
2.7.2 Mixfix Parsing of Terms in a Module

We can illustrate the notion of term and the idea of parsing terms in the signature of a module by means of the following `BINARY-NAT` module supporting natural number arithmetic in binary notation. The module includes the usual arithmetic operators `_+_`, `_*_`, and `__` of sum, product, and exponentiation on natural numbers in binary notation plus:

- Constants `0` and `1` as constructors of the sort `Bit`.
- The operator `__` to represent elements of the sort `Bits` as sequences of `0`'s and `1`'s.
- The operator `|_|`, to obtain the length of a binary number.
- The operator `normalize`, to compress the representation of a binary number by suppressing the 0's on the left of a number, if any.
- The "greater-than" Boolean predicate `_>_`.
- The `not_` operator, that performs the logical negation of a string of bits.

```
fmod BINARY-NAT is
  protecting MACHINE-INT .
  sorts Bit Bits .
  subsort Bit < Bits .

  ops 0 1 : -> Bit .

  op __ : Bits Bits -> Bits [assoc] .
  op |_| : Bits -> MachineInt .
  op not_ : Bits -> Bits .
  op normalize : Bits -> Bits .
  ops _+_ _*_ : Bits Bits -> Bits [assoc comm] .
  op _^_ : Bits Bits -> Bits .
  op _>_ : Bits Bits -> Bool .
  op _?_:_ : Bool Bits Bits -> Bits .

  vars S T : Bits .
  vars B C : Bit .
  var L : Bool .

  *** Length
  eq | B | = 1 .
  eq | S B | = | S | + 1 .

  *** Not
  eq not (S T) = (not S) (not T) .
  eq not 0 = 1 .
  eq not 1 = 0 .

  *** Normalize suppresses zeros at the left of a binary number
  eq normalize(0 S) = normalize(S) .
  eq normalize(1 S) = 1 S .

  *** Greater than
  eq 0 > S = false .
  eq 1 > (0).Bit = true .
  eq 1 > (1).Bit = false .
  eq B > (0 S) = B > S .
  eq B > (1 S) = false .
  eq (1 S) > B = true .
  eq (B S) > (C T)
    = if | normalize(B S) | > | normalize(C T) |
      then true
      else if | normalize(B S) | < | normalize(C T) |
           then false
           else (S > T)
           fi
      fi .

  *** Binary addition
  eq 0 + S = S .
  eq 1 + 1 = 1 0 .
  eq 1 + (T 0) = T 1 .
  eq 1 + (T 1) = (T + 1) 0 .
  eq (S B) + (T 0) = (S + T) B .
  eq (S 1) + (T 1) = (S + T + 1) 0 .

  *** Binary multiplication
  eq 0 * T = 0 .
  eq 1 * T = T .
  eq (S B) * T = ((S * T) 0) + (B * T) .

  *** Binary exponentiation
  eq T ^ 0 = 1 .
  eq T ^ 1 = T .
  eq T ^ (S B) = (T ^ S) * (T ^ S) * (T ^ B) .

  *** Mixfix ?: operator
  eq L ? S : T = if L then S else T fi .
endfm
```

Note the use of the sort `Bool`. This sort is not a proper sort of the signature of `BINARY-NAT`. However, `Bool`, together with other information about polymorphic operators, parentheses, subsort-overloaded operators, and so on, belongs to the extended signature of a module, which Maude generates automatically for each module (see Section 2.7.3).

This module illustrates several important aspects of the grammatical power of Maude.

- *Empty Syntax*: With the following declarations we can write natural numbers in binary notation such as 1 0 0 1 or 1 1.

  ```
  sorts Bit Bits .
  subsort Bit < Bits .

  ops 0 1 : -> Bit .

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

- *Outfix Syntax*: The length operator `|_|` is an example of outfix syntax specification for operators.

  ```
  Maude> red | 1 0 1 1 0 | .
  result NzMachineInt: 5
  ```

- *Prefix and Postfix Syntax*: `BINARY-NAT` includes the `not_` operator, defined with prefix syntax.

  ```
  Maude> red 0 (not 1) 0 .
  result Bits: 0 0 0

  Maude> red not (1 0 1) .
  result Bits: 0 1 0
  ```

- *Infix Syntax*: The operators `_+_`, `_*_` and `_>_` illustrate the model for the specification of infix operators in Maude.

  ```
  Maude> red (1 0 0) + (1 1 1) .
  result Bits: 1 0 1 1

  Maude> red (1 1 1) * (1 1 0) .
  result Bits: 1 0 1 0 1 0

  Maude> red (0 1 0 1) > (1 1) .
  result Bool: true
  ```

- *Mixfix Syntax*: The operator `_?_:_` is an example of mixfix notation in Maude. In fact, this operator combines both outfix and infix notation.

  ```
  Maude> red ((1 0) > (0 1 1)) ? ((1 0) * 1) : ((1 0) + 1) .
  result Bits: 1 1
  ```

The previous discussion on the user-definable notational power of Maude is a good basis for discussing the process of parsing terms in the context of a signature. This process is divided in two phases. In a first step, Maude collects all the information pertinent to the parsing problem included in a module: set of sorts, subsort relations, operators (paying special attention to the notational pattern of each operator) and variables. All this information is translated into a context-free grammar. In a second step, each time Maude detects a term in the corresponding signature, the MSCP algorithm is used to obtain the grammatical structure of the term according to the context-free grammar previously obtained from the module.
