A Library for PSF

J.J. van Wamel

University of Amsterdam, Programming Research Group

Kruislaan 403, 1098 SJ Amsterdam, The Netherlands.

E-mail: `vanw@fwi.uva.nl`

Abstract

A library for the specification language PSF is specified that contains some standard numerical data types. The following numerical data are specified: binary numbers, natural numbers, integer numbers and floating point numbers. For these data types a number of standard functions, as they are available in current programming languages, are specified. On top of the natural numbers, the data types character strings, tables and queues are specified. The specification of data types in PSF is exclusively a matter of algebraic data specification, based on the formalism ASF, so the library can rather be regarded an ASF specification than a typical PSF specification.

1991 Mathematics Subject Classification: 68Q40.

1987 CR Categories: D.1.3, D.2.1, F.1.2.

Key Words & Phrases: Specification Language, PSF, Abstract Data Types.

1 Introduction

The specification language PSF (Process Specification Formalism [MV90]) is a formal description technique based on two algebraic formalisms, which are ASF (Algebraic Specification Formalism [BHK89]) and ACP (Algebra of Communicating Processes [BW90]). These two formalisms divide PSF in two functionally different halves to which we will refer as the data part and the process part.

The specification of data types and functions on data types happens in the *data modules* of PSF, which are based on ASF. Processes are specified in the *process modules* of PSF, which are based on abstract or untimed ACP.

The data part forms the lower level of a PSF specification and the process part forms the higher level. This is because processes are usually parameterised with data and data not with processes. For examples of specifications the reader is referred to the vast amount of literature that is written on this subject.

A problem that often arises in specification in PSF is that there are no standard data types available, such as binary numbers or natural numbers. Apart from being inefficient gives this often rise to a lack of uniformity in the specifications of these data types and operators on it. For this reason, the specification of a numerical library may be useful and somewhat more substantial than just an interesting exercise.

Another point of interest is that a specification of data in PSF can be easily transformed into a specification in the language μCRL [GP90]. μCRL is also a language based on ACP but it does not provide modular structuring of specifications nor parameterisation, contrary to PSF.

The design of the numbers for the library is made under the following considerations:

1. The representation of the elements of the different data types has to be uniform, as well as the representation of the standard functions. So we will try to avoid a mixture of notations.

2. The rewriting procedures on the different functions such as addition, multiplication and so on, must be efficient. Preferably, significantly faster than those for existing specifications in PSF. This criterion is somewhat arbitrary because it depends on the current implementation of the PSF *term rewriter*.

3. We will restrict ourselves to elementary school mathematics for two reasons. The first is that PSF is simply not designed for the purpose of number crunching. The second is of a more fundamental nature; the greater the library is, the more time the compilation of it will cost.

4. We want to have a specification of floating point numbers with at least some basic operations on it such as addition, subtraction, multiplication and division.

In the specification of the library for PSF in this paper, we will motivate the different choices we will have to make following the considerations above. We will also give some examples, as generated by the PSF term rewriter, that may help to demonstrate the use of the specification. In section 10, also some attention is paid to the notion of correctness of an algebraic specification and to the testing of the specification with the term rewriter.

It may be clear that we will mainly restrict ourselves to numerical data types, but we will also specify characters, character strings and some basic functions on these types. The data types *queues* and *tables*, which turned out to be useful for the specification of communication protocols in the past, are also included. The numerical library together with the remaining modules will be referred to as the *PSF library.*

acknowledgements. The author would like to thank Gert Veltink for his help by making a first sketch of the specification and Sjouke Mauw for 'shooting some holes' in earlier versions of the specification and for giving some good suggestions. Thanks to Jacob Brunekreef for reading a draft version of this paper.

Contents

2 Some Remarks on the Specification of Numbers

In this section we will motivate some choices we will make for the specification of numbers. Choices concerning questions such as 'are natural numbers or binary numbers the most efficient type for the execution of calculations', and 'what to do with division by zero'. Moreover the modular structure of the specification will be given.

2.1 Choices

One of the first questions that arise when the numbers are to be specified is whether decimal numbers or binary numbers form the most efficient basis for calculations. Also other numerical types, such as hexadecimal numbers can be considered in this context. The first we have to remark is that the natural or decimal numbers will probably be the most used numerical type. If we choose another type for the basis of calculations, then for each calculation conversions to and from this basis type are needed. Some trial and error shows that the greater the radix of a number is, the faster the term rewriting goes. This is easy to understand because numbers with a small radix consist of relatively more symbols than numbers with a greater radix. So natural numbers in a decimal form are a better basis for efficient term rewriting than binary numbers. Nevertheless, we will specify the binary numbers independently from natural numbers. The reason for not converting binary numbers to natural numbers and back for calculations, is that this way the binaries can be used in a specification without the automatic import of the natural numbers.

Note that we implicitly discarded the option of specifying the natural numbers using the so-called *successor* naturals, which lead to very short but inefficient specifications. The specification will be based on simple tables of addition, subtraction, and so on, because this makes fast term rewriting possible. The way the functions are specified is very similar to the 'paper and pencil' methods from elementary school.

We will not only restrict ourselves as much as possible to elementary school methods, but also to the functions as taught there. If a user needs higher order functions, such as trigonomic or logarithmic functions, these functions can be specified on top of the floating point numbers, which will also be specified in this paper. Apart from the extra work there is a practical reason for not implementing higher functions; every extra function will slow down the compilation of the imported modules. Moreover, higher order functions often lead to such involved (and slow) rewrite procedures, that their practical use is quite restricted.

Now we get to the interpretation of two partial functions. Subtraction is defined as a *monus* function; this implies that

$$x - y = 0 \;\; when \;\; y \geq x$$

Division by zero will always yield a zero. This saves us from the introduction of error elements for *every* sort and from the specification of many extra rewrite rules.

Last but not least we will have to make a choice for the representation of the elements of the different numerical types. A nice uniform notation is the following: the type of every number can be identified when a constructor is used that symbolises the sort name. These constructors on their part can be parameterised with one or more strings of symbols, which represent the value of the number. For integer numbers and floating point numbers a sign can be added, and for floating point numbers an integer number can be added as exponent.

2.2 The Structure of the Specification

For the readability of the specification, we will first give the modular structure of the specification.

```
Strings

Numerical-Library imports

  Booleans

  Bits                    imports Booleans
  Bit-Strings             imports Bits, Strings
  Binaries                imports Bit-Strings

  Digits
  Digit-Strings           imports Booleans, Digits, Strings
  Natural-Representations imports Digit-Strings
  Natural-Addition        imports Natural-Representations
  Natural-Inequalities    imports Natural-Representations
  Natural-Multiplication  imports Natural-Addition
  Natural-Subtraction     imports Natural-Addition, Natural-Inequalities
  Digit-String-Functions  imports Natural-Subtraction
  Naturals                imports Natural-Multiplication, Digit-String-Functions

  Conversion-Functions    imports Binaries, Naturals

  Signs                   imports Booleans
  Integers                imports Naturals, Signs

  Natural-Names           imports Naturals
  Floating-Point-Repr     imports Integers
  Floating-Points-M-P     imports Floating-Point-Repr
  Floating-Points-3-4     imports Natural-Names, Floating-Points-M-P

Character-Library imports

  Characters              imports Naturals
  Character-Strings       imports Characters, Strings

Queues                    imports Naturals

Tables                    imports Naturals

PSF-Library               imports Strings, Numerical-Library,
                                  Character-Library, Queues, Tables
```

3 Strings

The first specification we give is the specification of the data types STRING and ITEM in the module *Strings*. The sort ITEM is specified as a parameter because the strings will form the basis for all numerical types as well as the character strings. Using a parameter ITEM makes it possible to construct strings of bits, digits, characters and so on with the help of this module. Strings are constructed from items with a string constructor ^, and a number of operations on strings are defined.

The string constructor ^ is specified both unary and binary, because the use of a binary function only would lead to the introduction of a *default item*, making the specification of strings with one item possible. This default item would lead to more (and more complicated) equations for all operations on strings.

- *first* is the leftmost item of a string,
- *last* is the rightmost item of a string,
- *tail* is the part of a string that follows *first* (defining the tail of a string with one element as the identity requires special care when applying this function to strings),
- *concat* concatenates two strings,
- *reverse* is the reverse of its input string.

```
data module Strings
begin

  parameters String-Parameters
    begin

      sorts
        ITEM

  end String-Parameters

  exports
    begin

      sorts
        STRING

      functions
        ^_       : ITEM            -> STRING
        _^_      : STRING # ITEM   -> STRING
        first    : STRING          -> ITEM
        last     : STRING          -> ITEM
        tail     : STRING          -> STRING
```

```
    concat  : STRING # STRING -> STRING
    reverse : STRING          -> STRING

  end

variables
  item, item1, item2       :  -> ITEM
  string, string1, string2 :  -> STRING

equations
  ['] first(^item) = item
  ['] first(string^item) = first(string)

  ['] last(^item) = item
  ['] last(string^item) = item

  ['] tail(^item) = ^item
  ['] tail(^item1^item2) = ^item2
  ['] tail(string^item1^item2) = tail(string^item1)^item2

  ['] concat(string,^item) = string^item
  ['] concat(string1,string2^item) = concat(string1,string2)^item

  ['] reverse(^item) = ^item
  ['] reverse(string^item) = concat(^item,reverse(string))

end Strings
```

4 Booleans

The booleans are specified in a straightforward way. The two constants *true* and *false* are defined as well as the well-known functions on the booleans: *and* (conjunction), *or* (disjunction) and *not* (negation). Moreover, an equality function *eq* is specified, to make it possible to decide equality between two boolean expressions.

```
data module Booleans
begin

  exports
    begin

      sorts
        BOOLEAN

      functions
```

```
      true  :                      -> BOOLEAN
      false :                      -> BOOLEAN
      eq    : BOOLEAN # BOOLEAN    -> BOOLEAN
      not   : BOOLEAN              -> BOOLEAN
      and   : BOOLEAN # BOOLEAN    -> BOOLEAN
      or    : BOOLEAN # BOOLEAN    -> BOOLEAN

    end

  equations
    ['] eq(true,true) = true
    ['] eq(false,false) = true
    ['] eq(true,false) = false
    ['] eq(false,true) = false

    ['] not(true) = false
    ['] not(false) = true

    ['] and(true,true) = true
    ['] and(true,false) = false
    ['] and(false,true) = false
    ['] and(false,false) = false

    ['] or(true,true) = true
    ['] or(true,false) = true
    ['] or(false,true) = true
    ['] or(false,false) = false

end Booleans
```

5 Binary Numbers

Binary numbers are defined as a constructor *bin*, parameterised with a string of bits. The bits are defined as the constants *0'* and *1'*.

5.1 Bits

The bits consist of the constants *0'* and *1'*. There is one function that could be regarded a standard function on bits; the *bit-inv* function, which inverts the value of a bit. The special function *xor* is defined for the specification of addition and subtraction on the binary numbers.

```
data module Bits
begin

  exports
```

```
    begin

      sorts
        BIT

      functions
        0'         :           -> BIT
        1'         :           -> BIT
        eq         : BIT # BIT -> BOOLEAN
        xor        : BIT # BIT -> BIT
        bit-inv    : BIT       -> BIT

    end

  imports
    Booleans

  equations
    ['] eq(0',0') = true
    ['] eq(0',1') = false
    ['] eq(1',0') = false
    ['] eq(1',1') = true

    ['] xor(0',0') = 0'
    ['] xor(0',1') = 1'
    ['] xor(1',0') = 1'
    ['] xor(1',1') = 0'

    ['] bit-inv(0') = 1'
    ['] bit-inv(1') = 0'

end Bits
```

5.2 Bit Strings

Bit strings are constructed from bits with the string operator ^. Some additional operations on bit strings are defined: *cut-rightzeros* cuts all zero bits from the right side of a bit string, and *cut-leftzeros* cuts all zero bits from the left side of a bit string. The equality function *eq* on bit strings is defined straightforwardly, but the greater than function *gt* is less obvious: a bit string *bstr1* is greater than a bit string *bstr2* when *bstr1* is longer than *bstr2*. If the strings are of equal length, the strings are interpreted as binary numbers.

```
data module Bit-Strings
begin

  exports
```

```
    begin

      functions
        cut-rightzeros : BIT-STRING              -> BIT-STRING
        cut-leftzeros  : BIT-STRING              -> BIT-STRING
        eq             : BIT-STRING # BIT-STRING -> BOOLEAN
        gt             : BIT-STRING # BIT-STRING -> BOOLEAN

    end

  imports
    Strings {String-Parameters bound by [ITEM -> BIT] to Bits
             renamed by [STRING -> BIT-STRING]}

  variables
    bstr, bstr1, bstr2 :  -> BIT-STRING
    bit, bit1, bit2    :  -> BIT

  equations
    ['] cut-rightzeros(^bit) = ^bit
    ['] cut-rightzeros(bstr^0') = cut-rightzeros(bstr)
    ['] cut-rightzeros(bstr^1') = bstr^1'

    ['] cut-leftzeros(bstr) = reverse(cut-rightzeros(reverse(bstr)))

    ['] eq(^bit1,^bit2) = eq(bit1,bit2)
    ['] eq(bstr^bit1,^bit2) = false
    ['] eq(^bit1,bstr^bit2) = false
    ['] eq(bstr1^bit1,bstr2^bit2) = and(eq(bstr1,bstr2),eq(bit1,bit2))

    ['] gt(^0',^0') = false
    ['] gt(^0',^1') = false
    ['] gt(^1',^0') = true
    ['] gt(^1',^1') = false
    ['] gt(^bit1,bstr^bit2) = false
    ['] gt(bstr^bit1,^bit2) = true
    ['] gt(bstr1^bit1,bstr2^bit2) =
          or(gt(bstr1,bstr2),and(eq(bstr1,bstr2),gt(^bit1,^bit2)))

end Bit-Strings
```

```
Examples
reverse(^0'^0'^1'^1'^0') = ^0'^1'^1'^0'^0'
cut-leftzeros(^0'^0'^1'^1'^0') = ^1'^1'^0'
```

5.3 Binaries

A binary number is defined as a constructor *bin*, parameterised with a string of bits. Most functions specified in the module *Binaries* can be considered standard functions on binary numbers such as addition, subtraction, multiplication and so on. As mentioned above, the subtraction function is a monus function, and division by zero yields a zero. Some functions, *add-carry*, *sub-carry*, *bin2bit-str* and *if*, are defined to facilitate the specification of the standard functions. Note that, once the zero bits are removed from the left side of a bit string, the equality and greater than functions on binary numbers can be defined in terms of the corresponding functions on bit strings.

```
data module Binaries
begin

  exports
    begin

      sorts
        BINARY

      functions
        bin         : BIT-STRING                   -> BINARY
        _+_         : BINARY # BINARY              -> BINARY
        _-_         : BINARY # BINARY              -> BINARY
        _*_         : BINARY # BINARY              -> BINARY
        div         : BINARY # BINARY              -> BINARY
        mod         : BINARY # BINARY              -> BINARY
        min         : BINARY # BINARY              -> BINARY
        max         : BINARY # BINARY              -> BINARY
        eq          : BINARY # BINARY              -> BOOLEAN
        gt          : BINARY # BINARY              -> BOOLEAN
        gte         : BINARY # BINARY              -> BOOLEAN
        lt          : BINARY # BINARY              -> BOOLEAN
        lte         : BINARY # BINARY              -> BOOLEAN

        add-carry   : BIT # BIT                    -> BINARY
        sub-carry   : BIT # BIT                    -> BINARY
        bin2bit-str : BINARY                       -> BIT-STRING
        if          : BOOLEAN # BINARY # BINARY -> BINARY

    end

  imports
    Bit-Strings

  variables
```

```
  bit, bit1, bit2    :  -> BIT
  bstr, bstr1, bstr2 :  -> BIT-STRING
  bin1, bin2         :  -> BINARY

equations
  [’] bin(bstr^bit) = bin(cut-leftzeros(bstr^bit))
        when eq(first(bstr),0’) = true

  [’] bin2bit-str(bin(bstr)) = cut-leftzeros(bstr)

  [’] add-carry(0’,0’) = bin(^0’)
  [’] add-carry(0’,1’) = bin(^0’)
  [’] add-carry(1’,0’) = bin(^0’)
  [’] add-carry(1’,1’) = bin(^1’)

  [’] bin(^0’) + bin(^0’) = bin(^0’)
  [’] bin(^1’) + bin(^0’) = bin(^1’)
  [’] bin(^0’) + bin(^1’) = bin(^1’)
  [’] bin(^1’) + bin(^1’) = bin(^1’^0’)
  [’] bin(bstr^bit) + bin(^0’) = bin(bstr^bit)
  [’] bin(^0’) + bin(bstr^bit) = bin(bstr^bit)
  [’] bin(bstr^bit) + bin(^1’) =
        bin(bin2bit-str(bin(bstr) + bin(^bit))^bit-inv(bit))
  [’] bin(^1’) + bin(bstr^bit) =
        bin(bin2bit-str(bin(bstr) + bin(^bit))^bit-inv(bit))
  [’] bin(bstr1^bit1) + bin(bstr2^bit2) =
        bin(bin2bit-str((bin(bstr1) + bin(bstr2)) + add-carry(bit1,bit2))^
            xor(bit1,bit2))

  [’] sub-carry(0’,0’) = bin(^0’)
  [’] sub-carry(0’,1’) = bin(^1’)
  [’] sub-carry(1’,0’) = bin(^0’)
  [’] sub-carry(1’,1’) = bin(^0’)

  [’] bin(bstr1) - bin(bstr2) = bin(^0’)
        when gte(bin(bstr2),bin(bstr1)) = true
  [’] bin(bstr) - bin(^0’) = bin(bstr)
  [’] bin(bstr^bit) - bin(^1’) =
        bin(bin2bit-str(bin(bstr) - bin(^bit-inv(bit)))^bit-inv(bit))
        when gt(bin(bstr^bit),bin(^1’)) = true
  [’] bin(^1’) - bin(bstr^bit) = bin(^1’)
        when eq(bin(bstr^bit),bin(^0’)) = true
  [’] bin(bstr1^bit1) - bin(bstr2^bit2) =
       bin(bin2bit-str((bin(bstr1) - bin(bstr2)) - sub-carry(bit1,bit2))^
           xor(bit1,bit2))
```

```
      when gt(bin(bstr1^bit1),bin(bstr2^bit2)) = true

  [’] bin(bstr) * bin(^0’) = bin(^0’)
  [’] bin(bstr) * bin(^1’) = bin(bstr)
  [’] bin(bstr1) * bin(bstr2^bit) =
      bin(bin2bit-str(bin(bstr1) * bin(bstr2))^0’) + (bin(bstr1) * bin(^bit))

  [’] div(bin1,bin2) = bin(^0’)
      when or(eq(bin2,bin(^0’)),gt(bin2,bin1)) = true
  [’] div(bin1,bin2) = div(bin1 - bin2,bin2) + bin(^1’)
      when or(eq(bin2,bin(^0’)),gt(bin2,bin1)) = false

  [’] mod(bin1,bin2) = bin(^0’)
      when or(eq(bin2,bin(^0’)),eq(bin1,bin2)) = true
  [’] mod(bin1,bin2) = bin1
      when and(gt(bin2,bin(^0’)),gt(bin2,bin1)) = true
  [’] mod(bin1,bin2) = mod(bin1 - bin2,bin2)
      when and(gt(bin2,bin(^0’)),gt(bin1,bin2)) = true

  [’] if(true,bin1,bin2) = bin1
  [’] if(false,bin1,bin2) = bin2

  [’] min(bin1,bin2) = if(gt(bin2,bin1),bin1,bin2)
  [’] max(bin1,bin2) = if(gt(bin1,bin2),bin1,bin2)

  [’] eq(bin(bstr1),bin(bstr2)) = eq(cut-leftzeros(bstr1),cut-leftzeros(bstr2))
  [’] gt(bin(bstr1),bin(bstr2)) = gt(cut-leftzeros(bstr1),cut-leftzeros(bstr2))
  [’] gte(bin1,bin2) = not(gt(bin2,bin1))
  [’] lt(bin1,bin2)  = gt(bin2,bin1)
  [’] lte(bin1,bin2) = gte(bin2,bin1)

end Binaries


Examples:
bin(^1’^1’^1’^1’^1’^1’^1’) + bin(^0’^1’) = bin(^1’^0’^0’^0’^0’^0’^0’^0’)
bin(^0’^1’^1’^1’) - bin(^1’^0’^0’) = bin(^1’^1’)
bin(^1’^0’^0’) * bin(^0’^1’^1’^1’) = bin(^1’^1’^1’^0’^0’)
div(bin(^1’^1’^0’^0’),bin(^1’^0’)) = bin(^1’^1’^0’)
gte(bin(^1’^0’^0’),bin(^0’^1’^1’^1’)) = false
```

6 Natural Numbers

Natural numbers are defined as strings of digits, embedded in a constructor *nat*. The operations defined on natural numbers will form the basis for operations on integer numbers and floating point numbers as well.

Examples of the term rewriting of the functions on natural numbers are all given in section 6.9.

6.1 Digits

The digits range from *0* to *9*.

```
data module Digits
begin

  exports
    begin

      sorts
        DIGIT

      functions
        0 :  -> DIGIT
        1 :  -> DIGIT
        2 :  -> DIGIT
        3 :  -> DIGIT
        4 :  -> DIGIT
        5 :  -> DIGIT
        6 :  -> DIGIT
        7 :  -> DIGIT
        8 :  -> DIGIT
        9 :  -> DIGIT

    end

end Digits
```

6.2 Digit Strings

The digit strings will form the basis for the natural numbers, as well as for the integer and floating point numbers. Again, the data types STRING and ITEM are used. The functions in the module *Digit-Strings* have their equivalent in *Bit-strings*. The dots in the specification stand for the large table of definitions of equality between digit strings of length one.

```
data module Digit-Strings
begin
```

```
  exports
    begin

      functions
        cut-rightzeros : DIGIT-STRING                -> DIGIT-STRING
        cut-leftzeros  : DIGIT-STRING                -> DIGIT-STRING
        eq             : DIGIT-STRING # DIGIT-STRING -> BOOLEAN

    end

  imports
    Booleans,
    Strings {String-Parameters bound by [ITEM -> DIGIT] to Digits
             renamed by [STRING -> DIGIT-STRING]}

  variables
    dstr, dstr1, dstr2 :   -> DIGIT-STRING
    dig, dig1, dig2    :   -> DIGIT

  equations
    ['] cut-rightzeros(^dig) = ^dig
    ['] cut-rightzeros(dstr^0) = cut-rightzeros(dstr)
    ['] cut-rightzeros(dstr^1) = dstr^1
    ['] cut-rightzeros(dstr^2) = dstr^2
    ['] cut-rightzeros(dstr^3) = dstr^3
    ['] cut-rightzeros(dstr^4) = dstr^4
    ['] cut-rightzeros(dstr^5) = dstr^5
    ['] cut-rightzeros(dstr^6) = dstr^6
    ['] cut-rightzeros(dstr^7) = dstr^7
    ['] cut-rightzeros(dstr^8) = dstr^8
    ['] cut-rightzeros(dstr^9) = dstr^9

    ['] cut-leftzeros(dstr) = reverse(cut-rightzeros(reverse(dstr)))

    ['] eq(^dig,^dig) = true

    ['] eq(^0,^1) = false
    ...
    ['] eq(^9,^8) = false

    ['] eq(dstr^dig1,^dig2) = false
    ['] eq(^dig1,dstr^dig2) = false
    ['] eq(dstr1^dig1,dstr2^dig2) = and(eq(dstr1,dstr2),eq(^dig1,^dig2))

end Digit-Strings
```

```
Examples
concat(^7^5^4^8,^4^1^0) = ^7^5^4^8^4^1^0
reverse(^1^2^3^4^5) = ^5^4^3^2^1
cut-rightzeros(^1^5^0) = ^1^5
```

6.3 The Representation of Natural Numbers

The module *Natural-Representations* is a simple one. It defines a natural number as a constructor *nat* with a digit string as its argument. A conversion *nat2dig-str* from natural numbers to digit strings is also included.

```
data module Natural-Representations
begin

  exports
    begin

      sorts
        NATURAL

      functions
        nat          : DIGIT-STRING -> NATURAL
        nat2dig-str  : NATURAL      -> DIGIT-STRING

    end

  imports
    Digit-Strings

  variables
    dstr : -> DIGIT-STRING
    dig  : -> DIGIT

  equations
    ['] nat(dstr^dig) = nat(cut-leftzeros(dstr^dig))
          when eq(^first(dstr),^0) = true

    ['] nat2dig-str(nat(dstr)) = cut-leftzeros(dstr)

end Natural-Representations
```

6.4 Addition of Natural Numbers

The addition of natural numbers is specified straightforwardly. An extra function + on digit strings is defined so that we do not have a loss of efficiency in a match of each intermediate result with the function *nat* and the corresponding test on leading zeros in the digit string (see *Natural-Representations*). Note that the equation for *carry1* is also specified for digit strings with length greater than one. This is to make the function always rewrite to a digit string in string representation.

```
data module Natural-Addition
begin

  exports
    begin

      functions
        _+_    : NATURAL # NATURAL           -> NATURAL
        _+_    : DIGIT-STRING # DIGIT-STRING -> DIGIT-STRING
        carry1 : DIGIT-STRING                -> DIGIT-STRING

    end

  imports
    Natural-Representations

  variables
    dig, dig1, dig2    :  -> DIGIT
    dstr, dstr1, dstr2 :  -> DIGIT-STRING

  equations
    ['] nat(dstr1) + nat(dstr2) = nat(dstr1 + dstr2)

    ['] carry1(^dig) = ^0
    ['] carry1(^dig1^dig2) = ^dig1
    ['] carry1(dstr^dig1^dig2) =^0

    ['] (^dig) + (^0) = (^dig)

    ['] (^0) + (^dig) = (^dig)

    ['] (^1) + (^1) = (^2)
    ...
    ['] (^9) + (^9) = (^1^8)

    ['] (^dig1) + (dstr^dig2) =
          (dstr + carry1((^dig1) + (^dig2)))^last((^dig1) + (^dig2))
```

```
    ['] (dstr^dig1) + (^dig2) =
          (dstr + carry1((^dig1) + (^dig2)))^last((^dig1) + (^dig2))

    ['] (dstr1^dig1) + (dstr2^dig2) =
          (((dstr1) + (dstr2)) + carry1((^dig1) + (^dig2)))^
          last((^dig1) + (^dig2))

end Natural-Addition
```

6.5 Inequalities between Natural Numbers

The equality function on natural numbers is directly based on the equality function on digit strings from the module *Digit-Strings.* The greater than function on the natural numbers is also based on a greater than function on digit strings. Note the similarity with the *eq* and *gt* functions as defined on bit strings and binary numbers.

```
data module Natural-Inequalities
begin

  exports
    begin

      functions
        eq    : NATURAL # NATURAL           -> BOOLEAN
        gt    : NATURAL # NATURAL           -> BOOLEAN
        gte   : NATURAL # NATURAL           -> BOOLEAN
        lt    : NATURAL # NATURAL           -> BOOLEAN
        lte   : NATURAL # NATURAL           -> BOOLEAN
        gt    : DIGIT-STRING # DIGIT-STRING -> BOOLEAN

    end

  imports
    Natural-Representations

  variables
    dig, dig1, dig2    :  -> DIGIT
    dstr, dstr1, dstr2 :  -> DIGIT-STRING
    nat1, nat2         :  -> NATURAL

  equations
    ['] eq(nat(dstr1),nat(dstr2)) =
          eq(cut-leftzeros(dstr1),cut-leftzeros(dstr2))
    ['] gt(nat(dstr1),nat(dstr2)) =
          gt(cut-leftzeros(dstr1),cut-leftzeros(dstr2))
```

```
    ['] gte(nat1,nat2) = not(gt(nat2,nat1))
    ['] lt(nat1,nat2)  = gt(nat2,nat1)
    ['] lte(nat1,nat2) = gte(nat2,nat1)

    ['] gt(^1,^2) = false
    ...
    ['] gt(^9,^8) = true

    ['] gt(^0,^dig) = false
    ['] gt(^dig,^0) = not(eq(^dig,^0))
    ['] gt(^dig,^dig)  = false

    ['] gt(dstr^dig1,^dig2) = true
    ['] gt(^dig1,dstr^dig2) = false
    ['] gt(dstr1^dig1,dstr2^dig2) =
          or(gt(dstr1,dstr2),and(eq(dstr1,dstr2),gt(^dig1,^dig2)))

end Natural-Inequalities
```

6.6 Subtraction of Natural Numbers

The subtraction of natural numbers is based on the subtraction of digit strings, which is also specified in the module *Natural-Subtraction*.

```
data module Natural-Subtraction
begin

  exports
    begin

      functions
        _-_       : NATURAL # NATURAL            -> NATURAL
        _-_       : DIGIT-STRING # DIGIT-STRING  -> DIGIT-STRING
        carry2    : DIGIT # DIGIT                -> DIGIT-STRING
        carry3    : DIGIT # DIGIT                -> DIGIT-STRING

    end

  imports
    Natural-Addition, Natural-Inequalities

  variables
    dig, dig1, dig2    : -> DIGIT
    dstr, dstr1, dstr2 : -> DIGIT-STRING

  equations
```

```
    ['] nat(dstr1) - nat(dstr2) = nat(dstr1 - dstr2)
          when gt(nat(dstr1),nat(dstr2)) = true
    ['] nat(dstr1) - nat(dstr2) = nat(^0)
          when gt(nat(dstr1),nat(dstr2)) = false

    ['] carry2(dig1,dig2) = ^0 when gt(^dig2,^dig1) = false
    ['] carry2(dig1,dig2) = ^1 when gt(^dig2,^dig1) = true

    ['] carry3(dig1,dig2) = (^dig1) - (^dig2)
          when gt(^dig2,^dig1) = false
    ['] carry3(dig1,dig2) = ((^9) - ((^dig2) - (^dig1))) + (^1)
          when gt(^dig2,^dig1) = true

    ['] (^dig1) - (^dig2) = (^0) when gte(nat(^dig2),nat(^dig1)) = true

    ['] (^dig) - (^0) = (^dig)

    ['] (^2) - (^1) = (^1)
    ...
    ['] (^9) - (^8) = (^1)

    ['] (^dig1) - (dstr^dig2) =
          (((^0) - (dstr)) - carry2(dig1,dig2))^last(carry3(dig1,dig2))

    ['] (dstr^dig1) - (^dig2) =
          (dstr - carry2(dig1,dig2))^last(carry3(dig1,dig2))

    ['] (dstr1^dig1) - (dstr2^dig2) =
          ((dstr1 - carry2(dig1,dig2)) - dstr2)^last(carry3(dig1,dig2))

end Natural-Subtraction
```

6.7 Multiplication of Natural Numbers

The multiplication of natural numbers is based on the multiplication of digit strings.

```
data module Natural-Multiplication
begin

  exports
    begin

      functions
        _*_ : NATURAL # NATURAL           -> NATURAL
        _*_ : DIGIT-STRING # DIGIT-STRING -> DIGIT-STRING
```

```
    end

  imports
    Natural-Addition

  variables
    dig, dig1, dig2     :  -> DIGIT
    dstr, dstr1, dstr2 :  -> DIGIT-STRING

  equations
    ['] nat(dstr1) * nat(dstr2) = nat(dstr1 * dstr2)

    ['] (^dig) * (^0) = (^0)
    ['] (^0) * (^dig) = (^0)

    ['] (^dig) * (^1) = (^dig)
    ['] (^1) * (^dig) = (^dig)

    ['] (^2) * (^2) = (^4)
    ...
    ['] (^9) * (^9) = (^8^1)

    ['] (dstr^dig1) * (^dig2) =
          (((dstr) * (^dig2)) + carry1((^dig1) * (^dig2)))^
          last((^dig1) * (^dig2))

    ['] (dstr1) * (dstr2^dig) = (((dstr1) * (dstr2))^0) + ((dstr1) * (^dig))

end Natural-Multiplication
```

6.8 Functions on Digit Strings

Now some operations on digit strings that depend on numbers can be specified. For reasons of performance, we did not base these operations on binary numbers and place these operations in the module *Digit-Strings*. On a higher level of the specification, such as in the next subsection, it can be observed that we save ourselves this way from operations on natural numbers based on binary numbers and from conversions from natural numbers to binary numbers and back. The functions on digit strings are particularly interesting because they will make it possible to specify functions on naturals rather in terms of manipulations of (the structure of) digit strings than as straightforward, but inefficient calculations.

- *dstr-length* calculates the length of a digit string,
- *dstr-fill* adds a given number of zeros at the right side of a digit string,
- *dstr-cut* cuts a given number of digits from the right side of a digit string,

- *dstr-tail* is that part of a digit string when a given number of digits is cut from the left of this digit string.

```
data module Digit-String-Functions
begin

  exports
    begin

      functions
        dstr-length  : DIGIT-STRING           -> NATURAL
        dstr-fill    : NATURAL # DIGIT-STRING -> DIGIT-STRING
        dstr-cut     : NATURAL # DIGIT-STRING -> DIGIT-STRING
        dstr-tail    : NATURAL # DIGIT-STRING -> DIGIT-STRING

    end

  imports
    Natural-Subtraction

  variables
    dstr :   -> DIGIT-STRING
    dig  :   -> DIGIT
    nat  :   -> NATURAL

  equations
    ['] dstr-length(^dig) = nat(^1)
    ['] dstr-length(dstr^dig) = dstr-length(dstr) + nat(^1)

    ['] dstr-fill(nat(^0),dstr) = dstr
    ['] dstr-fill(nat,dstr) = dstr-fill(nat - nat(^1),dstr^0)
          when gt(nat,nat(^0)) = true

    ['] dstr-cut(nat(^0),dstr) = dstr
    ['] dstr-cut(nat,dstr^dig) = dstr-cut(nat - nat(^1),dstr)
          when and(gt(dstr-length(dstr^dig),nat),gt(nat,nat(^0))) = true
    ['] dstr-cut(nat,dstr) = ^0
          when gte(nat,dstr-length(dstr)) = true

    ['] dstr-tail(nat(^0),dstr) = dstr
    ['] dstr-tail(nat,dstr) = reverse(dstr-cut(nat,reverse(dstr)))

end Digit-String-Functions
```

```
Examples
dstr-length(^7^5^1^0) = nat(^4)
dstr-fill(nat(^2),^5^7^8^2) = ^5^7^8^2^0^0
dstr-cut(nat(^2),^5^7^8^2) = ^5^7
dstr-tail(nat(^2),^1^2^3^4^5) = ^3^4^5
```

6.9 Special Functions on Natural Numbers

Some standard functions on natural numbers are specified. Most of them are easy to understand. The *div* function needs some explanation. It is possible to only specify a more pleasant looking version of this function following its standard definition $x = y * div(x, y) + mod(x, y)$, but when the arguments have significant difference in size this approach becomes very inefficient. However, this standard approach of division, which is based on repeated subtraction, forms the core of a division algorithm (see *short-div* and also the *div* function on binary numbers).

The division algorithm is as follows: the lengths of the arguments are compared and then it is decided whether the *short-div* or the *long-div* is used. The *long-div* function is the long division from the elementary school. Some tricks are used to increase the efficiency such as the reversal of digit strings. This makes it possible to avoid some applications of the (relatively inefficient) *dstr-first* and *dstr-tail* functions. Note that the *long-div* function gives correct results only when the length of the divisor (digit string) is shorter than that of the numerator (digit string).

```
data module Naturals
begin

  exports
    begin

      functions
        div       : NATURAL # NATURAL                             -> NATURAL
        mod       : NATURAL # NATURAL                             -> NATURAL
        min       : NATURAL # NATURAL                             -> NATURAL
        max       : NATURAL # NATURAL                             -> NATURAL
        if        : BOOLEAN # NATURAL # NATURAL                   -> NATURAL
        short-div : NATURAL # NATURAL                             -> NATURAL
        long-div  : DIGIT-STRING # DIGIT-STRING # DIGIT-STRING -> DIGIT-STRING

    end

  imports
    Digit-String-Functions, Natural-Multiplication

  variables
    nat1, nat2                :   -> NATURAL
    dstr, dstr1, dstr2, dstr3 :   -> DIGIT-STRING
```

```
  dig                         :   -> DIGIT

 equations
  ['] div(nat(dstr1),nat(dstr2)) = short-div(nat(dstr1),nat(dstr2))
        when or(eq(nat(dstr2),nat(^0)),gt(nat(dstr2^0),nat(dstr1))) = true
  ['] div(nat(dstr1),nat(dstr2)) = nat(reverse(
        long-div(dstr-cut(dstr-length(cut-leftzeros(dstr1)) -
                            dstr-length(cut-leftzeros(dstr2)),
                          dstr1),
                 reverse(dstr-tail(dstr-length(cut-leftzeros(dstr2)),
                                        cut-leftzeros(dstr1))),
                 dstr2)))
        when or(eq(nat(dstr2),nat(^0)),gt(nat(dstr2^0),nat(dstr1))) = false

  ['] short-div(nat1,nat2) = nat(^0)
        when or(eq(nat2,nat(^0)),gt(nat2,nat1)) = true
  ['] short-div(nat1,nat2) = short-div(nat1 - nat2,nat2) + nat(^1)
        when or(eq(nat2,nat(^0)),gt(nat2,nat1)) = false

  ['] long-div(dstr1,dstr2^dig,dstr3) =
        long-div(nat2dig-str(nat(dstr1) -
                             (nat(dstr3) * short-div(nat(dstr1),nat(dstr3))))^dig,
                 dstr2,
                 dstr3)^
        last(nat2dig-str(short-div(nat(dstr1),nat(dstr3))))

  ['] long-div(dstr1,^dig,dstr2) =
          nat2dig-str(
            short-div(
              nat(nat2dig-str(nat(dstr1) -
                              (nat(dstr2) * short-div(nat(dstr1),nat(dstr2))))^dig),
              nat(dstr2)))^
          last(nat2dig-str(short-div(nat(dstr1),nat(dstr2))))

  ['] mod(nat1,nat2) = nat1 - (div(nat1,nat2) * nat2)

  ['] if(true,nat1,nat2) = nat1
  ['] if(false,nat1,nat2) = nat2

  ['] max(nat1,nat2) = if(gt(nat1,nat2),nat1,nat2)

  ['] min(nat1,nat2) = if(gt(nat1,nat2),nat2,nat1)

end Naturals
```

```
Examples
nat(^0^3^5) - nat(^4^1) = nat(^0)
nat(^1^3^7^3^4^5) * nat(^4^2^6) = nat(^5^8^5^0^8^9^7^0)
nat(^5^8^5^0^9^3^1^6) - nat(^5^8^5^0^8^9^7^0) = nat(^3^4^6)
div(nat(^5^8^5^0^9^3^1^6),nat(^4^2^6)) = nat(^1^3^7^3^4^5)
mod(nat(^5^8^5^0^9^3^1^6),nat(^4^2^6)) = nat(^3^4^6)
max(nat(^5^7),nat(^0^7^4^1)) = nat(^7^4^1)
```

6.10 Names of Natural Numbers

In *Natural-Names* the names *nat-zero* to *nat-sixteen* are defined in terms of the *nat(dstr)* representation. The choice of sixteen as the upper bound is completely arbitrary.

```
data module Natural-Names
begin

  exports
    begin
      functions
        nat-zero     : -> NATURAL
        ...
        nat-sixteen  : -> NATURAL

    end

  imports
    Naturals

  equations
    ['] nat-zero     = nat(^0)
    ...
    ['] nat-sixteen  = nat(^1^6)

end Natural-Names
```

7 Conversion Functions

In *Conversion-Functions* conversions between natural numbers and binary numbers numbers are specified.

```
data module Conversion-Functions
begin

  exports
    begin
```

```
    functions
      bin-ten :               -> BINARY
      bin2nat : BINARY        -> NATURAL
      nat2bin : NATURAL       -> BINARY

  end

imports
  Binaries, Naturals

variables
  bit  : -> BIT
  dig  : -> DIGIT
  bstr : -> BIT-STRING
  dstr : -> DIGIT-STRING

equations
  [’] bin-ten = bin(^1’^0’^1’^0’)

  [’] nat2bin(nat(^0)) = bin(^0’)
  [’] nat2bin(nat(^1)) = bin(^1’)
  [’] nat2bin(nat(^2)) = bin(^1’^0’)
  [’] nat2bin(nat(^3)) = bin(^1’^1’)
  [’] nat2bin(nat(^4)) = bin(^1’^0’^0’)
  [’] nat2bin(nat(^5)) = bin(^1’^0’^1’)
  [’] nat2bin(nat(^6)) = bin(^1’^1’^0’)
  [’] nat2bin(nat(^7)) = bin(^1’^1’^1’)
  [’] nat2bin(nat(^8)) = bin(^1’^0’^0’^0’)
  [’] nat2bin(nat(^9)) = bin(^1’^0’^0’^1’)
  [’] nat2bin(nat(dstr^dig)) =
        (nat2bin(nat(dstr)) * bin-ten) + nat2bin(nat(^dig))

  [’] bin2nat(bin(^0’)) = nat(^0)
  [’] bin2nat(bin(^1’)) = nat(^1)
  [’] bin2nat(bin(bstr^bit)) =
        (bin2nat(bin(bstr)) * nat(^2)) + bin2nat(bin(^bit))

end Conversion-Functions


Examples
nat2bin(nat(^2^5^7)) = bin(^1’^0’^0’^0’^0’^0’^0’^0’^1’)
bin2nat(bin(^1’^0’^0’^0’^0’^0’^0’^0’^1’)) = nat(^2^5^7)
```

8 Integer Numbers

Integer numbers are specified as a constructor *int* with a sign and a digit string as its arguments. The specification is primarily based on the specification of the natural numbers.

8.1 Signs

Signs are defined as *pos* and *neg*. An equality function *eq* and a multiplication are specified.

```
data module Signs
begin

  exports
    begin

      sorts
        SIGN

      functions
        pos        :               -> SIGN
        neg        :               -> SIGN
        sign-inv   : SIGN          -> SIGN
        _*_        : SIGN # SIGN -> SIGN
        eq         : SIGN # SIGN -> BOOLEAN

    end

  imports
    Booleans

  equations
    ['] sign-inv(pos) = neg
    ['] sign-inv(neg) = pos

    ['] pos * pos = pos
    ['] neg * neg = pos
    ['] pos * neg = neg
    ['] neg * pos = neg

    ['] eq(pos,pos) = true
    ['] eq(neg,neg) = true
    ['] eq(pos,neg) = false
    ['] eq(neg,pos) = false

end Signs
```

8.2 The Representation of and Functions on Integer Numbers

The specification on integer numbers is based on the specification of the natural numbers, only the signs require some extra attention.

Because the interpretation of the *div* and *mod* functions on integer numbers with negative signs is not straightforward, we chose a simple solution by defining these functions equal to zero in this case.

```
data module Integers
begin

  exports
    begin

      sorts
        INTEGER

      functions
        int     : SIGN # DIGIT-STRING         -> INTEGER
        int     : SIGN # NATURAL              -> INTEGER
        int     : DIGIT-STRING                -> INTEGER
        int     : NATURAL                     -> INTEGER
        int-inv : INTEGER                     -> INTEGER
        _-_     : INTEGER # INTEGER           -> INTEGER
        _+_     : INTEGER # INTEGER           -> INTEGER
        _*_     : INTEGER # INTEGER           -> INTEGER
        div     : INTEGER # INTEGER           -> INTEGER
        mod     : INTEGER # INTEGER           -> INTEGER
        eq      : INTEGER # INTEGER           -> BOOLEAN
        gt      : INTEGER # INTEGER           -> BOOLEAN
        gte     : INTEGER # INTEGER           -> BOOLEAN
        lt      : INTEGER # INTEGER           -> BOOLEAN
        lte     : INTEGER # INTEGER           -> BOOLEAN
        min     : INTEGER # INTEGER           -> INTEGER
        max     : INTEGER # INTEGER           -> INTEGER
        if      : BOOLEAN # INTEGER # INTEGER -> INTEGER
        abs     : INTEGER                     -> INTEGER
        int2nat : INTEGER                     -> NATURAL

    end

  imports
    Naturals, Signs

  variables
    dig                 :  -> DIGIT
```

```
  dstr, dstr1, dstr2 :  -> DIGIT-STRING
  s, t               :  -> SIGN
  int1, int2         :  -> INTEGER

equations
  [’] int(s,dstr^dig) = int(s,cut-leftzeros(dstr^dig))
        when eq(^first(dstr),^0) = true
  [’] int(dstr) = int(pos,dstr)
  [’] int(nat(dstr)) = int(pos,dstr)
  [’] int(s,nat(dstr)) = int(s,dstr)

  [’] int-inv(int(s,dstr)) = int(sign-inv(s),dstr)

  [’] int(neg,^0) = int(pos,^0)

  [’] int1 - int2 = int1 + int-inv(int2)

  [’] int(s,dstr1) + int(t,dstr2) = int(s,nat(dstr1) + nat(dstr2))
        when eq(s,t) = true
  [’] int(s,dstr1) + int(t,dstr2) = int(s,nat(dstr1) - nat(dstr2))
        when and(not(eq(s,t)),gte(nat(dstr1),nat(dstr2))) = true
  [’] int(s,dstr1) + int(t,dstr2) = int(t,nat(dstr2) - nat(dstr1))
        when and(not(eq(s,t)),gt(nat(dstr2),nat(dstr1)))  = true

  [’] int(s,dstr1) * int(t,dstr2) = int(s * t,nat(dstr1) * nat(dstr2))

  [’] div(int(pos,dstr1),int(pos,dstr2)) =
        int(pos,div(nat(dstr1),nat(dstr2)))
  [’] div(int(s,dstr1),int(t,dstr2)) = int(pos,^0)
        when and(eq(s,pos),eq(t,pos)) = false

  [’] mod(int(pos,dstr1),int(pos,dstr2)) =
        int(pos,mod(nat(dstr1),nat(dstr2)))
  [’] mod(int(s,dstr1),int(t,dstr2)) = int(pos,^0)
        when and(eq(s,pos),eq(t,pos)) = false

  [’] eq(int(s,dstr1),int(t,dstr2)) =
        or(and(eq(nat(dstr1),nat(dstr2)),eq(s,t)),
           and(eq(nat(dstr1),nat(^0)),eq(nat(dstr2),nat(^0))) )

  [’] gte(int(neg,dstr1),int(neg,dstr2)) = gte(nat(dstr2),nat(dstr1))
  [’] gte(int(pos,dstr1),int(pos,dstr2)) = gte(nat(dstr1),nat(dstr2))
  [’] gte(int(neg,dstr1),int(pos,dstr2)) =
        and(eq(nat(dstr1),nat(^0)),eq(nat(dstr2),nat(^0)))
  [’] gte(int(pos,dstr1),int(neg,dstr2)) = true
```

```
    [’] gt(int1,int2)  = not(gte(int2,int1))
    [’] lt(int1,int2)  = gt(int2,int1)
    [’] lte(int1,int2) = gte(int2,int1)

    [’] if(true,int1,int2)  = int1
    [’] if(false,int1,int2) = int2

    [’] min(int1,int2) = if(gte(int1,int2),int2,int1)
    [’] max(int1,int2) = if(gte(int1,int2),int1,int2)

    [’] abs(int(s,dstr)) = int(pos,dstr)

    [’] int2nat(int(s,dstr)) = nat(dstr)

end Integers


Examples
int(pos,nat(^7^8)) - int(neg,^2^2) = int(pos,^1^0^0)
int(^3) * int(neg,^7^1) = int(neg,^2^1^3)
div(int(pos,^2^5),int(pos,^8)) = int(pos,^3)
mod(int(pos,^2^5),int(pos,^8)) = int(pos,^1)
div(int(neg,^1^0),int(^3)) = int(pos,^0)
mod(int(neg,^1^0),int(^3)) = int(pos,^0)
gte(int(neg,^5),int(neg,^2^4)) = true
abs(int(neg,^0^5^4)) = int(pos,^5^4))
int2nat(int(neg,^0^5^4)) = nat(^5^4)
```

9 Floating Point Numbers

A floating point number is specified as a constructor *fp* with a sign, one or two digit strings, and an integer number as its arguments. The integer number represents the exponent of the floating point number.

9.1 The Representation of Floating Point Numbers

In the module *Floating-Point-Repr* some different (input) formats of floating point numbers are defined to give the user the freedom to choose between different representations. If, for instance, no exponents are desired in the input format, they need not be arguments of *fp*; in this case a zero exponent is implicit. Every floating point number rewrites to an expression of the following form (*normal form equation*):

$$fp(s,\hat{}\,0,dstr,exp)\quad with\ eq(\hat{}\,first(dstr),\hat{}\,0) \rightarrow eq(nat(dstr),\hat{}\,0) \wedge eq(exp,int(pos,\hat{}\,0))$$

```
data module Floating-Point-Repr
begin
```

```
exports
  begin

    sorts
      FP

    functions
      fp        : SIGN # DIGIT-STRING # DIGIT-STRING # INTEGER -> FP
      fp        : SIGN # DIGIT-STRING # DIGIT-STRING           -> FP
      fp        : SIGN # DIGIT-STRING                # INTEGER -> FP
      fp        : SIGN # DIGIT-STRING                          -> FP
      fp        :        DIGIT-STRING # DIGIT-STRING # INTEGER -> FP
      fp        :        DIGIT-STRING # DIGIT-STRING           -> FP
      fp        :        DIGIT-STRING                # INTEGER -> FP
      fp        :        DIGIT-STRING                          -> FP
      fp        :        INTEGER                     # INTEGER -> FP
      fp        :        INTEGER                               -> FP

  end

imports
  Integers

variables
  dig                 :  -> DIGIT
  dstr, dstr1, dstr2 :  -> DIGIT-STRING
  s                   :  -> SIGN
  exp                 :  -> INTEGER

equations
  ['] fp(s,dstr1^dig,dstr2,exp) = fp(s,cut-leftzeros(dstr1^dig),dstr2,exp)
        when eq(^first(dstr1),^0) = true

  ['] fp(s,dstr1,dstr2,exp) =
        fp(s,^0,concat(dstr1,dstr2),exp + int(pos,dstr-length(dstr1)))
        when eq(^first(dstr1),^0) = false

  ['] fp(s,^0,dstr,exp) = fp(s,^0,tail(dstr),exp - int(pos,^1))
        when and(eq(^first(dstr),^0),not(eq(nat(dstr),nat(^0)))) = true

  ['] fp(s,^0,dstr,exp) = fp(pos,^0,dstr,int(pos,^0))
        when and(eq(^first(dstr),^0),not(eq(exp,int(pos,^0)))) = true

  ['] fp(dstr)            = fp(pos,dstr)
```

```
    [’] fp(dstr,exp)        = fp(pos,dstr,exp)
    [’] fp(dstr1,dstr2)     = fp(pos,dstr1,dstr2,int(pos,^0))
    [’] fp(dstr1,dstr2,exp) = fp(pos,dstr1,dstr2,exp)
    [’] fp(s,dstr)          =
          fp(s,^0,cut-leftzeros(dstr),int(pos,dstr-length(cut-leftzeros(dstr))))
    [’] fp(s,dstr,exp)      =
          fp(s,^0,cut-leftzeros(dstr),exp + int(pos,dstr-length(cut-leftzeros(dstr))))
    [’] fp(s,dstr1,dstr2)   = fp(s,dstr1,dstr2,int(pos,^0))
    [’] fp(int(s,dstr),exp) = fp(s,dstr,exp)
    [’] fp(int(s,dstr))     = fp(s,dstr)

end Floating-Point-Repr


Examples
fp(^0^0^1) = fp(pos,^0,^1,int(pos,^1))
fp(^7,^5,int(^2)) = fp(pos,^0,^7^5,int(pos,^3))
fp(^9^9^9) = fp(pos,^0,^9^9^9,int(pos,^3))
fp(^0,^0^0^5^3,int(neg,^7)) = fp(pos,^0,^5^3,int(neg,^9))
```

9.2 Functions on Floating Point Numbers

For the simplification of the functions on floating point numbers a normal form function *nf* is defined, which shapes every floating point number into a fixed format. The variable *nat* symbolises the number of digits behind the point of a normal form. *nf* uses a function *resize* to force a floating point number into the form given by the normal form equation in the previous section.

Every arithmetic function defined on floating point numbers is defined twice. The $+$, $-$, $*$, $/$, *round* and *trunc* functions are meant for the user of the numerical library; they operate properly on all arguments of the sort FP. Their counterparts *add*, *sub*, *mul*, *div*, *fp-round* and *fp-trunc* operate properly only on normal forms of the sort FP as induced by *nf*. The reason for defining two functions for each operation is that it simplifies the specification of the functions significantly.

For reasons of performance, we omitted some conditions from the equations for the hidden functions: conditions on the form of the *second* digit string of the input floating point numbers. For instance, from the equations for *resize* the condition of equation [$*$] is left out. This omission leads to a term rewriting system with non-unique normal forms. For this reason we hide these functions for the users of the PSF library.

Two parameters are defined such that the user of the numerical library is free to choose the length of the external representation *mantisse* and the length of the internal representation *precision* him/herself. Note that for precise results *precision* must be chosen greater than *mantisse*. Any numerical analysis of the precision of the results is left to the user.

```
data module Floating-Points-M-P
begin
```

```
parameters
  Representation-Lengths
  begin

    functions
      mantisse  : -> NATURAL
      precision : -> NATURAL

   end Representation-Lengths

exports
  begin

    functions
      _+_      : FP # FP      -> FP
      _-_      : FP # FP      -> FP
      _*_      : FP # FP      -> FP
      _/_      : FP # FP      -> FP
      round    : FP           -> INTEGER
      trunc    : FP           -> INTEGER
      nf       : NATURAL # FP -> FP

  end

imports
  Floating-Point-Repr

functions
  resize   : NATURAL # FP -> FP
  add      : FP # FP      -> FP
  sub      : FP # FP      -> FP
  mul      : FP # FP      -> FP
  div      : FP # FP      -> FP
  nf-round : FP           -> INTEGER
  nf-trunc : FP           -> INTEGER

variables
  nat                    :  -> NATURAL
  fp, fp1, fp2           :  -> FP
  exp, exp1, exp2        :  -> INTEGER
  s, s1, s2, t, t1, t2   :  -> SIGN
  dstr, dstr1, dstr2     :  -> DIGIT-STRING
  d                      :  -> DIGIT
  m, m1, m2              :  -> DIGIT-STRING
```

```
equations
  [’] nf(nat,fp(s,^0,dstr,exp)) = resize(nat,fp(s,^0,dstr,exp))
        when or(not(eq(^first(dstr),^0)),
                and(eq(nat(dstr),nat(^0)),eq(exp,int(pos,^0)))) = true

  [’] resize(nat(^0),fp) = fp(pos,^0,^0,int(pos,^0))

  [’] resize(nat,fp(s,^0,dstr^d,exp)) = resize(nat,fp(s,^0,dstr,exp))
        when and(gt(dstr-length(dstr),nat),gt(nat,nat(^0))) = true

  [’] resize(nat,fp(s,^0,dstr^d,exp)) = fp(s,^0,dstr,exp)
        when and(eq(dstr-length(dstr),nat),lt(nat(^d),nat(^5))) = true

  [’] resize(nat,fp(s,^0,dstr^d,exp)) = fp(s,^0,dstr + (^1),exp)
        when and(and(eq(dstr-length(dstr),nat),gte(nat(^d),nat(^5))),
                 eq(dstr-length(dstr),dstr-length(dstr + (^1)))) = true

  [’] resize(nat,fp(s,^0,dstr^d,exp)) =
        fp(s,^0,dstr-cut(nat(^1),dstr + (^1)),exp + int(pos,^1))
        when and(and(eq(dstr-length(dstr),nat),gte(nat(^d),nat(^5))),
                 lt(dstr-length(dstr),dstr-length(dstr + (^1)))) = true

  [’] resize(nat,fp(s,^0,dstr,exp)) = fp(s,^0,dstr,exp)
        when eq(dstr-length(dstr),nat) = true

  [’] resize(nat,fp(s,^0,dstr,exp)) = resize(nat,fp(s,^0,dstr^0,exp))
        when gt(nat,dstr-length(dstr)) = true

  [’] fp1 + fp2 = nf(mantisse,add(nf(precision,fp1),nf(precision,fp2)))
  [’] fp1 - fp2 = nf(mantisse,sub(nf(precision,fp1),nf(precision,fp2)))
  [’] fp1 * fp2 = nf(mantisse,mul(nf(precision,fp1),nf(precision,fp2)))
  [’] fp1 / fp2 = nf(mantisse,div(nf(precision,fp1),nf(precision,fp2)))

  [’] round(fp) = nf-round(nf(mantisse,fp))
  [’] trunc(fp) = nf-trunc(nf(mantisse,fp))

  [’] sub(fp1,fp(s,m1,m2,exp)) = add(fp1,fp(sign-inv(s),m1,m2,exp))

  [’] add(fp(s1,^0,m1,int(t1,dstr1)),fp(s2,^0,m2,int(t2,dstr2))) =
        fp(int(s1,dstr-fill(int2nat(int(t1,dstr1) - int(t2,dstr2)),m1)) +
             int(s2,m2),
           int(t2,dstr2) - int(pos,precision))
        when gt(int(t1,dstr1),int(t2,dstr2)) = true

  [’] add(fp(s1,^0,m1,int(t1,dstr1)),fp(s2,^0,m2,int(t2,dstr2))) =
```

```
      fp(int(s2,dstr-fill(int2nat(int(t2,dstr2) - int(t1,dstr1)),m2)) +
           int(s1,m1),
         int(t1,dstr1) - int(pos,precision))
      when gt(int(t2,dstr2),int(t1,dstr1)) = true

  ['] add(fp(s1,^0,m1,int(t1,dstr1)),fp(s2,^0,m2,int(t2,dstr2))) =
      fp(int(s1,m1) + int(s2,m2),int(t1,dstr1) - int(pos,precision))
      when eq(int(t1,dstr1),int(t2,dstr2)) = true

  ['] mul(fp(s1,^0,m1,exp1),fp(s2,^0,m2,exp2)) =
      fp(s1 * s2,nat2dig-str(nat(m1) * nat(m2)),^0,
         exp1 + exp2 - int(pos,nat(^2) * precision))

  ['] div(fp(s1,^0,m1,exp1),fp(s2,^0,m2,exp2)) =
      fp(s1 * s2,
         nat2dig-str(div(nat(dstr-fill(precision,m1)),nat(m2))),
         ^0,exp1 - exp2 - int(pos,precision))

  ['] nf-round(fp(s,^0,m,int(t,dstr))) = int(s,^0)
      when eq(t,neg) = true

  ['] nf-round(fp(s,^0,m,int(t,dstr))) =
     int(s,dstr-cut(mantisse,dstr-fill(nat(dstr),m))) +
     int(s,short-div(
             nat(^last(dstr-cut(mantisse - nat(^1),dstr-fill(nat(dstr),m)))),
             nat(^5)))
        when eq(t,pos) = true

  ['] nf-trunc(fp(s,^0,m,int(t,dstr))) = int(s,^0)
      when eq(t,neg) = true

  ['] nf-trunc(fp(s,^0,m,int(t,dstr))) =
     int(s,dstr-cut(mantisse,dstr-fill(nat(dstr),m)))
        when eq(t,pos) = true

end Floating-Points-M-P
```

9.3 Floating Point Numbers with a Fixed Length

The parameter *mantisse* is bound to *nat-three* and the parameter *precision* is bound to *nat-four*. Of course, any user of the numerical library is free to define as many modules *Floating-Points-x-y* with different instances of the parameters as desired.

```
data module Floating-Points-3-4
begin
```

```
  imports
    Floating-Points-M-P {Representation-Lengths bound by
                           [mantisse -> nat-three, precision -> nat-four]
                           to Natural-Names}

end Floating-Points-3-4
```

```
Examples
fp(^3^1) + fp(neg,^3^0) = fp(pos,^0,^1^0^0,int(pos,^1))
fp(pos,^1^5^9,^7^5^4) - fp(^0,^5^9,int(pos,^2)) = fp(pos,^0,^1^0^1,int(pos,^3))
fp(^1^5^9,^7^5^4) * fp(^5^9,int(pos,^3)) = fp(pos,^0,^9^4^3,int(pos,^7))
fp(pos,^7^5^8^9^1,^5^7^4,int(pos,^7^3)) / fp(int(neg,^5^8^3),int(pos,^4^8)) =
  fp(neg,^0,^1^3^0,int(pos,^2^8))
round(fp(^9^9,^9)) = int(pos,^1^0^0)
round(fp(pos,^5^4,^2^7,int(pos,^1))) = int(pos,^5^4^3)
round(fp(^0,^6)) = int(pos,^1)
trunc(fp(^0,^6)) = int(pos,^0)
```

10 Testing the Numerical Library

A specification of data in PSF can be tested in many ways, just as other types of software. The same question arises, namely: when can it be concluded that the software is correct? Criteria for correctness of the numerical library can be formulated as follows:

- The equations must form a confluent term rewrite system, so the normal forms must be unique,
- The equations must rewrite to as least as possible normal forms,
- The specification has to yield the same numerical results as other programming languages.

Unfortunately, there is no other way to obtain absolute certainty about these things than through extremely elaborate algebraic proofs. An option would be to translate the specification into a μCRL specification and prove statements with the proof theory for this language [GP91]. However, since there are no tools yet to support formal proofs in μCRL we will not attempt this.

The first point, on the confluency of the term rewrite system, is mainly a matter of experience, just as in normal programming practice, and careful specification. Many errors due to non-confluency will automatically be found during tests. The main resource for testing a specification on this criterion is the PSF term term rewriter, which offers the possibility to perform the term rewriting in two directions along the table of equations (in TIL, the Tool Interface Language for PSF). The hidden functions on the floating point numbers do not form a confluent term rewriting system; some conditions are omitted to increase the performance of the functions (see section 9.2).

The second criterion can be tested with the *initial* tool, which calculates the initial algebra of a specification up to a fixed number of elements for each sort. With the initial tool it becomes very plausible, for instance, that all functions on bit string and digit strings rewrite to strings, and that the specification consists of two booleans, two bits and two signs. Also that each natural number is of the form *nat(dstr)*, that each binary is of the form *bin(bstr)*, and so on.

The third criterion, on correct numerical results, is also a matter of testing. Again we can use the term PSF rewriter to verify whether the results are conform our intuition. For each function, boundary cases can be tested, and many random calculations can be made. Moreover, some test runs, involving calculations with many different elements, can be performed in order to obtain more certainty about the correctness of the specification.

For testing the numerical correctness, we specify a module *Test* with some complicated functions and make some calculations. The following functions are specified on natural numbers and binary numbers

- $plus(n) = \sum_{i=0}^{n} n$, this sum is equal to $\frac{1}{2}n(n+1)$
- $fac(n) = n!$
- $minus(m, n)$ is a function that subtracts all numbers less than or equal to n one by one from m, so $minus(m, n) = m - plus(n)$,
- $division(m, n)$ divides m by all numbers less than or equal to n, so $division(fac(k), n) = div(fac(k), fac(n))$.

For testing the floating point numbers we calculate some trigonometric functions up to a reasonable precision. For this purpose, *sin* and *cos* functions are specified as power series. Also a function *sqrt2(n)* is specified, which approximates $\sqrt{2}$ with a continued fraction up to depth n.

```
data module Test
begin

  exports
    begin

      functions
        plus      : NATURAL           -> NATURAL
        minus     : NATURAL # NATURAL -> NATURAL
        fac       : NATURAL           -> NATURAL
        division  : NATURAL # NATURAL -> NATURAL

        plus      : BINARY            -> BINARY
        minus     : BINARY # BINARY   -> BINARY
        fac       : BINARY            -> BINARY
        division  : BINARY # BINARY   -> BINARY
```

```
    even      : NATURAL           -> BOOLEAN
    coeff     : FP # NATURAL      -> FP
    exp       : FP # NATURAL      -> FP
    sin       : FP # NATURAL      -> FP
    cos       : FP # NATURAL      -> FP
    sqrt2     : NATURAL           -> FP

  end

imports
  Conversion-Functions,
  Floating-Points-M-P {Representation-Lengths bound by
                      [mantisse -> nat-six, precision -> nat-eight]
                      to Natural-Names}

variables
  nat, nat1, nat2 : -> NATURAL
  bin, bin1, bin2 : -> BINARY
  x               : -> FP

equations
  [’] plus(nat(^0)) = nat(^0)
  [’] plus(nat) = plus(nat - nat(^1)) + nat when eq(nat,nat(^0)) = false

  [’] minus(nat,nat(^0)) = nat
  [’] minus(nat1,nat2) = minus(nat1 - nat2,nat2 - nat(^1))
      when eq(nat2,nat(^0)) = false

  [’] fac(nat(^0)) = nat(^1)
  [’] fac(nat) = nat * fac(nat - nat(^1)) when eq(nat,nat(^0)) = false

  [’] division(nat,nat(^0)) = nat
  [’] division(nat1,nat2) = division(div(nat1,nat2), nat2 - nat(^1))
      when eq(nat2,nat(^0)) = false

  [’] plus(bin(^0’)) = bin(^0’)
  [’] plus(bin) = plus(bin - bin(^1’)) + bin when eq(bin,bin(^0’)) = false

  [’] minus(bin,bin(^0’)) = bin
  [’] minus(bin1,bin2) = minus(bin1 - bin2,bin2 - bin(^1’))
      when eq(bin2,bin(^0’)) = false

  [’] fac(bin(^0’)) = bin(^1’)
  [’] fac(bin) = bin * fac(bin - bin(^1’)) when eq(bin,bin(^0’)) = false
```

```
    ['] division(bin,bin(^0')) = bin
    ['] division(bin1,bin2) = division(div(bin1,bin2), bin2 - bin(^1'))
          when eq(bin2,bin(^0')) = false

    ['] even(nat) = eq(mod(nat,nat(^2)),nat(^0))

    ['] exp(x,nat(^0)) = fp(^1)
    ['] exp(x,nat) = x * exp(x,nat - nat(^1)) when eq(nat,nat(^0)) = false

    ['] even(nat) = eq(mod(nat,nat(^2)),nat(^0))

    ['] coeff(x,nat) = exp(x,nat)/fp(nat2dig-str(fac(nat)))

    ['] sin(x,nat(^0)) = x
    ['] sin(x,nat) = sin(x,nat - nat(^1)) + coeff(x,(nat(^2) * nat) + nat(^1))
          when and(gt(nat,nat(^0)),even(nat)) = true
    ['] sin(x,nat) = sin(x,nat - nat(^1)) - coeff(x,(nat(^2) * nat) + nat(^1))
          when and(gt(nat,nat(^0)),not(even(nat))) = true

    ['] cos(x,nat(^0)) = fp(^1)
    ['] cos(x,nat) = cos(x,nat - nat(^1)) + coeff(x,nat(^2) * nat)
          when and(gt(nat,nat(^0)),even(nat)) = true
    ['] cos(x,nat) = cos(x,nat - nat(^1)) - coeff(x,nat(^2) * nat)
          when and(gt(nat,nat(^0)),not(even(nat))) = true

    ['] sqrt2(nat(^0)) = fp(^1)
    ['] sqrt2(nat) = fp(^1) + (fp(^1)/(sqrt2(nat - nat(^1)) + fp(^1)))
          when eq(nat,nat(^0)) = false

end Test


Examples of test results:


plus(nat(^3^2^5)) = nat(^5^2^9^7^5)
nat(^3^2^5) * nat(^1^6^3) = nat(^5^2^9^7^5)

minus(plus(nat(^3^2^5)),nat(^1^1^6))  = nat(^4^6^1^8^9)
plus(nat(^3^2^5)) - plus(nat(^1^1^6)) = nat(^4^6^1^8^9)

division(fac(nat(^2^5)),nat(^1^7)) = nat(^4^3^6^0^9^1^0^4^0^0^0)
div(fac(nat(^2^5)),fac(nat(^1^7))) = nat(^4^3^6^0^9^1^0^4^0^0^0)

fac(nat(^5^0)) = nat(^3^0^4^1^4^0^9^3^2^0^1^7^1^3^3^7^8^0^4^3^6^1^2^6^0^8^1^
```

```
6^6^0^6^4^7^6^8^8^4^4^3^7^7^6^4^1^5^6^8^9^6^0^5^1^2^0^0^0^0^0^0^0^0^0^0^0)

bin2nat(plus(nat2bin(nat(^3^2^5)))) = nat(^5^2^9^7^5)

bin2nat(minus(plus(nat2bin(nat(^3^2^5))),nat2bin(nat(^1^1^6)))) =
  nat(^4^6^1^8^9)

bin2nat(fac(nat2bin(nat(^5^0)))) = nat(^3^0^4^1^4^0^9^3^2^0^1^7^1^3^3^7^8^
0^4^3^6^1^2^6^0^8^1^6^6^0^6^4^7^6^8^8^4^4^3^7^7^6^4^1^5^6^8^9^6^0^5^1^2^0^
0^0^0^0^0^0^0^0^0^0^0)

bin2nat(division(fac(nat2bin(nat(^5))),bin(^1'^1'))) = nat(^2^0)

sin(fp(^1),nat(^5)) = fp(pos,^0,^8^4^1^4^7^1,int(pos,^0))
cos(fp(^1),nat(^5)) = fp(pos,^0,^5^4^0^3^0^3,int(pos,^0))
sin(fp(^2),nat(^5)) = fp(pos,^0,^9^0^9^3^0^0,int(pos,^0))
cos(fp(^2),nat(^5)) = fp(neg,^0,^4^1^6^1^5^5,int(pos,^0))

sqrt2(nat(^8)) = fp(pos,^0,^1^4^1^4^2^1,int(pos,^1))
```

11 The Character Library

11.1 Characters

The characters are specified as constants. Note that once the characters are imported, all character names are reserved and cannot be declared again as constants of another sort; constants can be declared only once in a PSF specification. To avoid name clashes with the digits and the bits, the characters 0...9 are placed between quotes. Some simple functions are specified; *ord* assigns a number to each character, *eq* makes it possible to decide equality of two characters, *suc* returns the next character, and *prd* returns the previous one. Finally, *char* returns the character that has the input ordinal number. Modulo arithmetic is implicitly used for all functions.

```
data module Characters
begin

  exports
  begin

    sorts
      CHARACTER

    functions
      ord  : CHARACTER             -> NATURAL
      eq   : CHARACTER # CHARACTER -> BOOLEAN
```

```
    suc  : CHARACTER              -> CHARACTER
    prd  : CHARACTER              -> CHARACTER
    char : NATURAL                -> CHARACTER

    '0'  : -> CHARACTER
    ...
    '9'  : -> CHARACTER

    a    : -> CHARACTER
    ...
    z    : -> CHARACTER

    A    : -> CHARACTER
    ...
    Z    : -> CHARACTER

end

imports
  Naturals

variables
  char, char1, char2 : -> CHARACTER
  nat                : -> NATURAL

equations
  [’] eq(char1,char2) = eq(ord(char1),ord(char2))

  [’] suc(char) = char(ord(char) + nat(^1)) when eq(char,Z) = false
  [’] suc(char) = '0'                       when eq(char,Z) = true

  [’] prd(char) = char(ord(char) - nat(^1)) when eq(char,'0') = false
  [’] prd(char) = Z                         when eq(char,'0') = true

  [’] ord('0') = nat(^0)
  ...
  [’] ord('9') = nat(^9)
  [’] ord(a) = nat(^1^0)
  ...
  [’] ord(z) = nat(^3^5)
  [’] ord(A) = nat(^3^6)
  ...
  [’] ord(Z) = nat(^6^1)

  [’] char(nat(^0)) = '0'
```

```
    ...
    [’] char(nat(^9)) = ’9’
    [’] char(nat(^1^0)) = a
    ...
    [’] char(nat(^3^5)) = z
    [’] char(nat(^3^6)) = A
    ...
    [’] char(nat(^6^1)) = Z

    [’] char(nat) = char(mod(nat,nat(^6^2))) when gt(nat,nat(^6^1)) = true

end Characters
```

11.2 Character-Strings

Two extra functions on character strings are specified; an equality function and a function *cstr-elem*, which decides whether or not a given character is in a string.

```
data module Character-Strings
begin

  exports
  begin

    functions
      cstr-elem   : CHARACTER # CHAR-STRING   -> BOOLEAN
      eq          : CHAR-STRING # CHAR-STRING -> BOOLEAN

  end

  imports
    Strings {String-Parameters bound by [ITEM -> CHARACTER] to Characters
             renamed by [STRING -> CHAR-STRING]}

  variables
    char, char1, char2 : -> CHARACTER
    cstr, cstr1, cstr2 : -> CHAR-STRING

  equations
    [’] eq(cstr^char1,^char2) = false
    [’] eq(^char1,cstr^char2) = false
    [’] eq(^char1,^char2) = eq(char1,char2)
    [’] eq(cstr1^char1,cstr2^char2) = and(eq(cstr1,cstr2),eq(char1,char2))

    [’] cstr-elem(char1,^char2) = eq(char1,char2)
    [’] cstr-elem(char1,cstr^char2) = true
```

```
        when eq(char1,char2) = true
  ['] cstr-elem(char1,cstr^char2) = cstr-elem(char1,cstr)
        when eq(char1,char2) = false

end Character-Strings
```

```
Examples:
suc(prd('0')) = '0'
prd(char(nat(^1^2^4))) = Z
eq(^h^e^n^k,^k^e^e^s) = false
cstr-elem(r,^m^a^r^i^e) = true
```

12 Queues

Queues are specified as a parameterised data type. The parameter Q-ELEMENT represents the type of the elements that is placed in a queue, and *queue-nil* is the default queue element. The function *enqueue* adds an element on top of a queue, *serve* returns the bottom element, and *dequeue* is the remainder of a queue after removal of the bottom element.

```
data module Queues
begin

  parameters Queue-Parameters
  begin

    sorts
      Q-ELEMENT

    functions
      queue-nil : -> Q-ELEMENT

  end Queue-Parameters

  exports
    begin

      sorts
        QUEUE

      functions
         _~_         : Q-ELEMENT # QUEUE -> QUEUE
        empty-queue :                   -> QUEUE
        enqueue     : Q-ELEMENT # QUEUE -> QUEUE
        serve       : QUEUE             -> Q-ELEMENT
```

```
      dequeue     : QUEUE              -> QUEUE
      length      : QUEUE              -> NATURAL

    end

  imports
    Naturals

  variables
    qe, qe1, qe2 : -> Q-ELEMENT
    q            : -> QUEUE

  equations
    ['] enqueue(qe,q) = qe~q

    ['] serve(empty-queue) = queue-nil
    ['] serve(qe~empty-queue) = qe
    ['] serve(qe1~(qe2~q)) = serve(qe2~q)

    ['] dequeue(empty-queue) = empty-queue
    ['] dequeue(qe~empty-queue) = empty-queue
    ['] dequeue(qe1~(qe2~q)) = qe1~dequeue(qe2~q)

    ['] length(empty-queue) = nat(^0)
    ['] length(qe~q) = length(q) + nat(^1)

end Queues
```

13 Tables

Tables is also a parameterised data type. What is placed in a table are entries, and the parameter KEY is an identification for the entries in the table. An entry contains next to a key also an item. Functions on tables are the following: *insert* places a new item in a table, given a key. If there already exists an entry with this key, then this entry is replaced by the new one, otherwise an entry is added. *retrieve* returns one item from a table, given a key. If there exists no entry with this key, the *table-nil* is returned. The function *delete* is the remainder of a table after an application of *retrieve*.

Note that this specification of tables does not require that the keys of the entries in a table are unique. For this reason, the effect of the functions on tables with not unique keys may not be satisfactory.

```
data module Tables
begin

  parameters
```

```
  Keys
  begin

    sorts
      KEY

    functions
      eq : KEY # KEY -> BOOLEAN

  end Keys,

  Items
  begin

    sorts
      ITEM

    functions
      table-nil : -> ITEM

  end Items

exports
  begin

    sorts
      TABLE, ENTRY

    functions
      _~_         : ENTRY # TABLE      -> TABLE
      empty-table :                    -> TABLE
      insert      : TABLE # ITEM # KEY -> TABLE
      retrieve    : TABLE # KEY        -> ITEM
      delete      : TABLE # KEY        -> TABLE
      in-table    : TABLE # KEY        -> BOOLEAN
      length      : TABLE              -> NATURAL
      entry       : KEY # ITEM         -> ENTRY

  end

imports
  Naturals

variables
  k, k1 ,k2 : -> KEY
```

```
    i, i1, i2 : -> ITEM
    t         : -> TABLE

  equations
    [’] insert(empty-table,i,k) = entry(k,i)˜empty-table
    [’] insert(entry(k1,i1)˜t,i2,k2) = entry(k1,i2)˜t
          when eq(k1,k2) = true
    [’] insert(entry(k1,i1)˜t,i2,k2) = entry(k1,i1)˜insert(t,i2,k2)
          when eq(k1,k2) = false

    [’] retrieve(empty-table,k) = table-nil
    [’] retrieve(entry(k1,i)˜t,k2) = i
          when eq(k1,k2) = true
    [’] retrieve(entry(k1,i)˜t,k2) = retrieve(t,k2)
          when eq(k1,k2) = false

    [’] delete(empty-table,k) = empty-table
    [’] delete(entry(k1,i)˜t,k2) = t
          when eq(k1,k2) = true
    [’] delete(entry(k1,i)˜t,k2) = entry(k1,i)˜delete(t,k2)
          when eq(k1,k2) = false

    [’] in-table(empty-table,k) = false
    [’] in-table(entry(k1,i)˜t,k2) = or(eq(k1,k2),in-table(t,k2))

    [’] length(empty-table) = nat(ˆ0)
    [’] length(entry(k,i)˜t) = length(t) + nat(ˆ1)

end Tables
```

14 A Manual for the PSF Library

Below, a manual-like overview of those definitions is given that are especially interesting when working with the library of PSF. Also the main line of the specification is given.

14.1 Numbers; Representations and Functions

All functions can be used in the naive way except the division functions, *div* and /, which yield a zero result in case of division by zero. So also the modulo functions *mod* have to be handled with care in case the second argument equals zero. First we define the set of numerical types as they are available, then the signature of the numerical types.

```
Numerical types

  NUM-TYPES = {BINARY, NATURAL, INTEGER, FP}


Booleans

  true, false :                   -> BOOLEAN
  not         : BOOLEAN           -> BOOLEAN
  and, or, eq : BOOLEAN # BOOLEAN -> BOOLEAN


Bits and bit strings

  0', 1'      :                  -> BIT
  ^_          : BIT              -> BIT-STRING
  _^_         : BIT-STRING # BIT -> BIT-STRING


Digits and digit strings

  0, 1, 2, 3, 4, 5, 6, 7, 8, 9 :      -> DIGIT
  ^_           : DIGIT                -> DIGIT-STRING
  _^_          : DIGIT-STRING # DIGIT -> DIGIT-STRING


Signs

  pos, neg : -> SIGN


Representation of numbers

  bin : BIT-STRING                                  -> BINARY
```

```
nat : DIGIT-STRING                                     -> NATURAL
int : SIGN # DIGIT-STRING                              -> INTEGER
int :        DIGIT-STRING                              -> INTEGER
int : SIGN # NATURAL                                   -> INTEGER
int :        NATURAL                                   -> INTEGER
fp  : SIGN # DIGIT-STRING # DIGIT-STRING # INTEGER -> FP
fp  : SIGN # DIGIT-STRING # DIGIT-STRING           -> FP
fp  : SIGN # DIGIT-STRING                # INTEGER -> FP
fp  : SIGN # DIGIT-STRING                          -> FP
fp  :        DIGIT-STRING # DIGIT-STRING # INTEGER -> FP
fp  :        DIGIT-STRING # DIGIT-STRING           -> FP
fp  :        DIGIT-STRING                # INTEGER -> FP
fp  :        DIGIT-STRING                          -> FP
fp  :        INTEGER                     # INTEGER -> FP
fp  :        INTEGER                               -> FP
```

Examples

```
bin(^1'^0'^1'^1') = 1011
nat(^3^5) = 35
int(neg,^9^0) = -90
fp(^3,^7) = 3,7
fp(^1^0,int(neg,^3)) = 10 E-3
fp(neg,^5^8,^0^3,int(pos,^2)) = -58,03 E+2
```

Functions

```
for TYPE in NUM-TYPES the following functions are defined:
_+_, _-_, _*_        : TYPE # TYPE -> TYPE

for TYPE in NUM-TYPES\{FP} we have:
div, mod, min, max   : TYPE # TYPE -> TYPE
eq, gt, gte, lt, lte : TYPE # TYPE -> BOOLEAN

remaining functions:
abs : INTEGER -> INTEGER
_/_ : FP      -> FP
```

Conversion Functions:

```
int2nat : INTEGER -> NATURAL
bin2nat : BINARY  -> NATURAL
nat2bin : NATURAL -> BINARY
```

The module *Floating-Points-M-P* contains the parameters *mantisse* and *precision*, and the parameter section is labelled with the name *Representation Lengths*. The first parameter stands for the length of the external representation, the second for the length of the internal representation, which is used in the calculations.

In *Floating-Points-3-4* these parameters are bound to the naturals $nat(\hat{}3)$ and $nat(\hat{}4)$ respectively. This happens by binding *Floating-Points-M-P* to the module *Natural-Names*, which contains the names of the natural numbers from 0 to 16.

The user can choose other instances of *mantisse* and *precision* by simply creating a module similar to *Floating-Points-3-4* with other parameter bindings.

14.2 Strings, Characters and Character Strings

The module *Strings* contains the sorts STRING and ITEM as parameters, and the parameter section is labelled with the name *String-Parameters*. This module forms the basis for all string types used in the specification. Together with the characters, the character strings and their functions, the functions defined on strings and string items are listed below. For more detailed information, the reader is referred to section 3.

```
Strings and functions on strings

  ^_      : ITEM            -> STRING
  _^_     : STRING # ITEM   -> STRING
  first   : STRING          -> ITEM
  last    : STRING          -> ITEM
  tail    : STRING          -> STRING
  reverse : STRING          -> STRING
  concat  : STRING # STRING -> STRING


Characters and character strings

  '0',...,'9','a',...,'z','A',...,'Z' : -> CHARACTER
  ^_  : CHARACTER                       -> CHARACTER-STRING
  _^_ : CHARACTER-STRING # CHARACTER    -> CHARACTER-STRING


Functions

  ord       : CHARACTER                 -> NATURAL
  eq        : CHARACTER # CHARACTER     -> BOOLEAN
  suc, prd  : CHARACTER                 -> CHARACTER
  char      : NATURAL                   -> CHARACTER
  cstr-elem : CHAR-STRING # CHARACTER   -> BOOLEAN
  eq        : CHAR-STRING # CHAR-STRING -> BOOLEAN
```

14.3 Queues and Tables

The PSF library also contains a specification of qeueues and tables, both are specified as parameterised data types. For more information, the reader is referred to the sections 12 and 13.

14.4 The Structure of the Specification

For an efficient use of the PSF library, the user must not use more modules than strictly necessary. Therefore we give the structure of the specification, of which only the relevant modules are mentioned.

```
Numerical-Library imports

  Booleans

  Signs                imports Booleans
  Binaries             imports Booleans, Strings
  Naturals             imports Booleans, Strings

  Conversion-Functions imports Binaries, Naturals

  Natural-Names        imports Naturals
  Integers             imports Naturals, Signs

  Floating-Points-M-P  imports Integers
  Floating-Points-3-4  imports Natural-Names, Floating-Points-M-P


Character-Library imports

  Characters           imports Naturals
  Character-Strings    imports Characters, Strings


Parameterised data types:

  Strings
  Queues               imports Naturals
  Tables               imports Naturals

PSF-Library            imports all
```

15 Conclusions

A library for PSF was specified in this paper. It contains the following numerical data types: Booleans, Binaries, Naturals, Integers and Floating-Points. Also a number of standard functions on these data types were specified. We return to the considerations made in the introduction.

1. The representation of the elements of the different numerical types was chosen uniform. The type of each number, except of a boolean, can be identified by a prefix constructor: *bin*, *nat*, *int* or *fp*. The *bin* constructor is parameterised with a string of bits, the other constructors are parameterised with one or more strings of digits. *int* and *fp* can also have a sign (*pos* or *neg*) as a parameter. The string constructor is ^. the functions +, -, * are infix operators as well as the function / on floating point numbers. All other functions, for example *div*, *mod* and *eq*, were specified as prefix operators.

2. All calculations with the types mentioned above are based on the well-known tables of operations on decimal numbers. This makes the specification significantly larger than, for instance, a specification based on *successor naturals* or on binary numbers. Comparison of the performance of our specification with other specifications with the PSF term rewriter showed that our specification is relatively fast.

3. It may be clear from the specification that only the very basic mathematical operations were specified. Other functions, which a user may need for some more specific goal, can be specified on a higher level.

4. The floating point numbers have two parameters, *mantisse* and *precision*, which the user can choose him/herself. Also a module *Floating-Points-3-4* was specified, which binds the parameters to fixed values. *mantisse*, which represents the length of the external representation of a floating point number, is bound to 3 and *precision*, which is the length of the internal representation, is bound to 4.

Finally the data types Characters, Character-Strings and some functions on these data types were specified, as well as the parameterised types Tables and Queues.

We are confident that the specification given in this paper can be directly transformed into a specification in the language μCRL by removing the modular structure and by just changing the keywords of PSF into keywords of μCRL. Only the flexibility as provided by the parameterisation mechanism of PSF will have to be sacrificed.

Bugs. Please contact the author in case of bugs or for suggestions for improvements or useful extensions of the PSF library.

References

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[GP90] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Report CS-R9076, CWI, Amsterdam, 1990.

[GP91] J.F. Groote and A. Ponse. Proof theory for μCRL. Report CS-R9138, CWI, Amsterdam, 1991.

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.