battery-operated clock example. We may represent the state of such a clock as a term `clock(T,C)`, with `T` a natural number denoting the time, and `C` a positive real denoting the amount of battery charge. Each time the clock ticks, the time is increased by one unit, and the battery charge slightly decreases; however, the lower the battery charge, the greater the chance that the clock will stop, going into a state of the form `broken(T,C)`. We can model this system by means of the following Maude specification:

```
mod CLOCK is
  pr SAMPLER .
  sort Clock .
  op clock : Nat Float -> Clock [ctor] .
  op broken : Nat Float -> Clock [ctor] .
  var T : Nat .
  var C : Float .
  rl clock(T,C)
    => if sampleBernoulli(C / 1000.0)
       then clock(s(T), C - (C / 1000.0))
       else broken(T, C)
       fi .
endm
```

This rule models the fact that each time the clock is going to tick a coin is tossed; if the result is `true`, then the clock ticks normally, but if the result is `false`, then the clock breaks down. If the battery charge is high enough, the bias of the coin will be highly towards normal ticking, but as the battery charge decreases, the bias gradually decreases, so that a breakdown becomes increasingly likely.

One can use a module such as `CLOCK` above to perform *Monte Carlo simulations* of the probabilistic system we are interested in. Of course, we want different arguments for the random number generator to be used each time from the same initial state so that we obtain different behaviors. In Maude this can be easily achieved within the same Maude session by typing the command

```
set clear rules off .
```

which turns off the automatic clearing of rule state information, including counter values (see Section 18.2). This means that when we run several times the same computation, a different counter value will be initially used each time, therefore getting different behaviors in the expected Monte Carlo way. For example, we get the following simulations for the behavior of a clock until it breaks:

```
Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
result Clock: broken(40, 9.607702107358117e+2)

Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
result Clock: broken(46, 9.5501998182355942e+2)

Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
result Clock: broken(16, 9.8411944181564002e+2)

Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
result Clock: broken(6, 9.9401498001499397e+2)

Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
result Clock: broken(28, 9.7237474437709557e+2)
```

Since it is reasonable for a program to use multiple counters, the safe way to do this is to import renamed copies of `COUNTER`; for example

```
protecting COUNTER * (op counter to counter2) .
```

Counters are inactive with respect to search, model checking, and equational rewriting. Notice that there are potentially bad interactions with the debugger (see Section 14.1.3) since another `rewrite/frewrite/erewrite` executed in the debugger will lose the counter state of the interrupted `rewrite/frewrite/erewrite`.

7.4 Integer numbers

The module `INT` extends `NAT` with a unary minus `-_` on nonzero natural numbers to construct the negative integers. Integers can be input, and by default are output, in normal decimal notation; however, `-42` is just an alternative concrete syntax for `- 42`, which itself is just an alternative concrete syntax for `-s_^42(0)`.

```
fmod INT is
  protecting NAT .
  sorts NzInt Int .
  subsorts NzNat < NzInt Nat < Int .

  op -_ : NzNat -> NzInt  [ctor special (...)] .
```

Unary minus is then extended to `Int` so that

```
- - I:Int = I:Int
- 0 = 0
```

The arithmetic operations of `NAT` are extended to integers. In addition, there are operators for subtraction, `_-_`, and absolute value, `abs`.

```
  *** ARITHMETIC OPERATIONS
  *** unary minus
  op -_ : NzInt -> NzInt [ditto] .
  op -_ : Int -> Int [ditto] .
  *** addition
  op _+_ : Int Int -> Int  [assoc comm prec 33 special (...)] .
  *** subtraction
  op _-_ : Int Int -> Int  [prec 33 gather (E e) special (...)] .
  *** multiplication
  op _*_ : NzInt NzInt -> NzInt [assoc comm prec 31 special (...)] .
  op _*_ : Int Int -> Int [ditto] .
  *** quotient
  op _quo_ : Int NzInt -> Int [prec 31 gather (E e) special (...)] .
  *** remainder
  op _rem_ : Int NzInt -> Int [prec 31 gather (E e) special (...)] .
  *** exponentiation
  op _^_ : Int Nat -> Int [prec 29  gather (E e) special (...)] .
  op _^_ : NzInt Nat -> NzInt [ditto] .
  *** absolute value
  op abs : NzInt -> NzNat [special (...)] .
  op abs : Int -> Nat [ditto] .
  *** greatest common divisor
  op gcd : NzInt NzInt -> NzNat [assoc comm special (...)] .
  op gcd : Int Int -> Nat [ditto] .
  *** least common multiple
  op lcm : NzInt NzInt -> NzNat [assoc comm special (...)] .
  op lcm : Int Int -> Nat [ditto] .
  *** minimum
  op min : NzInt NzInt -> NzInt [assoc comm special (...)] .
  op min : Int Int -> Int [ditto] .
  *** maximum
  op max : NzInt NzInt -> NzInt [assoc comm special (...)] .
  op max : Int Int -> Int [ditto] .
  op max : NzNat Int -> NzNat [ditto] .
  op max : Nat Int -> Nat [ditto] .
```

The operators _quo_ and _rem_ satisfy the same equation for integer arguments as for natural numbers. The sign of the quotient is the product of the signs of the arguments.

```
Maude> red in INT : -11 quo 4 .
result NzInt: -2

Maude> red 11 quo -4 .
result NzInt: -2

Maude> red  -11 quo -4 .
result NzNat: 2

Maude> red 11 rem -4 .
result NzNat: 3

Maude> red -11 rem 4 .
result NzInt: -3

Maude> red -11 rem -4 .
result NzInt: -3
```

Bitwise operations on negative integers use the 2's complement representation and the operator ~_, computing the bitwise not operation, is added.

```
  *** BITSTRING MANIPULATION (TWO'S COMPLEMENT)
  *** bitwise not
  op ~_ : Int -> Int [special (...)] .
  *** bitwise exclusive or
  op _xor_ : Int Int -> Int [assoc comm prec 55 special (...)] .
  *** bitwise and
  op _&_ : Nat Int -> Nat [assoc comm prec 53 special (...)] .
  op _&_ : Int Int -> Int [ditto] .
  *** bitwise or
  op _|_ : NzInt Int -> NzInt [assoc comm prec 57 special (...)] .
  op _|_ : Int Int -> Int [ditto] .
  *** rightshift
  op _>>_ : Int Nat -> Int [prec 35 gather (E e) special (...)] .
  *** leftshift
  op _<<_ : Int Nat -> Int [prec 35 gather (E e) special (...)] .
```

Tests on integers extend those on the natural numbers.

```
  *** TESTS
  *** less than
  op _<_ : Int Int -> Bool [prec 37 special (...)] .
  *** less than or equal
  op _<=_ : Int Int -> Bool [prec 37 special (...)] .
  *** greater than
  op _>_ : Int Int -> Bool [prec 37 special (...)] .
  *** greater than or equal
  op _>=_ : Int Int -> Bool [prec 37 special (...)] .

  op _divides_ : NzInt Int -> Bool [prec 51 special (...)] .
endfm
```

Let us show with an example how a predefined module can be reused to define new subsorts that refine the sort structure of the data type. In the following example, we introduce additional subsorts and overload the successor operator s_ (originally coming from the module NAT imported in protecting

mode into INT) in order to specify the sort of integers greater than three.

```
fmod INT-GT-3 is
  protecting INT .
  sorts One Two Three IntGt3 .
  subsorts One Two Three IntGt3 < NzNat .
  op s_ : Zero -> One [ctor ditto] .
  op s_ : One -> Two [ctor ditto] .
  op s_ : Two -> Three [ctor ditto] .
  op s_ : Three -> IntGt3 [ctor ditto] .
  op s_ : IntGt3 -> IntGt3 [ctor ditto] .
endfm
```

We can check the sort of a number by “reducing” the corresponding constant, as follows:

```
Maude> red -1 .
result NzInt: -1

Maude> red 0 .
result Zero: 0

Maude> red 1 .
result One: 1

Maude> red 2 .
result Two: 2

Maude> red 3 .
result Three: 3

Maude> red 4 .
result IntGt3: 4

Maude> red 12345678901234567890 .
result IntGt3: 12345678901234567890
```

In theory, the sort of integers greater than three could also be specified by means of membership axioms (see Sections 4.2 and 4.3). However, memberships are not guaranteed to work correctly with the number hierarchy, because of the special internal representation for iterated towers of s_ symbols.

7.5 Machine integers

Versions of Maude prior to 2.0 supported machine integers in place of arbitrary size integers. Initially they were 32-bit in Maude 1.0 but were increased to 64-bit in Maude 1.0.5.

For certain applications, such as specifying programming languages that support machine integers as a built-in data type, it is convenient to have a predefined specification for machine integers. Fortunately, it is straightforward to efficiently emulate machine integers in terms of arbitrary size integers.

First we rename a copy of the regular integers, giving the sorts new names consistent with the new semantics and renaming those operators that either will not be defined on machine integers or else will have new semantics. Note that the operators ~_, _&_, _|_, _<_, _<=_, _>_, and _=>_ are not modified by the renaming.

```
fmod RENAMED-INT is
  protecting INT * (sort Zero to MachineZero,
```

```
                        sort NzNat to NzMachineNat,
                        sort Nat to MachineNat,
                        sort NzInt to NzMachineInt,
                        sort Int to MachineInt,
                        op s_ : Nat -> NzNat to $succ,
                        op sd : Nat Nat -> Nat to $sd,
                        op -_ : Int -> Int to $neg,
                        op _+_ : Int Int -> Int to $add,
                        op _-_ : Int Int -> Int to $sub,
                        op _*_ : NzInt NzInt -> NzInt to $mult,
                        op _quo_ : Int NzInt -> Int to $quo,
                        op _rem_ : Int NzInt -> Int to $rem,
                        op _^_ : Int Nat -> Int to $pow,
                        op abs : NzInt -> NzNat to $abs,
                        op gcd : NzInt Int -> NzNat to $gcd,
                        op lcm : NzInt NzInt -> NzNat to $lcm,
                        op min : NzInt NzInt -> NzInt to $min,
                        op max : NzInt NzInt -> NzInt to $max,
                        op _xor_ : Int Int -> Int to $xor,
                        op _>>_ : Int Nat -> Int to  $shr,
                        op _<<_ : Int Nat -> Int to $shl,
                        op _divides_ : NzInt Int -> Bool to $divides) .
  endfm
```

We then give a parameter theory that specifies the number of bits in a machine integer, which must be a power of 2, greater or equal to 2. Notice that this theory is based on the previous module, which is imported in `protecting` mode. Therefore, `$nrBits` is a *parameter constant* ranging over the `NzMachineNat` sort in the `RENAMED-INT` module, which is imported with an initial algebra semantics.

```
  fth BIT-WIDTH is
    protecting RENAMED-INT .
    op $nrBits : -> NzMachineNat .

    var N : NzMachineNat .
    eq $divides(2, $nrBits) = true [nonexec] .
    ceq $divides(2, N) = true
      if $divides(N, $nrBits) /\ N > 1 [nonexec] .
  endfth
```

Also provided are two predefined views that set the number of bits value `$nrBits` respectively to 32 and 64, the two most common sizes.

```
  view 32-BIT from BIT-WIDTH to RENAMED-INT is
    op $nrBits to term 32 .
  endv

  view 64-BIT from BIT-WIDTH to RENAMED-INT is
    op $nrBits to term 64 .
  endv
```

The module `MACHINE-INT` takes a bit width parameter and defines those operations that have a new semantics when applied to machine integers. In many cases this means applying the operation `$wrap` to the results to correctly simulate the wrap-around effect over an overflow on signed fixed bit width integers by, in effect, extending the sign bit infinitely to the left. In the case of `_^_` the meaning of the operation changes to exclusive or (from exponentiation on arbitrary size integers).

```
  fmod MACHINE-INT{X :: BIT-WIDTH} is
```

```
    vars I J : MachineInt .
    var K : NzMachineInt .

    op $mask : -> NzMachineInt [memo] .
    eq $mask = $sub($nrBits, 1) .

    op $sign : -> NzMachineInt [memo] .
    eq $sign = $pow(2, $mask) .

    op maxMachineInt : -> NzMachineInt [memo] .
    eq maxMachineInt = $sub($sign, 1) .

    op minMachineInt : -> NzMachineInt [memo] .
    eq minMachineInt = $neg($sign) .

    op $wrap : MachineInt -> MachineInt .
    eq $wrap(I) = (I & maxMachineInt) | $neg(I & $sign) .

    op _+_ : MachineInt MachineInt -> MachineInt
        [assoc comm prec 33] .
    eq I + J = $wrap($add(I, J)) .

    op -_ : MachineInt -> MachineInt .
    eq - I = $wrap($neg(I)) .

    op _-_ : MachineInt MachineInt -> MachineInt
        [prec 33 gather (E e)] .
    eq I - J = $wrap($sub(I, J)) .

    op _*_ : MachineInt MachineInt -> MachineInt
        [assoc comm prec 31] .
    eq I * J = $wrap($mult(I, J)) .

    op _/_ : MachineInt NzMachineInt -> MachineInt
        [prec 31 gather (E e)] .
    eq I / K = $wrap($quo(I, K)) .

    op _%_ : MachineInt NzMachineInt -> MachineInt
        [prec 31 gather (E e)] .
    eq I % K = $rem(I, K) .

    op _^_ : MachineInt MachineInt -> MachineInt
        [prec 55 gather (E e)] .
    eq I ^ J = $xor(I, J) .

    op _>>_ : MachineInt MachineInt -> MachineInt
        [prec 35 gather (E e)] .
    eq I >> J = $shr(I, ($mask & J)) .

    op _<<_ : MachineInt MachineInt -> MachineInt
        [prec 35 gather (E e)] .
    eq I << J = $wrap($shl(I, ($mask & J))) .
endfm
```

Notice that using out of range integer constants may cause incorrect results.

We consider now the instantiation with the predefined view `32-BIT`, and show the wrap-around effect in several examples.

```
fmod MACHINE-INT-TEST is
```

```
  protecting MACHINE-INT{32-BIT} .
endfm
```

In the first examples, we can see the wrap-around from negative to positive and vice versa:

```
Maude> red -2147483648 - 1 .
result NzMachineNat: 2147483647

Maude> red 2147483647 + 1 .
result NzMachineInt: -2147483648
```

In the following product, the negative case does not wrap-around but the positive case does:

```
Maude> red -1073741824 * 2 .
result NzMachineInt: -2147483648

Maude> red 1073741824 * 2 .
result NzMachineInt: -2147483648
```

Division can only cause a wrap-around in this one case:

```
Maude> red -2147483648 / -1 .
result NzMachineInt: -2147483648
```

Remainder never wraps around:

```
Maude> red -2147483648 % -1 .
result MachineZero: 0
```

Finally, we see that the sign bit “falls off the left end” in a left shift:

```
Maude> red -2147483648 << 1 .
result MachineZero: 0
```

The parameterized `MACHINE-INT` module is an interesting example of Maude’s support for what in type theory are called *dependent types* (see, for example, [79]). These are types like the power type $X^{[n]}$ or the `ARRAY{X,[n]}` type depending on a data parameter `n`, for example a natural number. We can view `MACHINE-INT` as the Maude analogue of a dependent type definition; however, note that the data parameter is not just any nonzero natural number, but must also satisfy *additional axioms*, specified in the `BIT-WIDTH` theory. For two other interesting examples of a Maude parameterized module defining the analogue of a dependent type, see the `POWER[n]` module in Section 15.3.1 (the exact analogue of the power type $X^{[n]}$) and the `NAT/{N}` module of natural numbers modulo `N` in Section 17.7. Similarly, the `TUPLE[n]` module in Section 15.3.1 provides a form of dependent type that is not even available in some type theories with dependent types.

7.6 Rational numbers

The module `RAT` extends `INT` with a binary division operator `_/_` to construct the rationals from integers and nonzero naturals. Rationals can be input, and by default are output, in normal decimal notation; however `-5/42` is equivalent to `-5 / 42`, which is equivalent to `- 5 / 42`, which really denotes `- s_^5(0) / s_^42(0)`. The command

```
set print rat off .
```

switches off the special printing for `_/_` so that rational numbers will be printed with spaces around the foreslash sign. Note that `set print number off` also affects the printing of rational numbers, so with both number and rational pretty-printing switches turned off `-5/42` is printed using the final notation given above.

The numerator and denominator of a rational may contain common factors but these are removed by a single built-in rewrite whenever the rational is reduced (thus `_/_` is *not* a free constructor).

Notice that, in addition to the subsort `NzRat` of nonzero rational numbers, there is a subsort `PosRat` of positive rational numbers.

```
fmod RAT is
  protecting INT .
  sorts PosRat NzRat Rat .
  subsorts NzInt < NzRat Int < Rat .
  subsorts NzNat < PosRat < NzRat .

  op _/_ : NzInt NzNat -> NzRat
      [ctor prec 31 gather (E e) special (...)] .
  vars I J : NzInt .
  vars N M : NzNat .
  var  K : Int .
  var  Z : Nat .
  var  Q : NzRat .
  var  R : Rat .
```

The basic arithmetic operations on integers are extended to rational numbers as usual. The operator `_/_` is declared special for the case when the first argument is of sort `NzInt` to enhance performance. The remaining operators are defined in Maude by equations and may do some rewriting even when their arguments are not properly constructed rationals. Note that the choice of equations for defining operators on the rationals is motivated by performance: simpler equations are possible in many cases but they turn out to incur a big performance penalty.

```
  *** ARITHMETIC OPERATIONS
  op _/_ : NzNat NzNat -> PosRat [ctor ditto] .
  op _/_ : PosRat PosRat -> PosRat [ditto] .
  op _/_ : NzRat NzRat -> NzRat [ditto] .
  op _/_ : Rat NzRat -> Rat [ditto] .
  eq 0 / Q = 0 .
  eq I / - N = - I / N .
  eq (I / N) / (J / M) = (I * M) / (J * N) .
  eq (I / N) / J = I / (J * N) .
  eq I / (J / M) = (I * M) / J .

  op -_ : NzRat -> NzRat [ditto] .
  op -_ : Rat -> Rat [ditto] .
  eq - (I / N) = - I / N .

  op _+_ : PosRat PosRat -> PosRat [ditto] .
  op _+_ : PosRat Nat -> PosRat [ditto] .
  op _+_ : Rat Rat -> Rat [ditto] .
  eq I / N + J / M = (I * M + J * N) / (N * M) .
  eq I / N + K = (I + K * N) / N .

  op _-_ : Rat Rat -> Rat [ditto] .
  eq I / N - J / M = (I * M - J * N) / (N * M) .
  eq I / N - K = (I - K * N) / N .
  eq K - J / M = (K * M - J ) / M .
```

```
op _*_ : PosRat PosRat -> PosRat [ditto] .
op _*_ : NzRat NzRat -> NzRat [ditto] .
op _*_ : Rat Rat -> Rat [ditto] .
eq Q * 0 = 0 .
eq (I / N) * (J / M) = (I * J) / (N * M).
eq (I / N) * K = (I * K) / N .

op _^_ : PosRat Nat -> PosRat [ditto] .
op _^_ : NzRat Nat -> NzRat [ditto] .
op _^_ : Rat Nat -> Rat [ditto] .
eq (I / N) ^ Z = (I ^ Z) / (N ^ Z) .

op abs : NzRat -> PosRat [ditto] .
op abs : Rat -> Rat [ditto] .
eq abs(I / N) = abs(I) / N .
```

The integer operations `quo`, `rem`, `gcd`, `lcm`, `min`, and `max` are also extended to the rational numbers. The operator `quo` gives the number of whole times a rational can be divided by another, `rem` gives the rational remainder. The operator `gcd` returns the largest rational that divides into each of its arguments a whole number of times, while `lcm` returns the smallest rational that is an integer multiple of its arguments.

```
op _quo_ : PosRat PosRat -> Nat [ditto] .
op _quo_ : Rat NzRat -> Int [ditto] .
eq (I / N) quo Q = I quo (N * Q) .
eq K quo (J / M) = (K * M) quo J .

op _rem_ : Rat NzRat -> Rat [ditto] .
eq (I / N) rem (J / M) = ((I * M) rem (J * N)) / (N * M) .
eq K rem (J / M) = ((K * M) rem J) / M .
eq (I / N) rem J = (I rem (J * N)) / N .

op gcd : NzRat Rat -> PosRat [ditto] .
op gcd : Rat Rat -> Rat [ditto] .
eq gcd(I / N, R) = gcd(I, N * R) / N .

op lcm : NzRat NzRat -> PosRat [ditto] .
op lcm : Rat Rat -> Rat [ditto] .
eq lcm(I / N, R) = lcm(I, N * R) / N .

op min : PosRat PosRat -> PosRat [ditto] .
op min : NzRat NzRat -> NzRat [ditto] .
op min : Rat Rat -> Rat [ditto] .
eq min(I / N, R) = min(I, N * R) / N .

op max : PosRat Rat -> PosRat [ditto] .
op max : NzRat NzRat -> NzRat [ditto] .
op max : Rat Rat -> Rat [ditto] .
eq max(I / N, R) = max(I, N * R) / N .
```

Some examples involving these operations are the following:

```
Maude> red in RAT : 1/2 quo 1/3 .
result NzNat: 1

Maude> red 1/2 rem 1/3 .
result PosRat: 1/6
```

```
Maude> red gcd(1/2, 1/3) .
result PosRat: 1/6

Maude> red lcm(1/2, 1/3) .
result NzNat: 1
```

Tests on integers are extended to rational numbers. The test `divides` returns true if a rational number divides another rational number a whole number of times.

```
*** tests
op _<_ : Rat Rat -> Bool [ditto] .
eq (I / N) < (J / M) = (I * M) < (J * N) .
eq (I / N) < K = I < (K * N) .
eq K < (J / M) = (K * M) < J .

op _<=_ : Rat Rat -> Bool [ditto] .
eq (I / N) <= (J / M) = (I * M) <= (J * N) .
eq (I / N) <= K = I <= (K * N) .
eq K <= (J / M) = (K * M) <= J .

op _>_ : Rat Rat -> Bool [ditto] .
eq (I / N) > (J / M) = (I * M) > (J * N) .
eq (I / N) > K = I > (K * N) .
eq K > (J / M) = (K * M) > J .

op _>=_ : Rat Rat -> Bool [ditto] .
eq (I / N) >= (J / M) = (I * M) >= (J * N) .
eq (I / N) >= K = I >= (K * N) .
eq K >= (J / M) = (K * M) >= J .

op _divides_ : NzRat Rat -> Bool [ditto] .
eq (I / N) divides K = I divides N * K .
eq Q divides (J / M) = Q * M divides J .
```

There are four new operators: `trunc`, `frac`, `floor`, and `ceiling`. The operator `floor` converts a rational number to an integer by rounding down to the nearest integer, `ceiling` rounds up, and `trunc` rounds towards 0. The operator `frac` gives the fraction part of its argument and this always has the same sign as its argument.

```
*** ROUNDING
op trunc : PosRat -> Nat .
op trunc : Rat -> Int .
eq trunc(K) = K .
eq trunc(I / N) = I quo N .

op frac : Rat -> Rat .
eq frac(K) = 0 .
eq frac(I / N) = (I rem N) / N .

op floor : PosRat -> Nat .
op floor : Rat -> Int .
eq floor(K) = K .
eq floor(N / M) = N quo M .
eq floor(- N / M) = - ceiling(N / M) .

op ceiling : PosRat -> NzNat .
op ceiling : Rat -> Int .
eq ceiling(K) = K .
eq ceiling(N / M) = ((N + M) - 1) quo M .
```

```
  eq ceiling(- N / M) = - floor(N / M) .
endfm
```

Here are some examples of reductions involving the rounding operators:

```
Maude> red in RAT : trunc(9/7) .
result NzNat: 1

Maude> red floor(9/7) .
result NzNat: 1

Maude> red ceiling(9/7) .
result NzNat: 2

Maude> red frac(9/7) .
result PosRat: 2/7

Maude> red trunc(-9/7) .
result NzInt: -1

Maude> red floor(-9/7) .
result NzInt: -2

Maude> red ceiling(-9/7) .
result NzInt: -1

Maude> red  frac(-9/7) .
result NzRat: -2/7
```

7.7 Floating-point numbers

The module `FLOAT` declares sorts and operators for manipulating floating-point numbers, which are implemented using double precision floating-point arithmetic of the underlying hardware platform, conforming to the IEEE-754 standard when supported by the hardware platform. Floating-point numbers are treated as a large set of constants, that is, a floating-point number has no algebraic structure (this is the reason for the special operator declaration `<Floats>`, as explained in the introduction of this chapter).

The sort `FiniteFloat` consists of the floating-point numbers that have a 64-bit representation. Finite floating-point numbers can be input, and by default are output, in scientific notation; they can also be input using decimal point notation. Thus `100.0` is equivalent to `1.0e+2`. The constants `Infinity` and `-Infinity` represent floating-point numbers that are outside the 64-bit representable range. Thus `Infinity` and `-Infinity` are of sort `Float` but not of sort `FiniteFloat`. Note that there are some surprises when using decimal notation to input floating-point numbers. For example, in the `FLOAT` module we have the reduction

```
Maude> red in FLOAT : 1.1 .
result FiniteFloat: 1.1000000000000001
```

This is because floating-point numbers are represented internally using a binary expansion rather than a decimal expansion and 1.1 does not have a finite length binary expansion.

```
fmod FLOAT is
  protecting BOOL .
  sorts FiniteFloat Float .
  subsort FiniteFloat < Float .
  op <Floats> : -> FiniteFloat [special (...)] .
```

```
op <Floats> : -> Float [ditto] .
```

The arithmetic operators -_, _-_, _+_, _*_, _/_, _^_, and abs have the usual interpretation, as in the module INT. Note that 1.2 / 0.0 is just an expression of kind [Float] and reducing it does not cause your system to crash!

```
*** ARITHMETIC OPERATIONS
op -_ : Float -> Float [prec 15 special (...)] .
op -_ : FiniteFloat -> FiniteFloat [ditto] .

op _+_ : Float Float -> Float
    [prec 33 gather (E e) special (...)] .
op _-_ : Float Float -> Float
    [prec 33 gather (E e) special (...)] .
op _*_ : Float Float -> Float
    [prec 31 gather (E e) special (...)] .
op _/_ : Float Float ~> Float
    [prec 31 gather (E e) special (...)] .
op _^_ : Float Float ~> Float
    [prec 29  gather (E e) special (...)] .

op abs : Float -> Float [special (...)] .
op abs : FiniteFloat -> FiniteFloat [ditto] .
```

The operator _rem_ computes the remainder of a division, floor rounds down to the nearest integer, ceiling rounds up, and sqrt computes the square root.

```
op _rem_ : Float Float ~> Float
    [prec 31 gather (E e)  special (...)] .
op floor : Float -> Float [special (...)] .
op ceiling : Float -> Float [special (...)] .
op sqrt : Float ~> Float [special (...)] .
```

For terms f1 and f2 of sort FiniteFloat, f1 rem f2 computes the remainder of dividing f1 by f2. Specifically, f1 rem f2 is equal to f1 - n * f2, where n is f1 / f2 rounded towards zero to the nearest integer. For example,

```
Maude> red in FLOAT : 5.0 rem 2.0 .
result FiniteFloat: 1.0

Maude> red -5.0 rem 2.0  .
result FiniteFloat: -1.0

Maude> red 5.0 rem 2.5 .
result FiniteFloat: 0.0
```

Some examples of reductions using the floor and ceiling operations are the following:

```
Maude> red in FLOAT : ceiling(2.5) .
result FiniteFloat: 3.0

Maude> red floor(2.5) .
result FiniteFloat: 2.0

Maude> red ceiling(- 2.5) .
result FiniteFloat: -2.0

Maude> red floor(- 2.5) .
```

```
result FiniteFloat: -3.0

Maude> red ceiling(Infinity) .
result Float: Infinity

Maude> red floor(-Infinity) .
result Float: -Infinity
```

The operators `max` and `min` for computing the maximum and the minimum, respectively, work as expected,

```
op min : Float Float -> Float [special (...)] .
op max : Float Float -> Float [special (...)] .
```

as we can see in the following examples:

```
Maude> red in FLOAT : min(2.0, -2.0) .
result FiniteFloat: -2.0

Maude> red max(2.0, -2.0) .
result FiniteFloat: 2.0

Maude> red max(2.0, Infinity) .
result Float: Infinity

Maude> red in FLOAT : min(Infinity, -Infinity) .
result Float: -Infinity
```

The operators `exp` and `log` compute the natural exponent and logarithm, respectively.

```
*** TRANSCENDENTAL OPERATIONS
op exp : Float -> Float [special (...)] .
op log : Float ~> Float [special (...)] .
```

Here are some examples:

```
Maude> red in FLOAT : exp(1.0) .
result FiniteFloat: 2.7182818284590451

Maude> red log(exp(1.0)) .
result FiniteFloat: 1.0

Maude> red log(0.0) .
result Float: -Infinity
```

The constant `pi` approximates the value of π. The number of digits is chosen to be the largest that can accurately be represented as a floating-point number. The trigonometric operators `sin`, `cos`, and `tan` expect arguments in radians. The operators `asin`, `acos`, `atan` are the corresponding inverses.

```
*** TRIGONOMETRIC OPERATIONS
op sin : Float -> Float [special (...)] .
op cos : Float -> Float [special (...)] .
op tan : Float -> Float [special (...)] .
op asin : Float ~> Float [special (...)] .
op acos : Float ~> Float [special (...)] .
op atan : Float -> Float [special (...)] .
op atan : Float Float -> Float [special (...)] .
```

```
op pi : -> FiniteFloat .
eq pi = 3.1415926535897931 .
```

Here are some examples of reductions of trigonometric expressions.

```
Maude> red in FLOAT : sin(0.0) .
result FiniteFloat: 0.0

Maude> red sin(pi) .
result FiniteFloat: 1.2246467991473532e-16

Maude> red cos(pi) .
result FiniteFloat: -1.0

Maude> red acos(cos(pi)) .
result FiniteFloat: 3.1415926535897931

Maude> red tan(pi) .
result FiniteFloat: -1.2246467991473532e-16

Maude> red sin(pi / 2.0) .
result FiniteFloat: 1.0

Maude> red cos(pi / 2.0) .
result FiniteFloat: 6.123233995736766e-17

Maude> red tan(pi / 2.0) .
result FiniteFloat: 1.633123935319537e+16

Maude> red atan(tan(pi / 2.0)) .
result FiniteFloat: 1.5707963267948966

Maude> red pi / 2.0 .
result FiniteFloat: 1.5707963267948966
```

Using the binary form of the arc tangent operator, `atan(f1, f2)`, is similar to computing `atan(f1 / f2)`, except that the signs of both arguments are used to control the quadrant of the result.

```
Maude> red in FLOAT : atan(tan(pi / 3.0)) .
result FiniteFloat: 1.0471975511965976

Maude> red atan(tan(pi / 3.0), 1.0) .
result FiniteFloat: 1.0471975511965976

Maude> red atan(tan(pi / 3.0), -1.0) .
result FiniteFloat: 2.0943951023931957

Maude> red atan(- tan(pi / 3.0), -1.0) .
result FiniteFloat: -2.0943951023931957

Maude> red atan(- tan(pi / 3.0), 1.0) .
result FiniteFloat: -1.0471975511965976
```

Numerical comparisons have the usual meaning on floating-point numbers.

```
*** TESTS
op _<_ : Float Float -> Bool [prec 51 special (...)] .
op _<=_ : Float Float -> Bool [prec 51 special (...)] .
op _>_ : Float Float -> Bool [prec 51 special (...)] .
```

```
  op _>=_ : Float Float -> Bool [prec 51 special (...)] .

  *** approximate equality
  op _=[_]_ : Float FiniteFloat Float -> Bool [prec 51] .
  vars X Y : Float .
  var  Z : FiniteFloat .
  eq X =[Z] Y = abs(X - Y) < Z .
endfm
```

The operator `_=[_]_` tests for approximate equality, where the second argument bounds the allowed error. For example:

```
Maude> red in FLOAT : 1.111111111 =[1.0e-9]  1.111111112 .
result Bool: true

Maude> red 1.111111111 =[1.0e-10] 1.111111112 .
result Bool: false
```

7.8 Strings

The module `STRING` declares sorts and operators for manipulating strings of characters. Strings of length one form a subsort `Char` of `String`. Operations on strings are based on the SGI rope package [9], 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 [70, Section A2.5.2].

To define the results of searching a string for an occurrence of another substring the sort `FindResult` is introduced. This sort consists of the natural numbers, returned as the index in the string where a found substring begins (string indexing begins with 0), and a special constant `notFound`, returned if no occurrence is found.

```
fmod STRING is
  protecting NAT .
  sorts String Char FindResult .
  subsort Char < String .
  subsort Nat < FindResult .
  op <Strings> : -> Char [special (...)] .
  op <Strings> : -> String [ditto] .
  op notFound : -> FindResult [ctor] .
```

The operators `ascii` and `char` convert between characters and ASCII codes.

```
  *** conversion between ascii code and character
  op ascii : Char -> Nat [special (...)] .
  op char : Nat ~> Char [special (...)] .
```

For a natural number `n` less than 256 and a character `c`, we have `ascii(char(n))` = `n` and `char(ascii(c))` = `c`. For a natural number `n` greater than 255, `char(n)` is an error term of kind `[String]`. For example,

```
Maude> red in STRING : ascii("#") .
result NzNat: 35
```

```
Maude> red char(35) .
result Char: "#"

Maude> red ascii("a") .
result NzNat: 97

Maude> red  char(97) .
result Char: "a"

Maude> red char(255) .
result Char: "\377"
```

On strings, _+_ denotes the concatenation operation, with identity the empty string, `""`. String length is computed by the `length` operator.

```
*** string concatenation
op _+_ : String String -> String
    [prec 33 gather (E e) special (...)] .

*** string length
op length : String -> Nat [special (...)] .
```

Here are some examples.

```
Maude> red in STRING : "abc" + "def" .
result String: "abcdef"

Maude> red "ab" + "cd" + "ef" .
result String: "abcdef"

Maude> red "abc" + "" .
result String: "abc"

Maude> red length("abcdef") .
result NzNat: 6

Maude> red length("") .
result Zero: 0
```

The operators `substr`, `find`, and `rfind` deal with finding and extracting substrings. Remember that string indexing begins with 0.

```
*** substring
*** second argument is starting position, third is length
op substr : String Nat Nat -> String [special (...)] .

*** starting position of substring (second argument)
*** least one >= third argument (find)
*** greatest one <= third argument (rfind)
op find : String String Nat -> FindResult [special (...)] .
op rfind : String String Nat -> FindResult [special (...)] .
```

The expression `substr(S:String, Start:Nat, Len:Nat)` returns the substring of `S:String` of length `Len:Nat` beginning at position `Start:Nat`. If the value of the term `Start:Nat + Len:Nat` is greater than `length(S:String)` then the returned substring is the tail of `S:String` starting from position `Start:Nat`. This will be empty if the starting position is past the end of the string.

```
Maude> red in STRING : substr("abc", 0, 2) .
```

```
result String: "ab"

Maude> red substr("abc", 1, 2) .
result String: "bc"

Maude> red substr("abc", 1, 3) .
result String: "bc"

Maude> red substr("abc", 3, 2) .
result String: ""
```

`find` searches for the first match from the beginning of the string, while `rfind` searches from the end of the string backwards.

`find(S:String, Pat:String, Start:Nat)` returns the least index of an occurrence of `Pat:String` in `S:String` that is greater than or equal to `Start:Nat`. If no such index exists the constant `notFound` is returned.

`rfind(S:String, Pat:String, Start:Nat)` returns the greatest index of an occurrence of `Pat:String` in `S:String` that is less than or equal to `Start:Nat`. If no such index exists the constant `notFound` is returned.

```
Maude> red in STRING : find("abc", "b", 0) .
result NzNat: 1

Maude> red find("abc", "b", 1) .
result NzNat: 1

Maude> red find("abc", "b", 2) .
result FindResult: notFound

Maude> red find("abc", "d", 2) .
result FindResult: notFound

Maude> red rfind("abc", "b", 2) .
result NzNat: 1

Maude> red rfind("abc", "b", 1) .
result NzNat: 1

Maude> red rfind("abc", "b", 0) .
result FindResult: notFound

Maude> red rfind("abc", "d", 2) .
result FindResult: notFound
```

Some properties relating `substr`, `find`, and `rfind` are the following, where *S* and *P* are variables of sort `String`, and *I*, *J*, and *K* are variables of sort `Nat` such that `length(`*S*`)` = *K* and `length(`*P*`)` = *J*.

$$I \leq \texttt{find}(S,P,I) \leq K\texttt{-}J$$

$$0 \leq \texttt{rfind}(S,P,I) \leq \texttt{min}(I, K\texttt{-}J)$$

$$\texttt{find}(S,S,0) = 0 = \texttt{rfind}(S,S,I)$$

$$\texttt{find}(S,\texttt{""},I) = \mathit{if}\ I \leq K\ \mathit{then}\ I\ \mathit{else}\ \texttt{notFound}$$

$$\texttt{rfind}(S,\texttt{""},I) = \mathit{if}\ I \geq K\ \mathit{then}\ K\ \mathit{else}\ I$$

$$\texttt{find}(S,P,I) \neq \texttt{notFound} \implies \texttt{substr}(S,0,\texttt{find}(S,P,I))\ \texttt{+}\ P\ \texttt{+}\ \texttt{substr}(S,\texttt{find}(S,P,I)\texttt{+}J,K) = S$$

$$\texttt{rfind}(S,P,I) \neq \texttt{notFound} \implies \texttt{substr}(S,0,\texttt{rfind}(S,P,I))\ \texttt{+}\ P\ \texttt{+}\ \texttt{substr}(S,\texttt{rfind}(S,P,I)\texttt{+}J,K) = S$$

The operators _<_, _<=_, _>_, and _>=_ denote string comparison operations using the lexicographic order, where characters are compared going through their ASCII codes.

```
    *** lexicographic string comparison
    op _<_ : String String -> Bool [prec 37 special (...)] .
    op _<=_ : String String -> Bool [prec 37 special (...)] .

    op _>_ : String String -> Bool [prec 37  special (...)] .
    op _>=_ : String String -> Bool [prec 37 special (...)] .
  endfm
```

Here are some examples.

```
Maude> red in STRING : "abc" < "abd" .
result Bool: true

Maude> red "abc" < "abb" .
result Bool: false

Maude> red "abc" < "abcd" .
result Bool: true
```

7.9 String and number conversions

The module `CONVERSION` consolidates all the conversion functions between the three major built-in data types: `Nat/Int/Rat`, `Float`, and `String`.

```
fmod CONVERSION is
  protecting RAT .
  protecting FLOAT .
  protecting STRING .

  *** number type conversions
  op float : Rat -> Float [special (...)] .
  op rat : FiniteFloat -> Rat [special (...)] .
```

The operation `float` computes the floating-point number nearest to a given rational number. If the value of the rational number falls outside the range representable by IEEE-754 double precision finite floating-point numbers, `Infinity` or `-Infinity` is returned as appropriate. This is in accord with the convention that `Infinity` and `-Infinity` are used to handle out-of-range situations in the floating-point world.

The operator `rat` converts finite floating-point numbers to rational numbers exactly (since every IEEE-754 finite floating-point number is a rational number). Of course, if the result happens to be a natural number or an integer, that is what you get. `rat(Infinity)` and `rat(-Infinity)` do not reduce, since they have no reasonable representation in the world of rational numbers. It is intended that the equation

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

is satisfied, although this holds only if the third party library (GNU GMP) being used in the implementation meets its related requirements.

```
*** string <-> number conversions
op string : Rat NzNat ~> String [special (...)] .
op rat : String NzNat ~> Rat [special (...)] .
op string : Float -> String [special (...)] .
op float : String ~> Float [special (...)] .
```

The operator `string` converts a rational number to a string using a given base, which must lie in the range 2..36. Rational numbers that are really natural numbers or integers are converted to string representations of natural numbers or integers, so we have for example

```
Maude> red in CONVERSION : string(-1, 10) .
result String: "-1"
```

The operator `rat` converts a string to a rational number using a given base, which must lie in the range 2..36. Of course, if the result happens to be a natural number or an integer, that is what you get. Currently the function is very strict about which strings are converted: the string must be something that the Maude parser would recognize as a natural number, an integer or a rational number. This could be changed to a more generous interpretation in the future.

The operators `string` and `float` for conversion between floating-point numbers and strings satisfy the equation

```
float(string(F:Float)) = F:Float
```

A new sort, `DecFloat`, is introduced to provide the means for arbitrary formatting of floating-point numbers.

```
sort DecFloat .
op <_,_,_> : Int String Int -> DecFloat [ctor] .
op decFloat : Float Nat -> DecFloat [special (...)] .
endfm
```

A `DecFloat` consists of a sign (1, 0 or -1), a string of digits, and a decimal point position (0 is just in front of first digit, -*n* is *n* positions to the left, and +*n* is *n* positions to the right). Thus, `< -1, "123", 11 >` represents `-1.23e10`. `decFloat(F, N)` converts `F` to a `DecFloat`, rounding to `N` significant digits using the IEEE-754 "round to nearest" rule with trailing zeros if needed. If `N` is 0, an *exact* `DecFloat` representation of `F` is produced—this may require hundreds of digits. For any natural number `N`, `decFloat(Infinity, N)` reduces to `< 1, "Infinity", 0 >`. Here are some examples.

```
Maude> red in CONVERSION : decFloat(Infinity, 9) .
result DecFloat: < 1,"Infinity",0 >

Maude> red decFloat(-Infinity, 9) .
result DecFloat: < -1,"Infinity",0 >
```

```
Maude> red decFloat(123.0, 5) .
result DecFloat: < 1,"12300",3 >

Maude> red decFloat(-123.0, 5) .
result DecFloat: < -1,"12300",3 >

Maude> red decFloat(.123, 5) .
result DecFloat: < 1,"12300",0 >

Maude> red decFloat(.00123, 5) .
result DecFloat: < 1,"12300",-2 >

Maude> red decFloat(0.0, 5) .
result DecFloat: < 0,"00000",0 >
```

Advisory. Counterintuitive results are possible when converting from the approximate world of floating-point numbers to the exact world of rational numbers. For example,

```
Maude> red in CONVERSION : rat(1.1) .
result PosRat: 2476979795053773/2251799813685248
```

This is because, as mentioned above, 1.1 cannot be represented exactly as a floating-point number, and the nearest floating-point number is

```
1.1000000000000000888178419700125232338905334472656 25
```

which is the above rational number. (Note that Maude prints the number 1.1 as 1.1000000000000001, using 17 significant digits. The above representation is obtained by reducing `decFloat(1.1, 52)`.)

7.10 Quoted identifiers

The module `QID` is a wrapper for strings in order to provide a Maude representation for tokens of Maude syntax. Quoted identifiers are input and output by preceding a Maude identifier[4] with a (fore) quote sign. Thus `'abc` is a quoted identifier whose underlying string is `"abc"`. A quoted identifier is also an identifier, as are strings. Thus `''abc` and `'"abc"` are both quoted identifiers.

```
fmod QID is
  protecting STRING .
  sort Qid .
  op <Qids> : -> Qid [special (...)] .

  ***  qid <-> string conversions
  op string : Qid -> String [special (...)] .
  op qid : String ~> Qid [special (...)] .
endfm
```

The operators `qid` and `string` do the wrapping and unwrapping. `string` is injective, since every quoted identifier has a unique corresponding string.

```
Maude> red in QID : string('abc) .
result String: "abc"

Maude> red qid("abc") .
result Qid: 'abc

Maude> red string('a\b) .
```

```
result String: "a\\b"

Maude> red qid("a\\b") .
result Qid: 'a\b

Maude> red string('a`[b) .
result String: "a`[b"

Maude> red qid("a[b") .
result Qid: 'a`[b
```

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

```
Maude> red in QID : qid("a b c") .
result Qid: 'a`b`c

Maude> red string('a`b`c) .
result String: "a`b`c"

Maude> red qid("a\t b") .
result Qid: 'a`b

Maude> red string('a`b) .
result String: "a`b"
```

An example of a string that cannot be converted to a quoted identifier is `"a\"b"` since identifiers are not allowed to have unpaired double quotes. Thus `qid("a\"b")` has kind `[Qid]` but does not reduce to something of sort `Qid`.

7.11 Basic theories and standard views

The library of predefined modules provided by Maude in the `prelude.maude` file includes some well-known parameterized data types that will be described in the following sections. Here we will introduce the standard theories that provide the requirements for those parameterized modules.

7.11.1 `TRIV`

As already described in Section 6.3.1, the simplest non-empty theory is called `TRIV` and consists of a single sort. A model of this theory is just a set of any cardinality (finite or infinite). The intuition behind this simple theory is that the minimum requirement possible on a parameterized data type construction is having a data type as a set of basic elements to build more data on top of it. For example, in the `LIST{X :: TRIV}` parameterized data type construction we need a data type (set) of basic elements satisfying `TRIV` to then build lists of such elements.

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

The file `prelude.maude` includes many views out of `TRIV` that select the main sort of the built-in modules that we have already described in the previous sections. All these views are named in the same way: by the sort they select; for example, the standard view from `TRIV` into `RAT` selecting the sort `Rat` is also named `Rat`.

```
view Bool from TRIV to BOOL is
  sort Elt to Bool .
endv

view Nat from TRIV to NAT is
  sort Elt to Nat .
endv

view Int from TRIV to INT is
  sort Elt to Int .
endv

view Rat from TRIV to RAT is
  sort Elt to Rat .
endv

view Float from TRIV to FLOAT is
  sort Elt to Float .
endv

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

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

7.11.2 DEFAULT

The theory `DEFAULT` is slightly more complex than `TRIV`, in that in addition to a sort it also requires that there be a distinguished "default" element in such a sort. Notice that `DEFAULT` imports `TRIV` in the following presentation:

```
fth DEFAULT is
  including TRIV .
  op 0 : -> Elt .
endfth
```

The inclusion of the theory `TRIV` into the theory `DEFAULT` is made explicit by the following view, whose name coincides with the name of the target theory.

```
view DEFAULT from TRIV to DEFAULT is
endv
```

The Maude library also includes several views that map from `DEFAULT` to the various built-in data type modules by selecting the main sort and a distinguished element in it. In the case of the number sorts, this element is the zero, while for strings it is the empty string and for quoted identifiers is just the quote. Notice that operator mappings that are the identity (i.e., of the form `op 0 to 0`) do not appear explicitly in the following views but are left implicit. These views are named by appending "0" to the name of the selected sort; for example, the standard view from `DEFAULT` into `RAT` selecting the sort `Rat` and `0` as the default element is named `Rat0`.

```
view Nat0 from DEFAULT to NAT is
  sort Elt to Nat .
endv
```

```
view Int0 from DEFAULT to INT is
  sort Elt to Int .
endv

view Rat0 from DEFAULT to RAT is
  sort Elt to Rat .
endv

view Float0 from DEFAULT to FLOAT is
  sort Elt to Float .
  op 0 to term 0.0 .
endv

view String0 from DEFAULT to STRING is
  sort Elt to String .
  op 0 to term "" .
endv

view Qid0 from DEFAULT to QID is
  sort Elt to Qid .
  op 0 to term ' .
endv
```

7.11.3 `STRICT-WEAK-ORDER` and `STRICT-TOTAL-ORDER`

Although in Section 6.3.6 we have defined the notion of sorted list as based on a totally ordered set of elements, we will see in Section 7.12.6 how to relax this requirement in two different ways. The first possibility is to consider a *partially* strictly ordered set where the incomparability relation is transitive, that is, if *a* is not comparable with *b* and *b* is not comparable with *c* with respect to the given order, then *a* and *c* are not comparable either. The predefined `STRICT-WEAK-ORDER` theory below specifies a strict partial order with this additional requirement, a concept known as *strict weak order*. The second possibility is to consider a *total preorder*, as specified in Section 7.11.4 below.

Given a strict partial order <, that is, an irreflexive and transitive binary relation, we define the *incomparability* relation by $x \sim y$ iff both $x \not< y$ and $y \not< x$. Incomparability is symmetric by definition, and its reflexivity follows from the irreflexivity of <. Therefore, when we impose the additional requirement of transitivity of incomparability, we get that the relation ~ for a strict weak order is an equivalence relation.

Notice that `STRICT-WEAK-ORDER`, as presented below, imports the theory `TRIV` and also (in `protecting` mode) the module `BOOL`. The three equations express the required properties (antisymmetry is derivable from irreflexivity and transitivity) of the binary relation `_<_` on the sort `Elt`, as is made explicit in the corresponding labels.

```
fth STRICT-WEAK-ORDER is
  protecting BOOL .
  including TRIV .
  op _<_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  ceq X < Z = true if X < Y /\ Y < Z [nonexec label transitive] .
  eq X < X = false [nonexec label irreflexive] .
  ceq X < Y or Y < X or Y < Z or Z < Y = true if X < Z or Z < X
    [nonexec label incomparability-transitive] .
endfth
```

The following theory extends the previous one with a totality requirement, thus specifying a strict

total order. Under these conditions, the incomparability relation reduces to the identity (because any pair of different elements is comparable) and the transitivity of incomparability holds trivially.

```
fth STRICT-TOTAL-ORDER is
  including STRICT-WEAK-ORDER .
  vars X Y : Elt .
  ceq X = Y if X < Y = false /\ Y < X = false [nonexec label total] .
endfth
```

The theory `STRICT-TOTAL-ORDER` is a different presentation of the equivalent theory `STOSET` for strict total orders introduced in Section 6.3.1.

There is a view from `TRIV` to `STRICT-WEAK-ORDER` that forgets the order and its properties. The name of this view coincides with the name of the target theory.

```
view STRICT-WEAK-ORDER from TRIV to STRICT-WEAK-ORDER is
endv
```

The inclusion from the theory `STRICT-WEAK-ORDER` into `STRICT-TOTAL-ORDER` gives rise to another view, which is also called as the target theory.

```
view STRICT-TOTAL-ORDER from STRICT-WEAK-ORDER
  to STRICT-TOTAL-ORDER is
endv
```

The Maude library includes views that map from `STRICT-TOTAL-ORDER` to built-in data type modules by selecting the main sort and the standard strict total order between the corresponding elements, namely, the “less than” comparison between numbers and the lexicographic ordering between strings, as described in previous sections. Again, operator mappings that are the identity (in this case of the form `op _<_ to _<_`) do not appear explicitly in the following views, but are left implicit. These views are named by appending “<” to the name of the selected sort; for example, the standard view from `STRICT-TOTAL-ORDER` into `RAT` is named `Rat<`.

```
view Nat< from STRICT-TOTAL-ORDER to NAT is
  sort Elt to Nat .
endv

view Int< from STRICT-TOTAL-ORDER to INT is
  sort Elt to Int .
endv

view Rat< from STRICT-TOTAL-ORDER to RAT is
  sort Elt to Rat .
endv

view Float< from STRICT-TOTAL-ORDER to FLOAT is
  sort Elt to Float .
endv

view String< from STRICT-TOTAL-ORDER to STRING is
  sort Elt to String .
endv
```

As explained in Section 6.3.2, these views impose some proof obligations corresponding in this case to the properties that are stated about the binary relation selected in the target module; recall that such proof obligations are not discharged or checked by the system.

7.11.4 TOTAL-PREORDER and TOTAL-ORDER

The predefined TOTAL-PREORDER theory specifies, as its name clearly suggests, a *total preorder*, that is, a total binary relation which is reflexive and transitive. This theory will also be used as requirement for sorting lists in Section 7.12.6.

The notions of strict weak order (see Section 7.11.3) and of total preorder are complementary: the set-theoretic complement of a strict weak order is a total preorder and vice versa. They can also be related in a way that preserves the direction of the order. Given a strict weak order $<$, a total preorder $\leq$ is obtained by defining $x \leq y$ whenever $y \not< x$. In the other direction, a strict weak order $<$ is obtained from a total preorder $\leq$ by defining $x < y$ whenever $y \not\leq x$.

Given a total preorder $\leq$, we say that two elements x and y are *equivalent* iff both $x \leq y$ and $y \leq x$. Then, it follows from the properties of a total preorder that this is an equivalence relation and, furthermore, two elements are equivalent in a total preorder if and only if they are incomparable in the associated strict weak order (we have seen in Section 7.11.3 that the incomparability relation $\sim$ associated to a strict weak order is an equivalence relation).

Both kinds of relations capture the notion that the set of elements is split into partitions which are linearly ordered. This situation naturally arises when records are compared on a given field.

The theory TOTAL-PREORDER, as presented below, imports the theory TRIV and the module BOOL. The three equations express the required properties of the binary relation _<=_ on the sort Elt.

```
fth TOTAL-PREORDER is
  protecting BOOL .
  including TRIV .
  op _<=_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  eq X <= X = true [nonexec label reflexive] .
  ceq X <= Z = true if X <= Y /\ Y <= Z [nonexec label transitive] .
  eq X <= Y or Y <= X = true [nonexec label total] .
endfth
```

A total *order* is a total preorder that, in addition, is antisymmetric.

```
fth TOTAL-ORDER is
  inc TOTAL-PREORDER .
  vars X Y : Elt .
  ceq X = Y if X <= Y /\ Y <= X  [nonexec label antisymmetric] .
endfth
```

The theory TOTAL-ORDER is a different presentation of the equivalent theory NSTOSET for *non-strict* total orders introduced in Section 6.3.1. Its name follows the usual convention according to which, when nothing is said, a total order is assumed to be reflexive, that is, non-strict.

There is a view from TRIV to TOTAL-PREORDER, named like the target theory, that forgets the binary relation and its preorder properties.

```
view TOTAL-PREORDER from TRIV to TOTAL-PREORDER is
endv
```

The following view represents the inclusion from the TOTAL-PREORDER theory into TOTAL-ORDER.

```
view TOTAL-ORDER from TOTAL-PREORDER to TOTAL-ORDER is
endv
```

In the Maude prelude we can also find views that map from `TOTAL-ORDER` to several built-in data type modules by selecting the main sort and the standard non-strict total order between the corresponding elements, namely, the “less than or equal to” comparison between numbers and the lexicographic ordering between strings. These views are named by appending “<=” to the name of the selected sort; for example, the standard view from `TOTAL-ORDER` into `FLOAT` is named `Float<`.

```
view Nat<= from TOTAL-ORDER to NAT is
  sort Elt to Nat .
endv

view Int<= from TOTAL-ORDER to INT is
  sort Elt to Int .
endv

view Rat<= from TOTAL-ORDER to RAT is
  sort Elt to Rat .
endv

view Float<= from TOTAL-ORDER to FLOAT is
  sort Elt to Float .
endv

view String<= from TOTAL-ORDER to STRING is
  sort Elt to String .
endv
```

Again, these views impose some proof obligations that are not discharged or checked by the system.

7.12 Containers: lists and sets

The current Maude prelude includes two parameterized containers: *lists* and *sets*.

Figure 7.2 shows the relationships between the modules described in this section specifying parameterized lists and sets, including the theory `TRIV`. The module specifying sortable lists is not included in this figure, because its relationship is more complex than `protecting` importations (see later Figure 7.4).

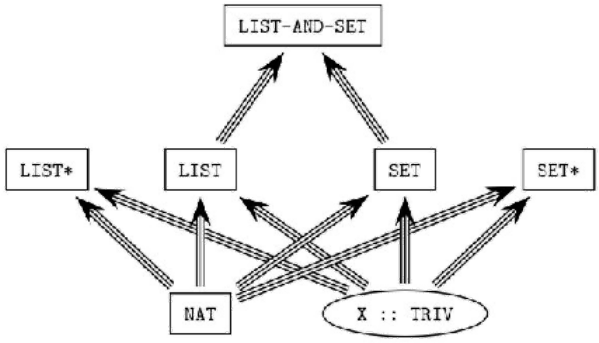

Figure 7.2: Importation graph of parameterized list and set modules

Other container data types may be added to the Maude prelude in the future.

7.12.1 Lists

Lists over a given sort of elements (provided by the theory `TRIV`) are constructed from the constant `nil` (representing the empty list) and singleton lists (identified with the corresponding elements by means of a subsort declaration) by means of an *associative* concatenation operator written as juxtaposition with empty syntax `__`.

Since there are several operations that are not well defined over the empty list, it is most useful to define the subsort of non-empty lists.

```
fmod LIST{X :: TRIV} is
  protecting NAT .
  sorts NeList{X} List{X} .
  subsort X$Elt < NeList{X} < List{X} .

  op nil : -> List{X} [ctor] .
  op __ : List{X} List{X} -> List{X} [ctor assoc id: nil prec 25] .
  op __ : NeList{X} List{X} -> NeList{X} [ctor ditto] .
  op __ : List{X} NeList{X} -> NeList{X} [ctor ditto] .

  vars E E' : X$Elt .
  vars A L : List{X} .
  var  C : Nat .
```

The operator `append` is just another name for concatenation.

```
  op append : List{X} List{X} -> List{X} .
  op append : NeList{X} List{X} -> NeList{X} .
  op append : List{X} NeList{X} -> NeList{X} .
  eq append(A, L) = A L .
```

The operations `head` and `tail` take and discard, respectively, the first (leftmost) element in a list. Analogously, the operations `last` and `front` take and discard, respectively, the last (rightmost) element in a list. It is enough to have one equation for each operation, because the case of a singleton list is obtained by matching modulo identity with `L = nil`.

```
  op head : NeList{X} -> X$Elt .
  eq head(E L) = E .

  op tail : NeList{X} -> List{X} .
  eq tail(E L) = L .

  op last : NeList{X} -> X$Elt .
  eq last(L E) = E .

  op front : NeList{X} -> List{X} .
  eq front(L E) = L .
```

The predicate `occurs` checks whether an element appears in any position in a list. The two equations in its specification correspond to the typical case analysis (or structural induction) over lists: either the list is empty or we consider the corresponding first element (in the latter case, again one equation is enough).

```
  op occurs : X$Elt List{X} -> Bool .
```

```
    eq occurs(E, nil) = false .
    eq occurs(E, E' L) = if E == E' then true else occurs(E, L) fi .
```

Reversing a list is accomplished by means of the operator `reverse`, which is efficiently defined through an auxiliary operator `$reverse` that has an additional *accumulator* argument. With this argument, `$reverse` has a simple *tail-recursive* and thus efficient definition.

```
    op reverse : List{X} -> List{X} .
    op reverse : NeList{X} -> NeList{X} .
    eq reverse(L) = $reverse(L, nil) .

    op $reverse : List{X} List{X} -> List{X} .
    eq $reverse(nil, A) = A .
    eq $reverse(E L, A) = $reverse(L, E A).
```

The tail-recursive method of definition just described will be used in the specification of several other operators, including the `size` operator on lists, which computes the number of elements in a list.

```
    op size : List{X} -> Nat .
    op size : NeList{X} -> NzNat .
    eq size(L) = $size(L, 0) .

    op $size : List{X} Nat -> Nat .
    eq $size(nil, C) = C .
    eq $size(E L, C) = $size(L, C + 1) .
  endfm
```

In the Maude prelude there are two list instantiations on built-in data types (natural numbers and quoted identifiers) that are needed by the metalevel (see Chapter 11).

```
  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
```

7.12.2 Sets

Sets over a given sort of elements (provided by the theory `TRIV`) are built from the constant `empty` and singleton sets (identified with the corresponding elements by means of a subsort declaration) with an *associative*, *commutative*, and *idempotent* union operator written `_,_`. The first two such properties are declared as attributes, while the third is written as an equation; remember that the attributes `idem` and `assoc` cannot be used together (see Section 4.4.1).

```
fmod SET{X :: TRIV} is
  protecting EXT-BOOL .
  protecting NAT .
  sorts NeSet{X} Set{X} .
  subsort X$Elt < NeSet{X} < Set{X} .

  op empty : -> Set{X} [ctor] .
  op _,_ : Set{X} Set{X} -> Set{X}
      [ctor assoc comm id: empty prec 121 format (d r os d)] .
  op _,_ : NeSet{X} Set{X} -> NeSet{X} [ctor ditto] .

  var  E : X$Elt .
  var  N : NeSet{X} .
  vars A S S' : Set{X} .
  var  C : Nat .

  eq N, N = N .
```

The prefix operator `union` is just another name for the infix operator `_,_`. Moreover, given the identification between elements and singleton sets, inserting an element is a particular case of union.

```
  op union : Set{X} Set{X} -> Set{X} .
  op union : NeSet{X} Set{X} -> NeSet{X} .
  op union : Set{X} NeSet{X} -> NeSet{X} .
  eq union(S, S') = S, S' .

  op insert : X$Elt Set{X} -> Set{X} .
  eq insert(E, S) = E, S .
```

The definitions of the operators `delete`, that deletes an element from a set, and `_in_`, that checks if an element belongs to a set, are based on the statement attribute `otherwise` (see Section 4.5.4):

1. When a given term representing a set matches the pattern `(E, S)` (modulo the equational attributes of the `_,_` operator), then we can delete the element `E` (and continue deleting, since there may be repetitions of such element in the given term), and state that indeed the element `E` belongs to the set.
2. *Otherwise*, the element `E` does not belong to the set and deleting such element does not change the set.

```
  op delete : X$Elt Set{X} -> Set{X} .
  eq delete(E, (E, S)) = delete(E, S) .
  eq delete(E, S) = S [owise] .

  op _in_ : X$Elt Set{X} -> Bool .
  eq E in (E, S) = true .
  eq E in S = false [owise] .
```

The operator `|_|` computes the cardinality of a set. Its definition goes through an auxiliary operator `$card` with an additional accumulator argument that allows a tail-recursive definition. In turn, the specification of `$card` is based on an equation that eliminates repetitions of elements in a term

representing a set; when such equation can no longer be applied (hence the `owise` attribute in the last equation), the accumulator argument does its job by counting once each different element.

```
op |_| : Set{X} -> Nat .
op |_| : NeSet{X} -> NzNat .
eq | S | = $card(S, 0) .

op $card : Set{X} Nat -> Nat .
eq $card(empty, C) = C .
eq $card((N, N, S), C) = $card((N, S), C) .
eq $card((E, S), C) = $card(S, C + 1) [owise] .
```

Both the intersection and set difference operations also use an auxiliary operation with a tail-recursive efficient definition. The accumulator argument keeps the elements that belong to both sets (for intersection) or to the first but not to the second set (for difference).

```
op intersection : Set{X} Set{X} -> Set{X} .
eq intersection(S, empty) = empty .
eq intersection(S, N) = $intersect(S, N, empty) .

op $intersect : Set{X} Set{X} Set{X} -> Set{X} .
eq $intersect(empty, S', A) = A .
eq $intersect((E, S), S', A)
  = $intersect(S, S', if E in S' then E, A else A fi) .

op _\_ : Set{X} Set{X} -> Set{X} [gather (E e)].
eq S \ empty = S .
eq S \ N = $diff(S, N, empty) .

op $diff : Set{X} Set{X} Set{X} -> Set{X} .
eq $diff(empty, S', A) = A .
eq $diff((E, S), S', A)
  = $diff(S, S', if E in S' then A else E, A fi) .
```

The following two predicates check whether their first argument is a (proper) subset of the second argument. The second one is defined in terms of the first, and in both cases the corresponding equations use the short-circuit version `_and-then_` of conjunction imported from the `EXT-BOOL` module.

```
op _subset_ : Set{X} Set{X} -> Bool .
eq empty subset S' = true .
eq (E, S) subset S' = E in S' and-then S subset S' .

op _psubset_ : Set{X} Set{X} -> Bool .
eq S psubset S' = S =/= S' and-then S subset S' .
endfm
```

The Maude metalevel (see Chapter 11) 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
```

7.12.3 Relating lists and sets

The following module provides some operations that involve both lists and sets; since these data types are not affected by the new operations, both of them are imported in `protecting` mode.

```
fmod LIST-AND-SET{X :: TRIV} is
  protecting LIST{X} .
  protecting SET{X} .

  var E : X$Elt .
  vars A L : List{X} .
  var S : Set{X} .
```

The operation `makeSet` transforms a list into a set, that is, it forgets the order between the elements and its repetitions; operationally, it simply transforms the constructors `nil` and `__` for lists into the constructors `empty` and `_,_` for sets, but this is done in an efficient way by using an auxiliary operator `$makeSet` with an accumulator argument that allows a tail-recursive definition by structural induction on the list given as first argument. Notice that both operators are overloaded to take into account in their declarations whether their arguments are empty or not.

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

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

An inverse operation `makeList` that transforms a set into a list will be seen in Section 7.12.7, because it only makes sense when we have additional information to put the elements of the set in a sequence in a univocally defined way.

The operations `filter` and `filterOut` take a list and a set as arguments, and return the list formed

by those elements of the given list that belong and that do not belong, respectively, to the given set, in their original order. Again, both are defined by means of auxiliary operations with accumulator arguments allowing efficient tail-recursive definitions.

```
  op filter : List{X} Set{X} -> List{X} .
  eq filter(L, S) = $filter(L, S, nil) .

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

  op filterOut : List{X} Set{X} -> List{X} .
  eq filterOut(L, S) = $filterOut(L, S, nil) .

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

For illustration, we consider the following instantiation and some reductions.

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

Maude> red in INT-LIST-AND-SET : filter((1 -1 1 -2 1), (1, 2)) .
result NeList{Int}: 1 1 1

Maude> red filterOut((1 -1 1 -2 1), (1, 2)) .
result NeList{Int}: -1 -2

Maude> red makeSet(1 -1 1 -2 1) .
result NeSet{Int}: 1, -1, -2
```

7.12.4 Generalized lists

With the construction of parameterized lists described in Section 7.12.1, 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 7.12.1, because we need something to distinguish the different levels of nesting of lists inside lists. We use an auxiliary sort `Item`, whose data are both elements and generalized lists (see the subsort declarations below); then we put such items next to each other by juxtaposition, getting in this way data of another auxiliary sort `PreList`, and finally we put square brackets around a “prelist” in order to get a generalized list. Notice that there is no empty “prelist” and that the empty generalized list `[]` is declared separately.

```
fmod LIST*{X :: TRIV} is
  protecting NAT .
  sorts Item{X} PreList{X} NeList{X} List{X} .
  subsort X$Elt List{X} < Item{X} < PreList{X} .
  subsort NeList{X} < List{X} .
```

```
op __ : PreList{X} PreList{X} -> PreList{X} [ctor assoc prec 25] .
op [_] : PreList{X} -> NeList{X} [ctor] .
op [] : -> List{X} [ctor] .

vars A P : PreList{X} .
var  L : List{X} .
vars E E' : Item{X} .
var  C : Nat .
```

The operator `append` now corresponds to concatenation of generalized lists and its definition is based on the juxtaposition of the "prelists" inside the generalized lists.

```
op append : List{X} List{X} -> List{X} .
op append : NeList{X} List{X} -> NeList{X} .
op append : List{X} NeList{X} -> NeList{X} .
eq append([], L) = L .
eq append(L, []) = L .
eq append([P], [A]) = [P A] .
```

The operations `head`, `tail`, `last`, and `front` work as for "standard" lists, but now they refer to the first or last *item* in the list, which can be either an element or a nested list. Now we need two equations for each operation, because the singleton case needs to be treated separately (recall that there is no empty "prelist").

```
op head : NeList{X} -> Item{X} .
eq head([E]) = E .
eq head([E P]) = E .

op tail : NeList{X} -> List{X} .
eq tail([E]) = [] .
eq tail([E P]) = [P] .

op last : NeList{X} -> Item{X} .
eq last([E]) = E .
eq last([P E]) = E .

op front : NeList{X} -> List{X} .
eq front([E]) = [] .
eq front([P E]) = [P] .
```

The predicate `occurs` checks whether an item (either an element or a list) appears in any position of the first level of a generalized list (but it does not go into deeper levels, that is, into nested lists). The three equations in its specification correspond to the typical case analysis (or structural induction) over these lists: either the list is empty, or it is a list with a single item, or it is a list with two or more items.

```
op occurs : Item{X} List{X} -> Bool .
eq occurs(E, []) = false .
eq occurs(E, [E']) = (E == E') .
eq occurs(E, [E' P])
  = if E == E' then true else occurs(E, [P]) fi .
```

The operators `reverse` and `size` for generalized lists work in a similar way to the operators with the same names in Section 7.12.1, 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
```

7.12.5 Generalized sets

The construction of generalized or nestable sets follows exactly the same pattern as the one we have seen for generalized lists in the previous section, but now we use braces instead of square brackets to make explicit the level of nesting. In particular, there is no empty "preset." Note that braces `{_}` and comma `_,_` exactly reflect standard set theory notation.

Notice that the sort named `Element` plays here the same role as `Item` played for nestable lists; do not confuse this sort with the sort `Elt` coming from the parameter theory `TRIV` in the form `X$Elt`.

The module `SET*` provides for generalized sets the same operations we have seen in Section 7.12.2 for "standard" sets, and, in addition, it specifies a powerset operator that was not possible in the previous setting.

```
  fmod SET*{X :: TRIV} is
    protecting EXT-BOOL .
    protecting NAT .
    sorts Element{X} PreSet{X} NeSet{X} Set{X} .
    subsort X$Elt Set{X} < Element{X} < PreSet{X} .
```

```
subsort NeSet{X} < Set{X} .

op _,_ : PreSet{X} PreSet{X} -> PreSet{X}
     [ctor assoc comm prec 121 format (d r os d)] .
op {_} : PreSet{X} -> NeSet{X} [ctor] .
op {} : -> Set{X} [ctor] .

vars P Q : PreSet{X} .
vars A S : Set{X} .
var  E : Element{X} .
var  N : NeSet{X} .
var  C : Nat .

eq {P, P} = {P} .
eq {P, P, Q} = {P, Q} .
```

The operations for insertion, deletion, and membership testing now work for items that can be either basic elements or nested sets, but always at the first level of nesting. For example, the membership predicate `_in_` cannot be used to test if a basic element belongs to a set inside another set, but on the other hand can check if a set is a member of another set. In other words, the operation `_in_` exactly corresponds to the set theory membership predicate ∈. As in Section 7.12.2, the operators `delete` and `_in_` are defined by means of the `otherwise` attribute. Moreover, each one has an additional equation for the singleton case, which is treated separately because there is no empty "preset."

```
op insert : Element{X} Set{X} -> Set{X} .
eq insert(E, {}) = {E} .
eq insert(E, {P}) = {E, P} .

op delete : Element{X} Set{X} -> Set{X} .
eq delete(E, {E}) = {} .
eq delete(E, {E, P}) = delete(E, {P}) .
eq delete(E, S) = S [owise] .

op _in_ : Element{X} Set{X} -> Bool .
eq E in {E} = true .
eq E in {E, P} = true .
eq E in S = false [owise] .
```

The cardinality operator `|_|` computes the number of items (either basic elements or other sets, at the first level of nesting) in a given set. It is defined with the help of an auxiliary tail-recursive operator `$card` on "presets."

```
op |_| : Set{X} -> Nat .
op |_| : NeSet{X} -> NzNat .
eq | {} | = 0 .
eq | {P} | = $card(P, 0) .

op $card : PreSet{X} Nat -> Nat .
eq $card(E, C) = C + 1 .
eq $card((N, N, P), C) = $card((N, P), C) .
eq $card((E, P), C) = $card(P, C + 1) [owise] .
```

The union operator `union` on generalized sets is based on the "union" operator `_,_` on the "presets" inside the generalized sets.

```
op union : Set{X} Set{X} -> Set{X} .
op union : NeSet{X} Set{X} -> NeSet{X} .
op union : Set{X} NeSet{X} -> NeSet{X} .
```

```
    eq union({}, S) = S .
    eq union(S, {}) = S .
    eq union({P}, {Q}) = {P, Q} .
```

The intersection and set difference operations for generalized sets have a specification very similar to the one seen in Section 7.12.2, including the use of tail-recursive auxiliary operations on "presets".

```
    op intersection : Set{X} Set{X} -> Set{X} .
    eq intersection({}, S) = {} .
    eq intersection(S, {}) = {} .
    eq intersection({P}, N) = $intersect(P, N, {}) .

    op $intersect : PreSet{X} Set{X} Set{X} -> Set{X} .
    eq $intersect(E, S, A) = if E in S then insert(E, A) else A fi .
    eq $intersect((E, P), S, A)
      = $intersect(P, S, $intersect(E, S, A)) .

    op _\_ : Set{X} Set{X} -> Set{X} [gather (E e)] .
    eq {} \ S = {} .
    eq S \ {} = S .
    eq {P} \ N = $diff(P, N, {}) .

    op $diff : PreSet{X} Set{X} Set{X} -> Set{X} .
    eq $diff(E, S, A) = if E in S then A else insert(E, A) fi .
    eq $diff((E, P), S, A) = $diff(P, S, $diff(E, S, A)) .
```

The powerset `2^ X` of a set `X` is computed by case analysis on the set `X`: it is either the empty set `{}` or a singleton set `{E}`, or it has two or more items `{E, P}`. In the last case we compute the total powerset `2^ X` by computing first the powerset `2^{P}` of the set without item `E` and then the union of this powerset `2^{P}` with the result of inserting the distinguished item `E` into all the items in the same powerset `2^{P}`. The last process is done by means of an auxiliary operation `$augment`.

```
    op 2^_ : Set{X} -> Set{X} .
    eq 2^{} = {{}} .
    eq 2^{E} = {{}, {E}} .
    eq 2^{E, P} = union(2^{P}, $augment(2^{P}, E, {})) .

    op $augment : NeSet{X} Element{X} Set{X} -> Set{X} .
    eq $augment({S}, E, A) = insert(insert(E, S), A) .
    eq $augment({S, P}, E, A)
      = $augment({P}, E, $augment({S}, E, A)) .
```

The specification of the subset predicates that check whether a set is included in another is completely analogous to the specification of the corresponding operations in Section 7.12.2.

```
    op _subset_ : Set{X} Set{X} -> Bool .
    eq {} subset S = true .
    eq {E} subset S = E in S .
    eq {E, P} subset S = E in S and-then {P} subset S .

    op _psubset_ : Set{X} Set{X} -> Bool .
    eq A psubset S = A =/= S and-then A subset S .
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}}
```

7.12.6 Sortable lists

In Section 6.3.6 we defined the notion of sorted list requiring a totally ordered set of elements, but this requirement can be relaxed. In principle, it is enough to have a transitive and antisymmetric order $<$ on a set E of elements (which are the requirements in the theory `TAOSET` from Section 6.3.1) to be able to define a *sorted list* L over E as a list such that for every pair (u,v) of members in L with u occurring before v and with $u \neq v$, it is the case that $v < u$ is false. However, in what follows we are not interested in defining sorted lists, but in specifying a sorting algorithm (more specifically, the *mergesort* algorithm) in a deterministic way. We require the sorting algorithm to be *stable*, so that incomparable elements remain in the same relative order as in the list provided as argument. For this notion to be well defined, we need to require either a strict weak order or a total preorder.

Sorting lists with respect to a strict weak order

Assume first that $<$ is a *strict weak order* over a set E, that is, a strict partial order with a transitive incomparability relation, which are precisely the requirements in the predefined theory `STRICT-WEAK-ORDER` of Section 7.11.3.

In order to define a *stable sorting* of a list L of elements over E, we consider each element of the list L as a pair (x,i), where x is the value of the element in E and i is the number indicating the position of x in L. We define an ordering $\ll$ on such pairs as follows: $(x,i) \ll (y,j)$ iff either $x < y$ or $(x \sim y$ and $i < j)$. Then, it follows from the properties of $<$ and $\sim$ that $\ll$ is a strict total order, i.e., it is irreflexive, transitive, and total.

We can now define the stable sorting under $<$ of a list $e_1,e_2,\ldots,e_n$ of elements from E as follows: Take the list $(e_1,1),(e_2,2),\ldots,(e_n,n)$, find its unique ordering $(e_{s_1},s_1),(e_{s_2},s_2),\ldots,(e_{s_n},s_n)$ under $\ll$, and output $e_{s_1},e_{s_2},\ldots,e_{s_n}$.

The parameterized module `WEAKLY-SORTABLE-LIST`, that specifies a stable version of mergesort on lists, imports "standard" lists (from Section 7.12.1), but first it is necessary to match the parameter theory `TRIV` of lists with the parameter theory `STRICT-WEAK-ORDER`. This is accomplished by means of the predefined view `STRICT-WEAK-ORDER` from `TRIV` to `STRICT-WEAK-ORDER` that forgets the order and its

properties (see Section 7.11.3). A renaming is also applied to this instantiation in order to have more convenient sort names. This process is illustrated in the diagram of Figure 7.3, where `STRICT-WEAK-ORDER` has been abbreviated to `S-W-O`, the sort renaming has been abbreviated to α, and where the different types of arrows represent the different relationships between modules: importation (triple arrow), views between theories (dashed arrow named `S-W-O`), instantiation (dashed arrow), and renaming (dotted arrow named `_*(`α`)`, meaning the renaming whose second argument is α and whose first argument is still unknown).

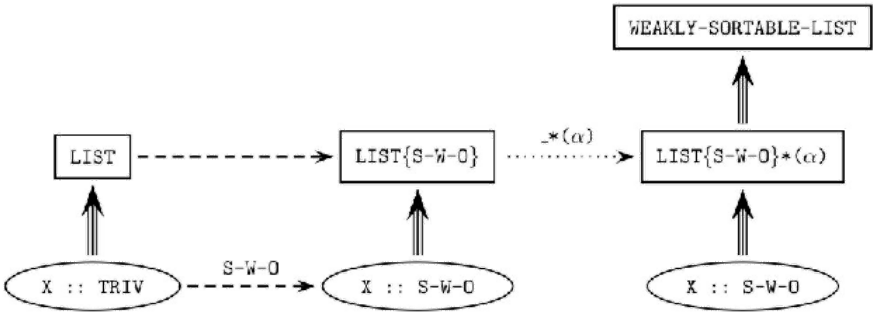

Figure 7.3: From lists to weakly sortable lists

```
fmod WEAKLY-SORTABLE-LIST{X :: STRICT-WEAK-ORDER} is
  pr LIST{STRICT-WEAK-ORDER}{X}
       * (sort NeList{STRICT-WEAK-ORDER}{X} to NeList{X},
          sort List{STRICT-WEAK-ORDER}{X} to List{X}) .
  sort $Split{X} .

  vars E E' : X$Elt .
  vars A A' L L' : List{X} .
  var  N : NeList{X} .
```

The main operation in this module is `sort`, that sorts a given list.[5] It is defined by case analysis on the list: if it is either the empty list or a singleton list, then it is already sorted; otherwise, we split the given list into two sublists, recursively sort both of them, and then merge the sorted results in order to obtain the final sorted list. This process is accomplished by means of three auxiliary operations, whose names are self-explanatory: `$split` (for the splitting, with an auxiliary result sort `$Split`), `$sort` (for the recursive sorting calls), and `$merge` (for the final merging).

```
  op sort : List{X} -> List{X} .
  op sort : NeList{X} -> NeList{X} .
  eq sort(nil) = nil .
  eq sort(E) = E .
  eq sort(E N) = $sort($split(E N, nil, nil)) .

  op $sort : $Split{X} -> List{X} .
  eq $sort($split(nil, L, L')) = $merge(sort(L), sort(L'), nil) .
```

The auxiliary operation `$split` has three arguments: the first one is the list to be split and the other two are accumulators (initially both empty) that keep the elements as they are moved from the main list into the appropriate sublists. In this way, we have an efficient tail-recursive definition.

```
  op $split : List{X} List{X} List{X} -> $Split{X} [ctor] .
  eq $split(E, A, A') = $split(nil, A E, A') .
```

```
    eq $split(E L E', A, A') = $split(L, A E, E' A') .
```

The auxiliary operation `$merge` also has three arguments, but now the first two are the lists to be merged and the third one is the accumulator where the result is incrementally computed by means of another efficient tail-recursive definition.

The module also provides an operation `merge` that simply calls the previous operation with the empty accumulator. Notice that if both lists are sorted then the result of calling `merge` on them is a sorted list, but in general `merge` is a total function that can be called on any two lists whatsoever.

```
    op merge : List{X} List{X} -> List{X} .
    op merge : NeList{X} List{X} -> NeList{X} .
    op merge : List{X} NeList{X} -> NeList{X} .
    eq merge(L, L') = $merge(L, L', nil) .

    op $merge : List{X} List{X} List{X} -> List{X} .
    eq $merge(L, nil, A) = A L .
    eq $merge(nil, L, A) = A L .
    eq $merge(E L, E' L', A)
      = if E' < E
        then $merge(E L, L', A E')
        else $merge(L, E' L', A E)
        fi .
endfm
```

The Maude prelude also provides another predefined module for sorting lists, namely, `SORTABLE-LIST`, where the required order is strict and total, as specified in the predefined theory `STRICT-TOTAL-ORDER` of Section 7.11.3. Since the theory `STRICT-TOTAL-ORDER` is a strengthening of `STRICT-WEAK-ORDER` with the additional requirement of totality, we can use it as a parameter theory to specialize our `WEAKLY-SORTABLE-LIST` module to strict total orders, thus getting the `SORTABLE-LIST` module. For this we need a view from the theory `STRICT-WEAK-ORDER` into the theory `STRICT-TOTAL-ORDER`, which is precisely the predefined inclusion view `STRICT-TOTAL-ORDER` in Section 7.11.3.

Moreover, since we also use another renaming to have more convenient sort names, the construction of the parameterized module `SORTABLE-LIST` on top of `WEAKLY-SORTABLE-LIST` mirrors the process of constructing `WEAKLY-SORTABLE-LIST` on top of `LIST`, as described in Figure 7.4, where the sort renaming has been abbreviated to α', `WEAKLY-SORTABLE-LIST` to `W-S-LIST`, `STRICT-WEAK-ORDER` to `S-W-O`, and `STRICT-TOTAL-ORDER` to `S-T-O`. The reader should compare this figure with Figure 7.3 to appreciate the similarity between both.

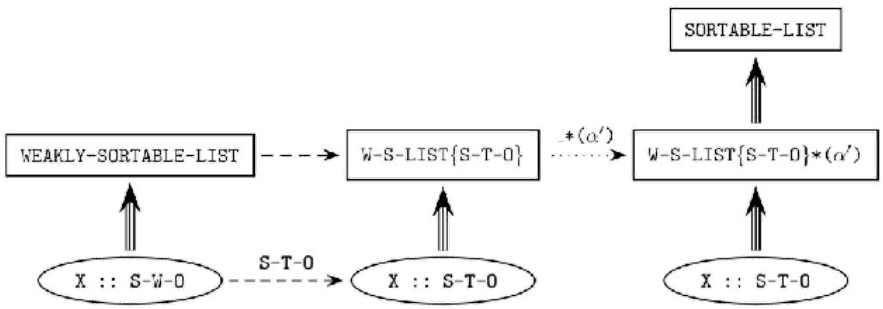

Figure 7.4: From weakly sortable lists to sortable lists

```
fmod SORTABLE-LIST{X :: STRICT-TOTAL-ORDER} is
  pr WEAKLY-SORTABLE-LIST{STRICT-TOTAL-ORDER}{X}
       * (sort NeList{STRICT-TOTAL-ORDER}{X} to NeList{X},
          sort List{STRICT-TOTAL-ORDER}{X} to List{X}) .
endfm
```

We can use the predefined view `String<` from `STRICT-TOTAL-ORDER` to `String` (where `<` is the lexicographic order on strings) to instantiate the previous module before doing some sample reductions.

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

Maude> red in STRING-SORTABLE-LIST :
         $split("a" "quick" "brown" "fox" "jumps"
                "over" "the" "lazy" "dog", nil, nil) .
result $Split{STRICT-TOTAL-ORDER}{String<}:
  $split(nil,
         "a" "quick" "brown" "fox" "jumps",
         "over" "the" "lazy" "dog")

Maude> red merge("a" "quick" "brown" "fox" "jumps",
                 "over" "the" "lazy" "dog") .
result NeList{String<}:
  "a" "over" "quick" "brown" "fox" "jumps" "the" "lazy" "dog"

Maude> red sort("a" "quick" "brown" "fox" "jumps"
                "over" "the" "lazy" "dog") .
result NeList{String<}:
  "a" "brown" "dog" "fox" "jumps" "lazy" "over" "quick" "the"

Maude> red sort("a" "quick" "brown" "fox" "jumps" "over" "the"
                "lazy" "dog" "a" "quick" "brown" "fox" "jumps"
                "over" "the" "lazy" "dog") .
result NeList{String<}: "a" "a" "brown" "brown" "dog" "dog" "fox"
  "fox" "jumps" "jumps" "lazy" "lazy" "over" "over" "quick" "quick"
  "the" "the"
```

Sorting lists with respect to a total preorder

Assume now that ≤ is a *total preorder* over a set E, that is, a binary relation satisfying the requirements in the predefined theory `TOTAL-PREORDER` of Section 7.11.4.

To define a stable sorting of a list L of elements over E, we consider again each element of the list L as a pair (x,i), where x is the value of the element in E and i is the number indicating the position of x in L. We define an ordering $\ll$ on such pairs as follows, where now the definition of $\ll$ is slightly different given the non-strict nature of total preorders: $(x,i) \ll (y,j)$ iff either $y \not\leq x$ or ($x \leq y$ and $i \leq j$). Then, the properties of ≤ imply that $\ll$ is a (non-strict) total order, i.e., it is reflexive, antisymmetric, transitive, and total. From this, the definition of a stable sorting under ≤ of a list $e_1,e_2,\ldots,e_n$ of elements from E follows exactly the same steps as before: Take the list $(e_1,1),(e_2,2),\ldots,(e_n,n)$, find its unique ordering $(e_{s_1},s_1),(e_{s_2},s_2),\ldots,(e_{s_n},s_n)$ under $\ll$, and output $e_{s_1},e_{s_2},\ldots,e_{s_n}$.

The following modules `WEAKLY-SORTABLE-LIST'` and `SORTABLE-LIST'` specify the mergesort algorithm with respect to a total preorder and a (non-strict) total order, respectively. Their structure is completely analogous to the structure of `WEAKLY-SORTABLE-LIST` and `SORTABLE-LIST` already explained

above. It is described in the two diagrams of Figure 7.5, where the sort renamings have been abbreviated to γ and γ', `TOTAL-PREORDER` to `T-PREORDER`, `TOTAL-ORDER` to `T-ORDER`, and `WEAKLY-SORTABLE-LIST'` `W-S-LIST'`.

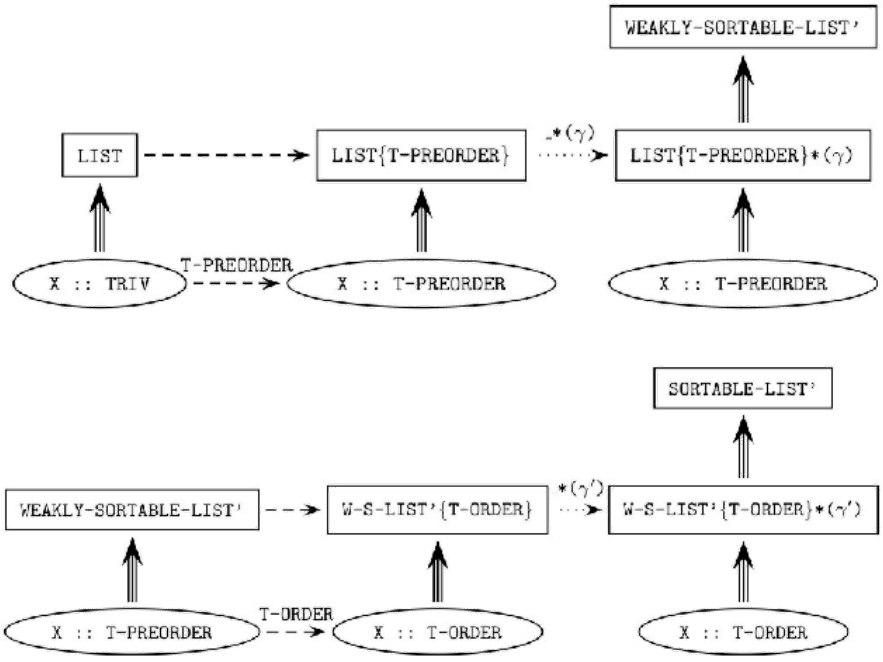

Figure 7.5: Another version of sortable lists

```
fmod WEAKLY-SORTABLE-LIST'{X :: TOTAL-PREORDER} is
  pr LIST{TOTAL-PREORDER}{X}
       * (sort NeList{TOTAL-PREORDER}{X} to NeList{X},
          sort List{TOTAL-PREORDER}{X} to List{X}) .
  sort $Split{X} .

  vars E E' : X$Elt .
  vars A A' L L' : List{X} .
  var  N : NeList{X} .

  op sort : List{X} -> List{X} .
  op sort : NeList{X} -> NeList{X} .
  eq sort(nil) = nil .
  eq sort(E) = E .
  eq sort(E N) = $sort($split(E N, nil, nil)) .

  op $sort : $Split{X} -> List{X} .
  eq $sort($split(nil, L, L')) = $merge(sort(L), sort(L'), nil) .

  op $split : List{X} List{X} List{X} -> $Split{X} [ctor] .
  eq $split(E, A, A') = $split(nil, A E, A') .
  eq $split(E L E', A, A') = $split(L, A E, E' A') .

  op merge : List{X} List{X} -> List{X} .
  op merge : NeList{X} List{X} -> NeList{X} .
  op merge : List{X} NeList{X} -> NeList{X} .
  eq merge(L, L') = $merge(L, L', nil) .

  op $merge : List{X} List{X} List{X} -> List{X} .
```

```
    eq $merge(L, nil, A) = A L .
    eq $merge(nil, L, A) = A L .
    eq $merge(E L, E' L', A)
      = if E <= E'
        then $merge(L, E' L', A E)
        else $merge(E L, L', A E')
        fi .
  endfm

  fmod SORTABLE-LIST'{X :: TOTAL-ORDER} is
    pr WEAKLY-SORTABLE-LIST'{TOTAL-ORDER}{X}
         * (sort NeList{TOTAL-ORDER}{X} to NeList{X},
            sort List{TOTAL-ORDER}{X} to List{X}) .
  endfm
```

Apart from the changes in the requirement theories and the module names, the main difference bewteen both approaches appears in the third `$merge` equation. In the `WEAKLY-SORTABLE-LIST` module we have

```
eq $merge(E L, E' L', A)
  = if E' < E
    then $merge(E L, L', A E')
    else $merge(L, E' L', A E)
    fi .
```

Here we are dealing with a strict weak order. We test `E' < E`. If it is true, then by irreflexivity we know that `E < E'` is false, and the element `E'` from the second list is appended to the merged list. Whereas if `E' < E` is false, we know that either `E < E'` holds or `E` and `E'` are incomparable. Either way, the element `E` from the first list is appended to the merged list, either because it is smaller or because it is incomparable and we are preserving the original relative positions in the list (stability).

On the other hand, in the `WEAKLY-SORTABLE-LIST'` module we have

```
eq $merge(E L, E' L', A)
  = if E <= E'
    then $merge(L, E' L', A E)
    else $merge(E L, L', A E')
    fi .
```

In this case we are dealing with a total preorder. We test `E <= E'`. If it is true, then either `E' <= E` is false or `E` and `E'` are equivalent. Either way, the element `E` from the first list is appended to the merged list, either because it is smaller or because it is equivalent and we are preserving the original relative positions in the list (stability). If `E <= E'` is false, then `E' <= E` holds by totality and therefore `E'` is appended to the merged list.

We can redo with these modules the same instantiation we considered above, but using now the predefined view `String<=` from `TOTAL-ORDER` to `String`, where `<=` is the non-strict lexicographic order on strings.

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

Maude> red in STRING-SORTABLE-LIST' :
         sort("a" "quick" "brown" "fox" "jumps"
              "over" "the" "lazy" "dog") .
result NeList{String<=}:
```

```
    "a" "brown" "dog" "fox" "jumps" "lazy" "over" "quick" "the"

Maude> red sort("a" "quick" "brown" "fox" "jumps" "over" "the"
                "lazy" "dog" "a" "quick" "brown" "fox" "jumps"
                "over" "the" "lazy" "dog") .
result NeList{String<=}: "a" "a" "brown" "brown" "dog" "dog" "fox"
  "fox" "jumps" "jumps" "lazy" "lazy" "over" "over" "quick" "quick"
  "the" "the"
```

7.12.7 Making lists out of sets

In Section 7.12.3 we have seen an operation `makeSet` that transforms a list into a set with the same elements. On the other hand, transforming a set into a list imposes some order on the given elements, which can be done in many different ways, and therefore only makes sense as a function when we have additional information over those elements that allows us to choose a unique sequence. The solution adopted here is to require either a strict or a non-strict total order on the elements, so that the resulting list is the corresponding sorted list. For this we use the `sort` operation defined either in the `SORTABLE-LIST` module or in the `SORTABLE-LIST'` module described in the previous section. In both versions the main operation `makeList` is defined in terms of an auxiliary operation `$makeList` with an accumulator in order to have a more efficient definition.

In both versions the `LIST-AND-SET` module is imported with a double renaming (different in each case), which is needed for correct sharing of a renamed copy of the `LIST` module, because Core Maude does not evaluate the composition of renamings but applies them sequentially. If we computed manually and used this simpler renaming, we would get a different renaming of `LIST` imported by each `protecting` declaration; then, while these renamings would have the same effect, we would import two renamed copies of `LIST` rather than a shared copy.

This is the first version, using a strict total order.

```
fmod SORTABLE-LIST-AND-SET{X :: STRICT-TOTAL-ORDER} is
  pr SORTABLE-LIST{X} .
  pr LIST-AND-SET{STRICT-WEAK-ORDER}{STRICT-TOTAL-ORDER}{X}
       * (sort NeList{STRICT-WEAK-ORDER}{STRICT-TOTAL-ORDER}{X}
            to NeList{STRICT-TOTAL-ORDER}{X},
          sort List{STRICT-WEAK-ORDER}{STRICT-TOTAL-ORDER}{X}
            to List{STRICT-TOTAL-ORDER}{X})
       * (sort NeList{STRICT-TOTAL-ORDER}{X} to NeList{X},
          sort List{STRICT-TOTAL-ORDER}{X} to List{X},
          sort NeSet{STRICT-WEAK-ORDER}{STRICT-TOTAL-ORDER}{X}
            to NeSet{X},
          sort Set{STRICT-WEAK-ORDER}{STRICT-TOTAL-ORDER}{X}
            to Set{X}) .

  var E : X$Elt .
  var L : List{X} .
  var S : Set{X} .

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

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

```
  eq $makeList((E, S), L) = $makeList(S, E L) [owise] .
endfm
```

Notice that `makeList` is only a partial inverse to `makeSet`, not only because of sorting the elements, but also because in a set repetitions do not matter. In general, for a set `S` and a list `L` we have `makeSet(makeList(S))` = `S`, but in general `makeList(makeSet(L))`≠`L`.

We consider an instantiation with the predefined view `Int<` and some sample reductions.

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

Notice that in the following first reduction we get a list different from the original one, while in the second reduction we get a different representation (where repetitions have been eliminated) of the same set. Those possible repetitions are already eliminated before producing the corresponding list, as shown in the third reduction.

```
Maude> red in INT-SORTABLE-LIST-AND-SET :
         makeList(makeSet(1 -1 1 -2 1 0)) .
result NeList{Int<}: -2 -1 0 1

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

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

This is the second version, using a non-strict total order.

```
fmod SORTABLE-LIST-AND-SET'{X :: TOTAL-ORDER} is
  pr SORTABLE-LIST'{X} .
  pr LIST-AND-SET{TOTAL-PREORDER}{TOTAL-ORDER}{X}
       * (sort NeList{TOTAL-PREORDER}{TOTAL-ORDER}{X}
            to NeList{TOTAL-ORDER}{X},
          sort List{TOTAL-PREORDER}{TOTAL-ORDER}{X}
            to List{TOTAL-ORDER}{X})
       * (sort NeList{TOTAL-ORDER}{X} to NeList{X},
          sort List{TOTAL-ORDER}{X} to List{X},
          sort NeSet{TOTAL-PREORDER}{TOTAL-ORDER}{X} to NeSet{X},
          sort Set{TOTAL-PREORDER}{TOTAL-ORDER}{X} to Set{X}) .

  var E : X$Elt .
  var L : List{X} .
  var S : Set{X} .

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

  op $makeList : Set{X} List{X} -> List{X} .
  op $makeList : NeSet{X} List{X} -> NeList{X} .
  op $makeList : Set{X} NeList{X} -> NeList{X} .
  eq $makeList(empty, L) = sort(L) .
  eq $makeList((E, E, S), L) = $makeList((E, S), L) .
  eq $makeList((E, S), L) = $makeList(S, E L) [owise] .
endfm
```

We redo the same instantiation, now with the non-strict total order on integers.

```
fmod INT-SORTABLE-LIST-AND-SET' is
  pr SORTABLE-LIST-AND-SET'{Int<=} .
endfm

Maude> red in INT-SORTABLE-LIST-AND-SET' :
         makeList(makeSet(1 -1 1 -2 1 0)) .
result NeList{Int<=}: -2 -1 0 1

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

Maude> red makeList((5, 4, 3, 4, 5)) .
result NeList{Int<=}: 3 4 5
```

7.13 Maps and arrays

Both *maps* and *arrays* represent a function *f* between two sets as a set of pairs of the form

$$\{(a_1, f(a_1)), (a_2, f(a_2)), \ldots, (a_n, f(a_n))\}$$

in the graph of the function; each pair (a_i,f(a_i)) is called an *entry* in both cases.

The difference between maps and arrays is that the former leave undefined the result of *f* over those values not present in the set above, while the latter assign a "default" value result in that case.

However, notice that the modules below *do not check*, for efficiency reasons, that all values a_i in the first components of a set of pairs like the previous one are *different* (although the operations for insertion and look up make sure that the corresponding result is well defined). The situation of having a set of entries with repeated first components never arises if such a map or array is initially the empty one and then it is only modified by means of the `insert` operation. See Section 15.3.2 for a more careful specification of (finite) partial functions checking these requirements.

7.13.1 Maps

As explained above, a map is defined as a "set" (built with the associative and commutative operator `_,_`) of entries. Notice that `Entry`, whose only constructor is the operator `_|->_`, is a subsort of `Map`.

The domain and codomain values of the map come from the parameters of the parameterized data type, both of them satisfying the theory `TRIV` and thus providing a set of elements.

The module `MAP` provides a constant `undefined` of the *kind* `[Y$Elt]` corresponding to the sort `Y$Elt` and representing the undefined result.

```
fmod MAP{X :: TRIV, Y :: TRIV} is
  protecting BOOL .
  sorts Entry{X,Y} Map{X,Y} .
  subsort Entry{X,Y} < Map{X,Y} .

  op _|->_ : X$Elt Y$Elt -> Entry{X,Y} [ctor] .
  op empty : -> Map{X,Y} [ctor] .
  op _,_ : Map{X,Y} Map{X,Y} -> Map{X,Y}
       [ctor assoc comm id: empty prec 121 format (d r os d)] .
```

```
    op undefined : -> [Y$Elt] [ctor] .

    var  D : X$Elt .
    vars R R' : Y$Elt .
    var  M : Map{X,Y} .
```

The operator `insert` adds a new entry to a map, but when the first argument already appears in the domain of definition of the map, the second argument is used to *update* the map. Notice the use of matching and of the `otherwise` attribute to distinguish these two cases in a simple way. Furthermore, in the first case, an auxiliary operation `$hasMapping` is used to make sure that in the resulting map only one entry is associated with the given value. The operation `$hasMapping` checks whether a domain value actually has an associated entry in a map.

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

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

The lookup operator is represented with the notation `_[_]`. Again, matching and `owise` are used to distinguish whether or not the second argument appears in the domain of definition of the map provided as first argument. When the answer is affirmative an the map contains exactly one entry associated with such argument (as checked with the auxiliary operation `$hasMapping`), the result is the value provided in that entry. When the answer is negative or the map is not well defined because there is more than one entry associated with the same argument, the result is the constant `undefined` in the kind, with the self-explanatory meaning that in those cases the map is undefined on the given argument.

```
    op _[_] : Map{X,Y} X$Elt -> [Y$Elt] [prec 23] .
    eq (M, D |-> R)[D]
      = if $hasMapping(M, D) then undefined else R fi .
    eq M[D] = undefined [owise] .
  endfm
```

We use the predefined views `String` and `Nat` (see Section 7.11.1) to define maps from strings to natural numbers, and do some sample reductions.

```
  fmod STRING-NAT-MAP is
    pr MAP{String, Nat} .
  endfm

  Maude> red in STRING-NAT-MAP :
           insert("one", 1,
             insert("two", 2, insert("three", 3, empty))) .
  result Map{String,Nat}: "one" |-> 1, "three" |-> 3, "two" |-> 2

  Maude> red insert("one", 1,
               insert("two", 2,
                 insert("three", 3, empty)))["two"] .
  result NzNat: 2

  Maude> red insert("one", 1,
```

```
               insert("two", 2,
                 insert("three", 3, empty)))["four"] .
  result [FindResult]: undefined

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

The last reduction shows that the undesired repetition of a domain value in two entries of the same map also produces the `undefined` constant as result.

7.13.2 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 7.11.2).

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

```
  fmod ARRAY{X :: TRIV, Y :: DEFAULT} is
    protecting BOOL .
    sorts Entry{X,Y} Array{X,Y} .
    subsort Entry{X,Y} < Array{X,Y} .

    op _|->_ : X$Elt Y$Elt -> Entry{X,Y} [ctor] .
    op empty : -> Array{X,Y} [ctor] .
    op _;_ : Array{X,Y} Array{X,Y} -> Array{X,Y}
         [ctor assoc comm id: empty prec 71 format (d r os d)] .

    var  D : X$Elt .
    vars R R' : Y$Elt .
    var  A : Array{X,Y} .
```

The definition of the operator `insert` for arrays adds a check to the definition of the same operator for maps so that, as mentioned above, entries whose second value is the default value `0` are never inserted. Note, however, that mappings to the default value `0` that are created with the constructors `_|->_` and `_;_`, rather than the `insert` operator, are not removed as doing this check each time a new array is formed would be excessively inefficient. Furthermore, as we have already seen for maps, in the first case, an auxiliary operation `$hasMapping` is used to make sure that in the resulting array only one entry is associated with the given value.

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

    op $hasMapping : Array{X,Y} X$Elt -> Bool .
    eq $hasMapping((A ; D |-> R), D) = true .
```

```
    eq $hasMapping(A, D) = false [owise] .
```

The definition of the lookup operator for arrays only differs from the one for maps in the occurrence of the default value `0` instead of the constant `undefined`. Now, if an argument has more than one associated entry (as checked with the auxiliar operation `$hasMapping`), it is considered to be "unmapped" and the result is also the default value.

```
  op _[_] : Array{X,Y} X$Elt -> Y$Elt [prec 23] .
  eq (A ; D |-> R)[D]
    = if $hasMapping(A, D) then 0 else R fi .
  eq A[D] = 0 [owise] .
endfm
```

We do the same instantiation for arrays as for maps, with the predefined views `String` from Section 7.11.1 and `Nat0` from Section 7.11.2).

```
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
```

7.14 A linear Diophantine equation solver

The Maude system includes a built-in linear Diophantine equation solver. The interface to the solver is defined in the file `linear.maude` which contains the functional module `DIOPHANTINE`. The current solver finds non-negative solutions of a system S of n simultaneous linear equations in m variables having the form $Mv = c$, where M is an $n \times m$ integer coefficient matrix, v is a column vector of m variables and c is a column vector of n integer constants.

Both matrices and vectors are represented as sparse arrays with their dimensions implicit and their indices starting from 0. For this we make heavy use of the parameterized module `ARRAY`, described in Section 7.13.2.

First, a data type of pairs of natural numbers to be used as indices for matrices is created.

```
fmod INDEX-PAIR is
  pr NAT .
  sort IndexPair .
```

```
    op _,_ : Nat Nat -> IndexPair [ctor] .
endfm
```

Then, we instantiate (and rename as desired) the parameterized module `ARRAY` to obtain matrices of integers. Notice that `Int0` is the view from `DEFAULT` to `INT` given in Section 7.11.2

```
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 7.11.1. 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 non-empty and *B* is empty), or infinitely many solutions (in this case *A* and *B* are both non-empty).

In either case, the solution encoding `[` *A* `|` *B* `]` is unique.

Deciding whether a linear Diophantine system admits a non-negative, nontrivial solution is NP-complete (stated as known in [104]). 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 [28] 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 [34],

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

which has a basis of size 13.

```
Maude> red in DIOPHANTINE :
         natSystemSolve(
           (0,0) |-> 1 ; (0,1) |-> 2 ; (0,2) |-> -1 ;
           (0,3) |-> 0 ; (0,4) |-> -2 ; (0,5) |-> -1 ;
           (1,0) |-> 0 ; (1,1) |-> -1 ; (1,2) |-> -2 ;
           (1,3) |-> 2 ; (1,4) |-> 0 ; (1,5) |-> 1 ;
           (2,0) |-> 2 ; (2,1) |-> 0 ; (2,2) |-> 1 ;
           (2,3) |-> -1 ; (2,4) |-> -2 ; (2,5) |-> 0,
           zeroVector,
           "cd") .
rewrites: 1 in 10ms cpu (46ms real) (100 rews/sec)
result IntVectorSetPair:
  [
    zeroVector
  |
    0 |-> 1 ; 1 |-> 1 ; 4 |-> 1 ; 5 |-> 1,
    0 |-> 1 ; 1 |-> 4 ; 2 |-> 9 ; 3 |-> 11,
    0 |-> 10 ; 1 |-> 4 ; 3 |-> 2 ; 4 |-> 9,
    1 |-> 1 ; 2 |-> 1 ; 3 |-> 1 ; 5 |-> 1,
    1 |-> 8 ; 2 |-> 2 ; 4 |-> 1 ; 5 |-> 12,
    0 |-> 2 ; 1 |-> 4 ; 2 |-> 8 ; 3 |-> 10 ; 4 |-> 1,
    0 |-> 3 ; 1 |-> 4 ; 2 |-> 7 ; 3 |-> 9 ; 4 |-> 2,
    0 |-> 4 ; 1 |-> 4 ; 2 |-> 6 ; 3 |-> 8 ; 4 |-> 3,
    0 |-> 5 ; 1 |-> 4 ; 2 |-> 5 ; 3 |-> 7 ; 4 |-> 4,
    0 |-> 6 ; 1 |-> 4 ; 2 |-> 4 ; 3 |-> 6 ; 4 |-> 5,
    0 |-> 7 ; 1 |-> 4 ; 2 |-> 3 ; 3 |-> 5 ; 4 |-> 6,
    0 |-> 8 ; 1 |-> 4 ; 2 |-> 2 ; 3 |-> 4 ; 4 |-> 7,
    0 |-> 9 ; 1 |-> 4 ; 2 |-> 1 ; 3 |-> 3 ; 4 |-> 8
  ]
```

The string `"gcd"` specifies an original algorithm based on integer Gaussian elimination followed by a sequence of extended greatest common divisor (gcd) computations. It can “home in” quickly on solutions involving large numbers but it is very sensitive to the number of degrees of freedom and can easily degenerate into a brute force search. Furthermore, termination depends on the bound on the sum of minimal solutions established in [95], 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 [28]:

```
red in DIOPHANTINE :
  natSystemSolve(
    (0,0) |-> 1 ; (0,1) |-> -5 ; (1,1) |-> 4 ; (1,2) |-> -5 ;
    (2,2) |-> 4 ; (2,3) |-> -5 ; (3,3) |-> 4 ; (3,4) |-> -5 ;
    (4,4) |-> 4 ; (4,5) |-> -5 ; (5,5) |-> 4 ; (5,6) |-> -5,
    0 |-> 1 ; 1 |-> 1 ; 2 |-> 1 ; 3 |-> 1 ; 4 |-> 1 ; 5 |-> 1,
    "gcd") .
result IntVectorSetPair:
  [
    0 |-> 15621 ; 1 |-> 3124 ; 2 |-> 2499 ; 3 |-> 1999 ;
    4 |-> 1599 ; 5 |-> 1279 ; 6 |-> 1023
  |
    0 |-> 15625 ; 1 |-> 3125 ; 2 |-> 2500 ; 3 |-> 2000 ;
    4 |-> 1600 ; 5 |-> 1280 ; 6 |-> 1024
  ]
```

Finally, the string `""` can be passed as third argument of `natSystemSolve`, thus allowing the system to choose which algorithm to use. For convenience, the operator

```
op natSystemSolve : IntMatrix IntVector -> IntVectorSetPair .
```

is equationally defined to invoke the built-in operator with `""`

```
  eq natSystemSolve(M:IntMatrix, V:IntVector)
    = natSystemSolve(M:IntMatrix, V:IntVector, "") .
endfm
```