On the Definition of Modular E-LOTOS *

Draft of 1996/12

Mihaela Sighireanu Hubert Garavel

Abstract

In this paper we propose a solution for the module system for E-LOTOS. This solution allows export and import, hiding, and generic modules. It is an algebraic one, inspired by the LOTOSPHERE proposal. We give the basic concepts of this module system, a complete abstract syntax and static semantics, a base environment, and examples of use by writing ISO-8807 libraries and LOTOSPHERE libraries.

Contents

*This work has been supported in part by the European Commission, under project ISC-CAN-65 "EUCALYPTUS-2: A European/Canadian LOTOS Protocol Tool Set".

1 Introduction

Modular programming/specification can be done in any language, with sufficient discipline from the programmers. However, it is facilitated if the programming/specification language provides constructs to express some aspects of the modular structure and check them automatically. For example, in Modula-2 [Wir83] are provided means as implementations and interfaces, in Ada packages, in SML [MTH90] signatures, structures, and functors, classes in C++ ...

LOTOS has only a limited form of modules, which encapsulate data types and operations, but not processes. Moreover, this mechanism does not support abstraction: every object declared in a module is exported outside.

Since 1988, there have been several proposals for enhancing LOTOS modules. An overview of those proposals is given in [Que96a, Section 3].

One of the goals of E-LOTOS is to develop a modularization system, which should allow export and import, hiding, and generic modules. The modules used in the data part should be the same as those used in the behavioural part, so "process" declarations should be allowed as well as "type" and "operation" declarations. For abstraction and code re-use, signatures and generic modules are very useful.

The purpose of the present paper is to propose a solution for the module system of E-LOTOS with a base language as defined in [Que96b]. To attain it, we present a complete syntax, static semantics, and a base environment. A short introduction to this proposal is given by presenting how this proposal respond to issues formulated in the questionnaire of [Que96a, Section 7.2].

It is worth noticing that we do not propose a concrete syntax for module, so the keywords are chosen for convenience purposes.

This paper is structured as follows:

In Section 2 we present a short introduction to the module system proposed.

Section 3 gives the syntax of the module system. Section 4 presents the static semantics of the module system. In Section 5 we outline the base environment of a such modular system. Annexes A and B indicate how the LOTOS and LOTOSPEHERE libraries can be translated in the present proposal.

2 Basic concepts

2.1 Specification structuring

A specification in modular-E-LOTOS is given as a sequence of *signature* and *module* declarations, followed by a behaviour expression or a value expression. So we provides separate declarations of module signatures and the module implementation.

There are several possible signatures for a given module (implementation), and several modules (implementations) for a given signatures, via renaming morphisms.

The signature of a module describes its exports. For example, the Monoid signature is:

signature Monoid **is**

type M

constant 0 : M

function + (M, M) : M

```
    eqns forall x, y, z : M
      (0 + x) = x ;
      (x + 0) = x ;
      ((x + y) + z) = (x + (y + z)) ;
    endeqns
  endsig
```

Many tools will treat equational specifications just as (type checked) comments, so we should ensure that (as with Extended ML) the equations can be commented out without effecting the semantics of the module.

There are a number of modules with this signature. The simplest is the one-point domain:

```
  module OnePoint is
  export Monoid in
    type M is zero () endtype ;
    constant 0 : M := zero () ;
    function + (x: M, y: M) : M is
      case
        !x::zero () and !y::zero () -> 0
      endcase
    endfunc
  endmod
```

2.2 Contents of modules (and interfaces)

As show above, the signatures are sequences of *description* of types, processes, functions, constants, equations, relations, and properties. Similar, modules are sequences of *declarations* of types, processes, functions, constants, equations, relations, and properties. The main difference is that in signatures, only types could be specified with an implementation (type expression). Functions, processes, and constants has no implementations (values).

It is worth noticing that we have no nesting modules. The SML module system is unusual in that it allows modules to be nested in other modules. This extension makes the module system more complex (as described below) and it is not obvious whether the extra complexity is necessary.

Also, the requirements of the E-LOTOS work to develop a module system for process declarations is satisfied, and it is compatible with the module system used for data.

As an example we shown how data-flow processes can be specified. A data-flow module is one with the signature:

```
  signature Dataflow is
    type In
    type Out
    process Flow [ in : In, out : Out ] : noexit
  endsig
```

For example, a data-flow module for addition is:

module Add **is**
  Integer +
  type In **is** <x:Int, y:Int> **endtype**
  type Out **is** <z:Int> **endtype**
  process Flow [in : In, out : Out] : **noexit is**
    in <?x:Int, ?y:Int> ; out <!(x+y)> ; Flow [in, out]
  endproc
endsig

2.3 Abstraction, hiding

At level of signatures, abstraction is present by allowing **declaration** of abstract data types in signatures. This abstract data types may be implemented by a lot of concrete (or manifest) data types. For example,if we declare:

signature Set :=
  type Element
  type Set
  constant empty : Set
  function insert (x: Element, s: Set) : Set
  function delete (x: Element, s: Set) : Set
  function member (x: Element, s: Set) : Bool
endsig

several implementations for type Set may be given: using list, binary trees ...

An issue of abstract data types is type equality. Equality is an important concept in LOTOS, since it is used implicitly by synchronization. So far, all types allow equality, but modules can introduce data abstraction, so it is no longer possible to see the internal representation of a data type. Our proposal consider abstract data types as non-equality types. However, a built-in equality may be provided. The advantages of not making abstract data types equality types are:

1. data abstraction, and
2. the dynamic semantics may be simpler.

Abstraction may be done at the level of modules by three constructs: exporting of a given signature, the hiding of a signature, and local declaration of an implementation (module).

2.4 Composition of modules (and interfaces)

Signatures can include other signatures. For example, the signature for pre-orders extends that for partial orders with one equation:

signature PreOrder **is**

```
    Boolean +
    type T
    function <= (x: T, y: T) : Bool
    eqns forall x, y, z: T
      x <= x ;
      (x <= y andthen y <= z) => x <= y ;
    endeqns
  endsig
  signature PartialOrder is
    PreOrder +
    eqns forall x,y: T
      (x <= y andthen y <= x) => x = y ;
    endeqns
  endsig
```

Similar for modules: the coercion of the natural numbers type (assuming an appropriate 'Natural' module) to a Monoid structure is made by:

```
  module NatMonoid is
    export Monoid in
    Natural +
    type M is Nat endtype
  endmod
```

Another example is a simple signature for monomorphic lists is:

```
  signature List is
    include Monoid
    type E
    function inj (x: E) : M
  endsig
```

2.5 Object designation

We consider that names are flat, so names are global for a given specification. The clash of names is suitable for sharing (see Section 2.7). Names can be coerce to unique names by prefixing with a given character string.

2.6 Genericity

As well as specifying the exports of a module, signatures are also used to specify the imports of a generic module. For example, a generic lists module can be implemented as:

```
signature EqType is
  type E
endsig
module GenericList is
  generic EqType in
  export List in
  type M :=
    nil () | cons (x: E, l: M)
  endtype
  function + (s1: M, s2: M) : M  is
    case
      s1::nil () -> s2
    | s1::cons(x, xs) -> cons (x, + (xs, ys))
  endfunc
  function inj (x : E) : M is
    cons (x, nil())
  endfunc
endmod
```

This module can be used as follows (for renaming see next section):

```
module Lists is
  (rename
   ListInt for M
   injInt  for inj
   in
      GenericList actualizedby Integer) +
  (rename
   ListBool for M
   injBool  for inj
   in
      GenericList actualizedby Boolean) +
  ...
endmod
```

Generic modules such as these allow standard libraries of components to be built up, and supports code reuse of both components and 'glue'.

2.7 Renaming

Renaming is used for two purposes: to give an unique name to objects, and to allow instantiation with eventually name clashing ("sharing" in SML).

Sometimes in building complex specifications it is necessary to build module hierarchies which are graphs rather than trees. When doing this, generic modules often need to contain sharing information about their parameters. For example, a generic module for function composition can be defined:

```
signature Morphism is
  type Source
  type Target
  function f (x: Source) : Target
endsig
module Compose is
  generic rename FTarget for GSource
        in
        (prefix F in Morphism) +
        (prefix G in Morphism)
  in
  type Source is FSource endtype
  type Target is GTarget endtype
  function f (x:Source) : Target  is
    Gf (Ff (x))
  endfunc
endmod
```

Data flow modules can be combined using generic modules, for example two modules can be composed in sequence as:

```
module Sequence is
  generic rename AOut for BIn
        in
        (prefix A in Dataflow) +
        (prefix B in Dataflow)
  in
  type In is AIn endtype
  type Out is BOut endtype
  process Flow [ in : In, out : Out ] is
    hide mid : AOut in
      AFlow [in, mid] |[mid]| BFlow [mid, out]
  endproc
```

```
endmod
```

and in parallel as:

```
module Sync is
  generic rename AIn for BIn
                 AOut for BOut
         in
         (prefix A in Dataflow) +
         (prefix B in Dataflow)
  in
  type In is AIn endtype
  type Out is BOut endtype
  process Flow [ in : In, out : Out ] is
      AFlow [ in, out ] |[in,out]| BFlow [ in, out ]
  endproc
endmod
```

2.8 Relationship with the external environment

One of the decisions of the Paris meeting was to support external declarations, interfacing to other specification or implementation languages. In the module language described below, the syntax of external objects is based on those of user language: types, functions, processes, and modules may be external.

```
module ExtMonoid is export Monoid external endmod
```

Any object declared to be **external** has no formal dynamic semantics.

2.9 Equational specifications

Another decision taken at the Paris meeting was to allow equational specifications in signatures, for example in the 'Monoid' signature:

```
eqns forall x,y,z: Nat
  (x + 0) = x ;
  x = (x + 0) ;
  (x + (y + z)) = ((x + y) + z) ;
endeqns
```

Many tools will treat these specifications just as (type checked) comments, so we should ensure that (as with Extended ML) the equations can be commented out without effecting the semantics of the module.

We adopt here the syntax proposed in [GM96] to specify equations, relations and properties.

We have to provide a formal semantics for when equations are valid (although this is obviously not computable, so we cannot expect automatic tools for checking validity).

2.10 Compatibility with ACTONE

The functional part of E-LOTOS will include algebraic specifications in signatures.

For example, we can compare the LOTOS specification:

type Monoid **is**
  sort M
  opns 0 : -> M
    _+_ : M, M -> M
  eqns forall x,y,z : M **ofsort** M
    x + 0 = x;
    0 + x = x;
    (x + y) + z = x + (y + z)
endtype

with the declaration from the example data language:

signature Monoid **is**
  type M
  constant 0 : M
  function + (x: M, y: M) : M
  eqns forall x, y, z : M
    (x + 0) = x ;
    (0 + x) = x ;
    ((x + y) + z) = (x + (y + z)) ;
  endeqns
endsig
module Monoid **is**
    export Monoid **in external**
endmod

There is a strong resemblance between such specifications and ACTONEdata type declarations. There are, however, a number of differences, which need to be resolved:

1. module Monoid is specified to be *any* structure which satisfies the axioms, not just the initial one (in particular we may wish to introduce an **initial** declaration similar to the current **external**),
2. the relationship between generic modules and ACTONEparameterized types and type renaming is not obvious.

It seems that the module system provides a good starting place for supporting equationally specified data types in a strict functional language, but it requires careful investigation to see if it is suitable for use with LOTOS.

2.11 Semantics

The semantics is given in terms of type calculus. Signatures and modules are types, their types being a collection of type, process, functions, and constants bindings.

3 Syntax

3.1 Notations

In additions to the *terminals* defined for the user language [Que96b, Section 2.2], we will use the following syntax classes for Modules:

identifier domain	*meaning*	*abbreviations*
SpecId	specification identifiers	*spec-id*
SigId	signature identifiers	*sig-id*
ModId	module identifiers	*mod-id*
RenId	renaming identifiers	*ren-id*

The classes *non-terminal* for Modules are:

symbol domain	*meaning*	*abbreviations*
Specs	specifications	*spec*
SigSpec	signature specification	*sig-spec*
SigBlock	signature blocks	*sig-block*
SigExp	signature expression	*sig-exp*
ModSpec	module specification	*mod-spec*
ModBlock	module block	*mod-block*
ModExp	module expression	*mod-exp*
EqnsDesc	equations description	*eqns-desc*
Relsdesc	relations description	*rels-desc*
PropsDesc	properties description	*props-desc*

To express the abstract syntax of user language we used the following meta-symbols:

symbol	*meaning*
::=	defined to be
|	alternatively
*	the previous syntactic unit may be repeated zero or more times
+	the previous syntactic unit may be repeated one or more times
{ }	the enclosed syntactic unit may be repeated zero or more times
[ ]	the enclosed syntactic unit is optional—may occur zero or onetime

In the grammars, non-primitive constructs (which are defined by translation into terms of primitive constructs) are marked with a "⋆".

3.2 Specifications

spec ::= **specification** *spec-id* [**: exit** *type-exp*] **is** E-LOTOS *behaviour specification* (spec1)
{*sig-spec* | *mod-spec*}
behaviour *B*
endspec

| **specification** *spec-id* [**:** *S*] **is** E-LOTOS *value specification* (spec2)

{ *sig-spec* | *mod-spec*}
value E
endspec

3.3 Signature specification

sig-spec ::= **signature** *sig-id* **is** — *specification of signatures* (SS1)
sig-exp
endsig

3.4 Signature block

sig-block ::= **type** S [**is** *type-exp* **endtype**] — *abstract/concrete type signature* (SB1)

| **process** Π [[RG]] [(RT)] [**:noexit**] [**raises** [RX]] — *process signature* (SB2)

| **function** F [(RT)] [: S] [**raises** [RX]] — *function signature* (SB3)

| **const** K : S — *constant signature* (SB4)

| **eqns** *eqns-desc* **endeqns** — *equation signature* (SB5)

| **rels** *rels-desc* **endrels** — *relation signature* (SB6)

| **props** *props-desc* **endprops** — *property signature* (SB7)

| *sig-block*$_1$; *sig-block*$_2$ — *signature blocks composition* (SB8)

3.5 Signature expression

sig-exp ::= *sig-block* — *signature block* (SE1)

| *sig-id* — *signature identifier* (SE2)

| *sig-exp*$_1$ + *sig-exp*$_2$ — *composition with name clash* (SE3)

| *sig-exp*$_1$ **in** *sig-exp*$_2$ — *inclusion* (SE4)

| **hide** *sig-exp*$_1$ **in** *sig-exp*$_2$ — *hiding* (SE5)

| **rename** *renaming* (SE6)
  {**type** S **for** S'}
  {**process** Π **for** Π'}
  {**function** F **for** F'}
  {**constant** K **for** K'}
in *sig-exp*

3.6 Module specification

mod-spec ::= **module** *mod-id* **is** *specification of modules* (MS1)
mod-exp
endmod

3.7 Module blocks

mod-block ::= D *declarations of base language* (MB1)

| **const** K : S := E *constant declaration* (MB2)

| **eqns** *eqns-desc* **endeqns** *equation declaration* (MB3)

| **rels** *rels-desc* **endrels** *relation declaration* (MB4)

| **props** *props-desc* **endprops** *property declaration* (MB5)

| *mod-block*$_1$; *mod-block*$_2$ *module blocks composition* (MB6)

3.8 Module expressions

mod-exp ::= *mod-block* *module block* (ME1)

| *mod-id* *module inclusion* (ME2)

| **external** *external module* (ME3)

| **export** *sig-exp* **in** *mod-exp* *exporting* (ME4)

| **generic** *sig-exp* **in** *mod-exp* *generic declaration* (ME5)

| *mod-exp*$_1$ **+** *mod-exp*$_2$ | *composition with name clash* (ME6)

| **local** *mod-exp* **in** *mod-exp* | *local modules* (ME7)

| **hide** *sig-exp* **in** *mod-exp* | *hiding of signatures* (ME8)

| **rename** | *renaming* (ME9)
  {**type** S **for** S'}
  {**process** Π **for** Π'}
  {**function** F **for** F'}
  {**constant** K **for** K'}
 in *mod-exp*

| *mod-exp* **actualizedby** *mod-exp* | *instantiation* (ME10)

$\star$ | **prefix** $<$ *string* $>$ **in** *mod-exp* | *prefixing rename*(ME11)

3.9 Equational descriptions

eqns-desc ::= [**forall** *IRT*] [**ofsort** S] E_0 ; | *true testing* (EQ1)

| [**forall** *IRT*] [**ofsort** S] E_1 **=>** E_2 ; | *true testing with premises* (EQ2)

| [**forall** *IRT*] [**ofsort** S] E_0
  raises X [(RE)] ; | *evaluation with exception raising* (EQ3)

| *eqns-desc* *eqns-desc* | *sequencing* (EQ4)

3.10 Relational descriptions

rels-desc ::= B_1 **=** B_2 **mod** R ; | *equivalence modulo* R (REL1)

| B_1 **<** B_2 **mod** R ; | *included in modulo* R (REL2)

| B_1 **>** B_2 **mod** R ; | *includes modulo* R (REL3)

| *rels-desc* *rels-desc* | *sequencing* (REL4)

We may think of various relations R (some of which are already defined in an Annex of the existing

LOTOS standard):

- strong equivalence [Par81]
- observational equivalence [Mil80]
- branching bisimulation [vGW89]
- $\tau^*.a$ bisimulation [Mou92]
- safety equivalence [BFG+91]
- etc.

and/or the corresponding pre-orders.

3.11 Property descriptions

props-desc ::= *B* |= *F* ; *satisfied formula* (PP1)

| *props-desc* *props-desc* *sequencing* (PP2)

F is a formula of the propositional μ-calculus (from which we can derive the ACTL temporal logic as a collection of shorthand notations).

4 Semantics

4.1 Notations

The static semantics is given by a series of judgments, defined by a set of rules. We assume given the typing judgments $\mathcal{C} \vdash E \Rightarrow t$ and $\mathcal{C} \vdash B \Rightarrow \textbf{exit}\ t$ for the base language [Que96b, Section 2.2]. The judgments are of following form:

assertion	*meaning*
$\mathcal{C} \vdash \mathit{spec} \Rightarrow \texttt{ok}$	The specification *spec* is well formed.
$\mathcal{C} \vdash \mathit{sig\text{-}spec} \Rightarrow \mathcal{C}'$	The signature specification *sig-spec* is well formed and gives the context $\mathcal{C}'$.
$\mathcal{C} \vdash \mathit{sig\text{-}block} \Rightarrow \mathit{mty}$	The signature block *sig-block* is well formed and has the type *mty*.
$\mathcal{C} \vdash \mathit{sig\text{-}exp} \Rightarrow \mathit{mty}$	The signature expression *sig-exp* is well formed and has the type *mty*.
$\mathcal{C} \vdash \mathit{mod\text{-}spec} \Rightarrow \mathcal{C}'$	The specification of module *mod-spec* is well formed and gives the context $\mathcal{C}'$.
$\mathcal{C} \vdash \mathit{mod\text{-}block} \Rightarrow \mathit{mty}$	The module block *mod-block* is well formed and has the type *mty*.
$\mathcal{C} \vdash \mathit{mod\text{-}exp} \Rightarrow \mathit{mty}$	The module expression *mod-exp* is well formed and has the type *mty*.
$\mathcal{C} \vdash \mathit{eqns\text{-}desc} \Rightarrow \texttt{ok}$	The equation description *eqns-desc* is well formed.
$\mathcal{C} \vdash \mathit{rels\text{-}desc} \Rightarrow \texttt{ok}$	The relations description *rels-desc* is well formed.
$\mathcal{C} \vdash \mathit{proposdesc} \Rightarrow \texttt{ok}$	The properties description *props-desc* is well formed.
$\mathcal{C} \vdash \mathit{mty}_1 <: \mathit{mty}_2$	The module type mty_1 is a subtype of the module type mty_2

The context for the base language (see [Que96b, Section 2.2]) is enriched with the following bindings for identifiers of signatures and modules as follows:

$$
\begin{array}{rcll}
\mathcal{C} & ::= & \text{base language context} & \\
 & | & \mathit{sig\text{-}id} \mapsto \mathit{mty} & \textit{signatures} \\
 & | & \mathit{mod\text{-}id} \mapsto \mathit{mty} & \textit{modules} \\
 & & & \\
\mathit{mty} & ::= & \mathit{decl}^* & \textit{non-parameterized type module} \\
 & | & \mathit{decl}^* \to \mathit{decl}^* & \textit{parameterized type module} \\
 & & & \\
\mathit{decl} & ::= & K \mapsto S & \textit{constant specification} \\
 & | & S \mapsto \texttt{atype} & \textit{abstract type specification} \\
 & | & S \mapsto (\texttt{type} \mid S') & \textit{concrete (manifest) type specification} \\
 & | & C \mapsto \{\mathit{profile}_1, \ldots, \mathit{profile}_n\} & \textit{constructor specification} \\
 & | & F \mapsto \{\mathit{profile}_1, \ldots, \mathit{profile}_n\} & \textit{function} \\
 & | & \Pi \mapsto \mathit{profile} & \textit{process}
\end{array}
$$

The operations on contexts and module types are similar to those defined in [Que96b, Section 2.2]: "+" is the disjunct composition, "$\oplus$" is the modified by composition, "$\ominus$" is the domain restriction, and "$\odot$" is the match composition.

We give below the definitions of the operations we will used over contexts. When A and B are sets, $\mathrm{Fin}(A)$ denotes the set of finite subsets of A, and $A \overset{\text{fin}}{\to} B$ denotes the set of *finite maps* (partial functions with finite domain) from A to B. The domain and range of a finite map, f, are denoted $\mathrm{Dom}(f)$ and $\mathrm{Ran}(f)$.

A finite map will often be written explicitly in the form $\{a_1 \mapsto b_1, \cdots, a_k \mapsto b_k\}$, $k \geq 0$; in particular, the empty map is $\{\}$.

When f and g are finite maps we define:

- $f + g$ with domain $\mathrm{Dom}(f) \cup \mathrm{Dom}(g)$ and values:

$$(f+g)(a) = \begin{cases} f(a) & \textbf{if } a \notin \mathrm{Dom}(g) \\ g(a) & \textbf{if } a \notin \mathrm{Dom}(f) \\ error & \textbf{otherwise} \end{cases}$$

  This operator is called *disjunct* composition of f and g.

- $f \oplus g$ with domain $\mathrm{Dom}(f) \cup \mathrm{Dom}(g)$ and values:

$$(f \oplus g)(a) = \begin{cases} f(a) & \textbf{if } a \notin \mathrm{Dom}(g) \\ g(a) & \textbf{otherwise} \end{cases}$$

  This operator is called f *modified by* (or *overridden*) g.

- $f \odot g$ with domain $\mathrm{Dom}(f) \cup \mathrm{Dom}(g)$ and values:

$$(f \odot g)(a) = \begin{cases} f(a) & \textbf{if } a \notin \mathrm{Dom}(g) \\ g(a) & \textbf{if } a \notin \mathrm{Dom}(f) \\ f(a) & \textbf{if } g(a) = f(a) \\ error & \textbf{otherwise} \end{cases}$$

  This operator is therefore called *match* composition of f and g.

- $f \ominus \{a_1, ..., a_n\}$ where $\{a_1, ..., a_n\} \subset \mathrm{Dom}(f)$, is a map with domain $\mathrm{Dom}(f) \setminus \{a_1, ..., a_n\}$ and values:

  $(f \ominus \{a_1, ..., a_n\})(a) = f(a)$ **if** $a \notin \{a_1, ..., a_n\}$

  This operator is usually called restriction of domain of f.

When the range of a partial map is a finite set of subsets, $A \overset{\text{sfin}}{\to} \mathrm{Fin}(B)$ denotes *finite set-maps* from A to B. If f and g are set finite maps, the set finite map $f + g$, called *disjoint* composition, has domain $\mathrm{Dom}(f) \cup \mathrm{Dom}(g)$ and values:

$$(f \oplus g)(a) = \begin{cases} f(a) & \textbf{if } a \notin \mathrm{Dom}(g) \\ g(a) & \textbf{if } a \notin \mathrm{Dom}(f) \\ f(a) \cup g(a) & \textbf{if } g(a) \cap f(a) = \emptyset \\ error & \textbf{otherwise} \end{cases}$$

4.2 Specifications

The static semantics is given by assertions of form:

$\vdash$ *spec* $\Rightarrow$ `ok`

meaning "The specification *spec* is well formed (typed)."

The semantics must ensure that there are no circular declarations in signature and module specification.

E-LOTOS **behaviour specification**

specification *spec-id* [: **exit** *type-exp*] **is**
  {*sig-spec* | *mod-spec*}
behaviour *B*
endspec

$$\frac{(\exists \rho : \{1,...,n\} \mapsto \{1,...,n\}) \quad (\forall i \in \{1,...,n\}) \quad (+_{j=1}^{j=i-1} \mathcal{C}_j \vdash \textit{sig/mod-spec}_{\rho(i)} \Rightarrow \mathcal{C}_i) \quad +_{j=1}^{j=n} \mathcal{C}_j \vdash \textit{type-exp} \Rightarrow t \quad +_{j=1}^{j=n} \mathrm{Ran}(\mathcal{C}_j) \vdash B \Rightarrow \textbf{exit}\ t \quad \textit{spec-id} \notin \mathrm{Dom}(+_{j=1}^{j=n} \mathcal{C}_j)}{\vdash \left(\begin{array}{l} \textbf{specification}\ \textit{spec-id} : \textbf{exit}\ \textit{type-exp} \\ \quad \textit{sig/mod-spec}_1 \ldots \textit{sig/mod-spec}_n \\ \textbf{behaviour}\ B \\ \textbf{endspec} \end{array} \right) \Rightarrow \texttt{ok}}$$

E-LOTOS **value specification**

specification *spec-id* [: S] **is**
$\{$*sig-spec* $\mid$ *mod-spec*$\}$
value E
endspec

$$\frac{(\exists \rho : \{1,...,n\} \mapsto \{1,...,n\}) \quad (\forall i \in \{1,...,n\}) \quad (+_{j=1}^{j=i-1} \mathcal{C}_j \vdash \textit{sig/mod-spec}_{\rho(i)} \Rightarrow \mathcal{C}_i) \quad +_{j=1}^{j=n} \mathcal{C}_j \vdash S \Rightarrow t \quad +_{j=1}^{j=n} \mathrm{Ran}(\mathcal{C}_j) \vdash E \Rightarrow t \quad \textit{spec-id} \notin \mathrm{Dom}(+_{j=1}^{j=n} \mathcal{C}_j)}{\vdash \left(\begin{array}{l} \textbf{specification}\ \textit{spec-id} : S\ \textbf{is} \\ \quad \textit{sig/mod-spec}_1 \ldots \textit{sig/mod-spec}_n \\ \textbf{value}\ E \\ \textbf{endspec} \end{array} \right) \Rightarrow \texttt{ok}}$$

4.3 Signature specification

The static semantics is given by assertions of form:

$\mathcal{C} \vdash$ *sig-spec* $\Rightarrow \mathcal{C}'$

meaning "In the context $\mathcal{C}$, the specification of signature *sig-spec* is well formed and gives the context $\mathcal{C}'$."

The signatures have to be closed w.r.t. signatures included.

Signature specification

signature *sig-id* **is** *sig-exp* **endsig**

$$\frac{\mathcal{C} \vdash \textit{sig-exp} \Rightarrow \textit{mty}}{\mathcal{C} \vdash (\textbf{signature}\ \textit{sig-id}\ \textbf{is}\ \textit{sig-exp}\ \textbf{endsig}) \Rightarrow \textit{sig-id} \mapsto \textit{mty}}$$

4.4 Signature block

The static semantics is given by assertions of form:

$\mathcal{C} \vdash$ *sig-block* $\Rightarrow$ *mty*

meaning "In the context $\mathcal{C}$, the signature block *sig-block* is well formed and has the type *mty*."

The module type sub-typing is a preorder:

$$\frac{}{\mathcal{C} \vdash \textit{mty} <: \textit{mty}}$$

$$\frac{\mathcal{C} \vdash mty <: myt' \qquad \mathcal{C} \vdash mty' <: myt''}{\mathcal{C} \vdash mty <: myt''}$$

We will write $mty \equiv myt'$ for $mty <: myt'$ and $mty' <: myt$, and we will write

$\mathcal{C} \vdash mty_1 \sqcup mty_2 \Rightarrow mty \qquad \mathcal{C} \vdash mty_1 \sqcap myt_2 \Rightarrow mty$

whenever (up to $\equiv$) mty_1 and myt_2 have at least upper bound (respectively a greatest lower bound, also obtained by "$\oplus$" operation on contexts) mty. The upper bound for module types is "**anymty**", lower bound is "{}" (the empty type).

Abstract type signature | **type** S

$$\frac{}{\mathcal{C} \vdash (\textbf{type}\ S) \Rightarrow (S \mapsto \texttt{atype})}$$

Concrete type signature | **type** S **is** *type-exp* **endtype**

As in base language, see [Que96b, Section 3.3].

$\mathcal{C} \vdash (S \mapsto \texttt{type}) <: (S \mapsto \texttt{atype})$

Process signature | **process** Π [`[`RG`]`] [`(`RT`)`] [**:noexit**] [**raises** [RX]]

$$\frac{\mathcal{C} \vdash RG \Rightarrow rg \qquad \mathcal{C} \vdash RT \Rightarrow rt \qquad \mathcal{C} \vdash RX \Rightarrow rx \qquad rt' = def(\textbf{out}(rt))}{\mathcal{C} \vdash (\textbf{process}\ \Pi\ \texttt{[}RG\texttt{]}\ \texttt{(}RT\texttt{)}\ [\textbf{:noexit}]\ [\textbf{raises}\ \texttt{[}\ RX\ \texttt{]}]) \Rightarrow (\Pi \mapsto rg, rt, rx \rightarrow \textbf{exit}\ [(rt')] \rightarrow newid)}$$

$$\frac{\mathcal{C} \vdash rg <: rg' \qquad \mathcal{C} \vdash rt <: rt' \qquad \mathcal{C} \vdash rx <: rx'}{\mathcal{C} \vdash (\Pi \mapsto rg, rt, rx \rightarrow \textbf{exit}\ (rt'') \rightarrow newid) <: (\Pi \mapsto rg', rt', rx' \rightarrow \textbf{exit}\ [(rt''')] \rightarrow newid')}$$

Function signature | **function** F [`(`RT`)`] [**:** S] [**raises** [`[`RX`]`]]

$$\frac{\mathcal{C} \vdash RX \Rightarrow rx \qquad \mathcal{C} \vdash RT \Rightarrow rt \qquad [\mathcal{C} \vdash S \Rightarrow t] \qquad \mathcal{C}(F)((), rt, rx)(S') = \emptyset}{\mathcal{C} \vdash (\textbf{function}\ F\ \texttt{(}RT\texttt{)}\ [\textbf{:}S]\ \textbf{raises}\ \texttt{[}RX\texttt{]}) \Rightarrow F \mapsto (), rt, rx \rightarrow \textbf{exit}\ [S] \rightarrow newuid}$$

$$\frac{\mathcal{C} \vdash rt <: rt' \qquad \mathcal{C} \vdash rx <: rx' \qquad \mathcal{C} \vdash (S \mapsto t) <: (S' \mapsto t')}{\mathcal{C} \vdash (F \mapsto (), rt, rx \rightarrow \textbf{exit}\ S \rightarrow newid) <: (F \mapsto (), rt', rx' \rightarrow \textbf{exit}\ S' \rightarrow newid')}$$

Constant signature | **const** K **:** S

$$\frac{\mathcal{C} \vdash S \Rightarrow \texttt{type}}{\mathcal{C} \vdash (\textbf{const } K : S) \Rightarrow K \mapsto S}$$

$$\frac{\mathcal{C} \vdash (S \mapsto t) <: (S' \mapsto t')}{\mathcal{C} \vdash (K \mapsto S) <: (K \mapsto S')}$$

Equation signatures **eqns** *eqns-desc* **endeqns**

$$\frac{\mathcal{C} \vdash \textit{eqns-desc} \Rightarrow \texttt{ok}}{\mathcal{C} \vdash (\textbf{eqns } \textit{eqns-desc } \textbf{endeqns}) \Rightarrow \{\}}$$

Relation signatures **rels** *rels-desc* **endrels**

$$\frac{\mathcal{C} \vdash \textit{rels-desc} \Rightarrow \texttt{ok}}{\mathcal{C} \vdash (\textbf{rels } \textit{rels-desc } \textbf{endrels}) \Rightarrow \{\}}$$

Property signatures **props** *props-desc* **endprops**

$$\frac{\mathcal{C} \vdash \textit{props-desc} \Rightarrow \texttt{ok}}{\mathcal{C} \vdash (\textbf{props } \textit{props-desc } \textbf{endprops}) \Rightarrow \{\}}$$

Blocks composition $\textit{sig-block}_1$; $\textit{sig-block}_2$

$$\frac{\begin{array}{rcl}\mathcal{C} & \vdash & \textit{sig-block}_1 \Rightarrow \textit{mty}_1 \\ \mathcal{C} + \textit{mty}_1 & \vdash & \textit{sig-block}_2 \Rightarrow \textit{mty}_2\end{array}}{\mathcal{C} \vdash (\textit{sig-block}_1 \ ; \ \textit{sig-block}_2) \Rightarrow \textit{mty}_1 + \textit{mty}_2}$$

$$\frac{\mathcal{C} \vdash \textit{mty}_1 <: \textit{mty}'_1 \qquad \mathcal{C} \vdash \textit{mty}_2 <: \textit{mty}'_2}{\mathcal{C} \vdash \textit{mty}_1 + \textit{mty}_2 <: \textit{mty}'_1 + \textit{mty}'_2}$$

$$\frac{}{\mathcal{C} \vdash \textit{mty}_1 + \textit{mty}_2 \equiv \textit{mty}_2 + \textit{mty}_1}$$

$$\frac{}{\mathcal{C} \vdash (\textit{mty}_1 + \textit{mty}_2) + \textit{mty}_3 \equiv \textit{mty}_1 + (\textit{mty}_2 + \textit{mty}_3)}$$

$$\frac{}{\mathcal{C} \vdash \{\} + \textit{mty}_1 \equiv \textit{mty}_1}$$

4.5 Signature expressions

The static semantics is given by assertions of form:

$$\mathcal{C} \vdash \textit{sig-exp} \Rightarrow \textit{mty}$$

meaning "In the context $\mathcal{C}$, the signature expression *sig-exp* is well formed and has the type *mty*." Also,

$$\frac{\mathcal{C} \vdash \mathit{sig\text{-}exp} \Rightarrow \mathit{mty} \qquad \mathcal{C} \vdash \mathit{mty} <: \mathit{mty}'}{\mathcal{C} \vdash \mathit{sig\text{-}exp} \Rightarrow \mathit{mty}'}$$

Signature block $\boxed{\mathit{sig\text{-}block}}$

$$\frac{\mathcal{C} \vdash \mathit{sig\text{-}block} \Rightarrow \mathit{mty}}{\mathcal{C} \vdash (\mathit{sig\text{-}block}) \Rightarrow \mathit{mty}}$$

Signature identifier $\boxed{\mathit{sig\text{-}id}}$

$$\frac{\mathcal{C} \vdash \mathit{sig\text{-}id} \Rightarrow \mathit{mty}}{\mathcal{C} \vdash (\mathit{sig\text{-}id}) \Rightarrow \mathit{mty}}$$

Composition with name clash $\boxed{\mathit{sig\text{-}exp}_1 \texttt{ + } \mathit{sig\text{-}exp}_2}$

$$\frac{\mathcal{C} \vdash \mathit{sig\text{-}exp}_1 \Rightarrow \mathit{mty}_1 \qquad \mathcal{C} \vdash \mathit{sig\text{-}exp}_2 \Rightarrow \mathit{mty}_2}{\mathcal{C} \vdash (\mathit{sig\text{-}exp}_1 \texttt{ + } \mathit{sig\text{-}exp}_2) \Rightarrow \mathit{mty}_1 \sqcap \mathit{mty}_2}$$

Inclusion $\boxed{\mathit{sig\text{-}exp}_1 \textbf{ in } \mathit{sig\text{-}exp}_2}$

$$\frac{\mathcal{C} \vdash \mathit{sig\text{-}exp}_1 \Rightarrow \mathit{mty}_1 \qquad \mathcal{C} + \mathit{mty}_1 \vdash \mathit{sig\text{-}exp}_2 \Rightarrow \mathit{mty}_2}{\mathcal{C} \vdash (\mathit{sig\text{-}exp}_1 \textbf{ in } \mathit{sig\text{-}exp}_2) \Rightarrow \mathit{mty}_1 + \mathit{mty}_2}$$

Hiding $\boxed{\textbf{hide } \mathit{sig\text{-}exp}_1 \textbf{ in } \mathit{sig\text{-}exp}_2}$

$$\frac{\mathcal{C} \vdash \mathit{sig\text{-}exp}_1 \Rightarrow \mathit{mty}_1 \qquad \mathcal{C} \vdash \mathit{sig\text{-}exp}_2 \Rightarrow \mathit{mty}_2 \qquad \mathcal{C} \vdash \mathit{mty}_2 <: \mathit{mty}_1 \qquad \mathcal{C} \vdash \mathit{mty}_2 \ominus \mathrm{Dom}(\mathit{mty}_1) \Rightarrow \mathit{mty}}{\mathcal{C} \vdash (\textbf{hide } \mathit{sig\text{-}exp}_1 \textbf{ in } \mathit{sig\text{-}exp}_2) \Rightarrow \mathit{mty}}$$

Renaming

$\boxed{\begin{array}{l}\textbf{rename} \\ \quad \{\textbf{type } S \textbf{ for } S'\ \} \\ \quad \{\textbf{process } \Pi \textbf{ for } \Pi'\ \} \\ \quad \{\textbf{function } F \textbf{ for } F'\ \} \\ \quad \{\textbf{constant } K \textbf{ for } K'\ \} \\ \textbf{in } \mathit{sig\text{-}exp}\end{array}}$

$$\frac{\begin{array}{rcl} \mathcal{C} & \vdash & \mathit{sig\text{-}exp} \Rightarrow \mathit{mty} \\ \mathcal{C} & \vdash & (\mathit{types\text{-}ren} \Rightarrow \mathit{mty}) \Rightarrow \mathit{mty}_S, \sigma_S \\ \mathcal{C} + \mathit{mty}_S & \vdash & (\mathit{processes\text{-}ren} \Rightarrow \mathit{mty}, \sigma_S) \Rightarrow \mathit{mty}_\Pi \\ \mathcal{C} + \mathit{mty}_S & \vdash & (\mathit{functions\text{-}ren} \Rightarrow \mathit{mty}, \sigma_S) \Rightarrow \mathit{mty}_F \\ \mathcal{C} + \mathit{mty}_S & \vdash & (\mathit{constants\text{-}ren} \Rightarrow \mathit{mty}, \sigma_S) \Rightarrow \mathit{mty}_K \end{array}}{\mathcal{C} \vdash \left(\begin{array}{l} \textbf{rename} \\ \quad \mathit{types\text{-}ren} \\ \quad \mathit{processes\text{-}ren} \\ \quad \mathit{functions\text{-}ren} \\ \quad \mathit{constants\text{-}ren} \\ \textbf{in } \mathit{sig\text{-}exp} \end{array}\right) \Rightarrow \mathit{mty}_S \sqcap \mathit{mty}_\Pi \sqcap \mathit{mty}_F \sqcap \mathit{mty}_K}$$

Type renaming $\{$ **type** S **for** S' $\}$

$$\frac{\begin{array}{rcl} \mathit{mty} & \vdash & S'_1 \Rightarrow t'_1 \quad \cdots \quad \mathit{mty} \vdash S'_n \Rightarrow t'_n \\ \mathcal{C} & \vdash & S_1 \mapsto t_1 \quad \cdots \quad \mathcal{C} \vdash S_n \mapsto t_n \end{array}}{\mathcal{C} \vdash ((\textbf{type } S_1 \textbf{ for } S'_1 \ldots \textbf{type } S_n \textbf{ for } S'_n) \Rightarrow \mathit{mty}) \Rightarrow \sqcap_{i=1}^{i=n} S_i \mapsto t_i, [S_1 \mapsto S'_1, \ldots, S_n \mapsto S'_n]}$$

Process renaming $\{$ **process** Π **for** Π' $\}$

$$\frac{\begin{array}{rcl} \mathit{mty} & \vdash & \Pi'_1 \Rightarrow \mathit{profile}'_1 \quad \cdots \quad \mathit{mty} \vdash \Pi'_n \Rightarrow \mathit{profile}'_n \\ \mathcal{C} & \vdash & \Pi_1 \mapsto \mathit{profile}_1 \quad \cdots \quad \mathcal{C} \vdash \Pi_n \mapsto \mathit{profile}_n \\ & & \sigma(\mathit{profile}'_1) \equiv \mathit{profile}_1 \quad \cdots \quad \sigma(\mathit{profile}'_n) \equiv \mathit{profile}_n \end{array}}{\mathcal{C} \vdash ((\textbf{process } \Pi_1 \textbf{ for } \Pi'_1 \ldots \textbf{process } \Pi_n \textbf{ for } \Pi'_n) \Rightarrow \mathit{mty}, \sigma) \Rightarrow \sqcap_{i=1}^{i=n} \Pi_i \mapsto \mathit{profile}_i}$$

Function renaming $\{$ **function** F **for** F' $\}$

$$\frac{\begin{array}{rcl} \mathit{mty} & \vdash & F'_1 \Rightarrow \mathit{profile}'_1 \quad \cdots \quad \mathit{mty} \vdash F'_n \Rightarrow \mathit{profile}'_n \\ \mathcal{C} & \vdash & F_1 \mapsto \mathit{profile}_1 \quad \cdots \quad \mathcal{C} \vdash F_n \mapsto \mathit{profile}_n \\ & & \sigma(\mathit{profile}'_1) \equiv \mathit{profile}_1 \quad \cdots \quad \sigma(\mathit{profile}'_n) \equiv \mathit{profile}_n \end{array}}{\mathcal{C} \vdash ((\textbf{function } F_1 \textbf{ for } F'_1 \ldots \textbf{function} F_n \textbf{ for } F'_n) \Rightarrow \mathit{mty}, \sigma) \Rightarrow \sqcup_{i=1}^{i=n} F_i \mapsto \mathit{profile}_i}$$

Constants renaming $\{$ **constant** K **for** K' $\}$

$$\frac{\begin{array}{rcl} \mathit{mty} & \vdash & K'_1 \Rightarrow S'_1 \quad \cdots \quad \mathit{mty} \vdash K'_n \Rightarrow S'_n \\ \mathcal{C} & \vdash & K_1 \mapsto S_1 \quad \cdots \quad \mathcal{C} \vdash K_n \mapsto S_n \\ & & \sigma(S'_1) = S_1 \quad \cdots \quad \sigma(S'_n) = S_n \end{array}}{\mathcal{C} \vdash ((\textbf{constant } K_1 \textbf{ for } K'_1 \ \ldots \textbf{constant } K_n \textbf{ for } K'_n) \Rightarrow \mathit{mty}, \sigma) \Rightarrow \sqcap_{i=1}^{i=n} K_i \mapsto S_i}$$

4.6 Module specification

The static semantics is given by assertions of form:

$$\mathcal{C} \vdash \mathit{mod\text{-}spec} \Rightarrow \mathcal{C}'$$

meaning "In the context $\mathcal{C}$, the specification of module *mod-spec* is well formed and gives the context $\mathcal{C}'$."

Module specification | **module** *mod-id* **is** *mod-exp* **endmod**

$$\frac{\mathcal{C} \vdash \textit{mod-exp} \Rightarrow \textit{mty}}{\mathcal{C} \vdash (\textbf{module}\ \textit{mod-id}\ \textbf{is}\ \textit{mod-exp}\ \textbf{endmod}) \Rightarrow \textit{mod-id} \mapsto \textit{mty}}$$

4.7 Module blocks

The static semantics is given by assertions of form:

$\mathcal{C} \vdash \textit{mod-block} \Rightarrow \textit{mty}$

meaning "In the context $\mathcal{C}$, the module block *mod-block* is well formed and has the type *mty*."

Also,

$$\frac{\mathcal{C} \vdash \textit{mod-block} \Rightarrow \textit{mty} \quad \mathcal{C} \vdash \textit{mty} <: \textit{mty}'}{\mathcal{C} \vdash \textit{mod-block} \Rightarrow \textit{mty}'}$$

Declarations of base language | *D*

Given in [Que96b, Section 3].

Constant declaration | **const** K $: S$ `:=` E

$$\frac{\mathcal{C} \vdash S \Rightarrow \texttt{type} \quad \mathcal{C} \vdash E \Rightarrow S}{\mathcal{C} \vdash (\textbf{const}\ K\ : S\ \texttt{:=}\ E) \Rightarrow K \mapsto S}$$

Equation declaration | **eqns** *eqns-desc* **endeqns**

$$\frac{\mathcal{C} \vdash \textit{eqns-desc} \Rightarrow \texttt{ok}}{\mathcal{C} \vdash (\textbf{eqns}\ \textit{eqns-desc}\ \textbf{endeqns}) \Rightarrow \{\}}$$

Relation declaration | **rels** *rels-desc* **endrels**

$$\frac{\mathcal{C} \vdash \textit{rels-desc} \Rightarrow \texttt{ok}}{\mathcal{C} \vdash (\textbf{rels}\ \textit{rels-desc}\ \textbf{endrels}) \Rightarrow \{\}}$$

Property declaration | **props** *props-desc* **endprops**

$$\frac{\mathcal{C} \vdash \textit{props-desc} \Rightarrow \texttt{ok}}{\mathcal{C} \vdash (\textbf{props}\ \textit{props-desc}\ \textbf{endprops}) \Rightarrow \{\}}$$

Blocks composition | $\textit{mod-block}_1$; $\textit{mod-block}_2$

$$\frac{\mathcal{C} \vdash \textit{mod-block}_1 \Rightarrow \textit{mty}_1 \quad \mathcal{C} + \textit{mty}_1 \vdash \textit{mod-block}_2 \Rightarrow \textit{mty}_2}{\mathcal{C} \vdash (\textit{mod-block}_1\ ;\ \textit{mod-block}_2) \Rightarrow \textit{mty}_1 + \textit{mty}_2}$$

4.8 Module expressions

The static semantics is given by assertions of form:

$$\mathcal{C} \vdash \textit{mod-exp} \Rightarrow \textit{mty}$$

meaning "In the context $\mathcal{C}$, the module expression *mod-exp* is well formed and has the type *mty*." Also,

$$\frac{\mathcal{C} \vdash \textit{mod-exp} \Rightarrow \textit{mty} \qquad \mathcal{C} \vdash \textit{mty} <: \textit{mty}'}{\mathcal{C} \vdash \textit{mod-exp} \Rightarrow \textit{mty}'}$$

Module block *mod-block*

$$\frac{\mathcal{C} \vdash \textit{mod-block} \Rightarrow \textit{mty}}{\mathcal{C} \vdash (\textit{mod-block}) \Rightarrow \textit{mty}}$$

Module identifier *mod-id*

$$\frac{\mathcal{C} \vdash \textit{mod-id} \Rightarrow \textit{mty}}{\mathcal{C} \vdash (\textit{mod-id}) \Rightarrow \textit{mty}}$$

External module **external**

The semantics must be given in the source language.

Exporting **export** *sig-exp* **in** *mod-exp*

$$\frac{\mathcal{C} \vdash \textit{sig-exp} \Rightarrow \textit{mty}_1 \qquad \mathcal{C} \vdash \textit{mod-exp} \Rightarrow \textit{mty}_2 \qquad \mathcal{C} \vdash \textit{mty}_2 \ominus \mathrm{Dom}(\textit{mty}_1) \Rightarrow \textit{mty}}{\mathcal{C} \vdash (\textbf{export}\ \textit{sig-exp}\ \textbf{in}\ \textit{mod-exp}) \Rightarrow \textit{mty}}$$

Generic declaration **generic** *sig-exp* **in** *mod-exp*

$$\frac{\mathcal{C} \vdash \textit{sig-exp} \Rightarrow \textit{mty}_1 \qquad \mathcal{C} + \textit{mty}_1 \vdash \textit{mod-exp} \Rightarrow \textit{mty}_2}{\mathcal{C} \vdash (\textbf{generic}\ \textit{sig-exp}\ \textbf{in}\ \textit{mod-exp}) \Rightarrow \textit{mty}_1 \to \textit{mty}_2}$$

$$\frac{\mathcal{C}\ \ ass \quad \textit{mty}_2 <: \textit{mty}_1 \qquad \mathcal{C} + \textit{mty}_2 \vdash \textit{mty}'_1 <: \textit{mty}'_2}{\mathcal{C} \vdash (\textit{mty}_1 \to \textit{mty}'_1) <: (\textit{mty}_2 \to \textit{mty}'_2)}$$

Composition with name clash $\textit{mod-exp}_1$ + $\textit{mod-exp}_2$

$$\frac{\mathcal{C} \vdash \textit{mod-exp}_1 \Rightarrow \textit{mty}_1 \qquad \mathcal{C} \vdash \textit{mod-exp}_2 \Rightarrow \textit{mty}_2}{\mathcal{C} \vdash (\textit{mod-exp}_1 + \textit{mod-exp}_2) \Rightarrow \textit{mty}_1 \sqcap \textit{mty}_2}$$

Local modules — **local** *mod-exp* **in** *mod-exp*

$$\frac{\begin{array}{rcl} \mathcal{C} & \vdash & \textit{mod-exp}_1 \Rightarrow \textit{mty}_1 \\ \mathcal{C} + \textit{mty}_1 & \vdash & \textit{mod-exp}_2 \Rightarrow \textit{mty}_2 \\ \mathcal{C} & \vdash & \textit{mty}_2 \ominus \mathrm{Dom}(\textit{mty}_1) \Rightarrow \textit{mty} \end{array}}{\mathcal{C} \vdash (\textbf{local } \textit{mod-exp} \textbf{ in } \textit{mod-exp}) \Rightarrow \textit{mty}}$$

Hiding — **hide** *sig-exp* **in** *mod-exp*

$$\frac{\begin{array}{rcl} \mathcal{C} & \vdash & \textit{mod-exp} \Rightarrow \textit{mty}_1 \\ \mathcal{C} & \vdash & \textit{sig-exp} \Rightarrow \textit{mty}_2 \\ \mathcal{C} & \vdash & \textit{mty}_2 \ominus \mathrm{Dom}(\textit{mty}_1) \Rightarrow \textit{mty} \end{array}}{\mathcal{C} \vdash (\textbf{hide } \textit{sig-exp} \textbf{ in } \textit{mod-exp}) \Rightarrow \textit{mty}}$$

Renaming

rename
{**type** S **for** S' }
{**process** Π **for** Π' }
{**function** F **for** F' }
{**constant** K **for** K' }
in *mod-exp*

$$\frac{\begin{array}{rcl} \mathcal{C} & \vdash & \textit{mod-exp} \Rightarrow \textit{mty} \\ \mathcal{C} & \vdash & (\textit{types-ren} \Rightarrow \textit{mty}) \Rightarrow \textit{mty}_S, \sigma_S \\ \mathcal{C} + \textit{mty}_S & \vdash & (\textit{processes-ren} \Rightarrow \textit{mty}, \sigma_S) \Rightarrow \textit{mty}_\Pi \\ \mathcal{C} + \textit{mty}_S & \vdash & (\textit{functions-ren} \Rightarrow \textit{mty}, \sigma_S) \Rightarrow \textit{mty}_F \\ \mathcal{C} + \textit{mty}_S & \vdash & (\textit{constants-ren} \Rightarrow \textit{mty}, \sigma_S) \Rightarrow \textit{mty}_K \end{array}}{\mathcal{C} \vdash \left(\begin{array}{l} \textbf{rename} \\ \quad \textit{types-ren} \\ \quad \textit{processes-ren} \\ \quad \textit{functions-ren} \\ \quad \textit{constants-ren} \\ \textbf{in } \textit{mod-exp} \end{array} \right) \Rightarrow \textit{mty}_S \oplus \textit{mty}_\Pi \oplus \textit{mty}_F \oplus \textit{mty}_K}$$

Renaming of sorts, process, functions, and types are the similar to those for signatures.

Instantiation — *mod-exp* **actualizedby** *mod-exp*

$$\frac{\begin{array}{rcl} \mathcal{C} & \vdash & \textit{mod-exp} \Rightarrow \textit{mty}_1 \to \textit{mty}_2 \\ \mathcal{C} & \vdash & \textit{mod-exp}' \Rightarrow \sigma(\textit{mty}_1) \\ & & \sigma \text{ injective} \end{array}}{\mathcal{C} \vdash (\textit{mod-exp} \textbf{ actualizedby } \textit{mod-exp}') \Rightarrow \sigma(\textit{mty}_2)}$$

Prefixing rename — **prefix** $<$ *string* $>$ **in** *mod-exp*

$$\frac{\begin{array}{rcl} \mathcal{C} & \vdash & \textit{mod-exp} \Rightarrow \textit{mty} \\ & & \sigma = \{\textit{id} \mapsto \textit{string-id}\} \end{array}}{\mathcal{C} \vdash (\textbf{prefix } < \textit{string} > \textbf{ in } \textit{mod-exp}) \Rightarrow \sigma(\textit{mty}_2)}$$

5 Base environment

The *base environment* is a collection of signatures (i.e., interfaces) and (possibly generic) modules which are predefined, and can be used in any E-LOTOS specification. They play the same role for E-LOTOS as the standard libraries do for LOTOS, and should be upwardly compatible with them.

For each module:

- We should give a signature and an implementation module.
- We should specify if the module is *pervasive* or not. A module is pervasive if it is available everywhere without explicit import reference. The identification of pervasive modules may be the subject of further discussions.
- In presence of genericity we should specify whether the module will be defined using genericity, using shorthant notations (see "rich-term syntax [Pec94]) or both.
- We should specify whether the types and functions contained in the module will be defined using the base language, or if they will be implemented externally (e.g., real or floating-point numbers).

In the following presentation, the built-in data types are not described in detail. In the given examples, the implementation part are often omitted and only signatures (interfaces) are provided.

5.1 Predefined and constructed types

A proposal for type *declaration* is made in [Que96b, Section 3]. This proposal can be enriched in presence of the module system by several predefined and constructed type declarations.

We propose the following *predefined types*:

The type "int" Values of type *int* are signed integers, in a range which is implementation defined (16- or 32-bits).

Operations available on *int* are for instance binary operations such as addition, subtraction, multiplication, (Euclidean) division and remainder, the unary opposite, all usual sorts of comparisons, and a number of conversion to other types.

The type "real" Values of type *real* are implementation defined approximations of real numbers in an implementation-defined range.

Operations on theses values are the usual arithmetic operations, comparisons, a number of conversions to an from *ints* and *strings*, and a number of predefined mathematical functions.

The type "bool" Values of type *bool* are usual truth values `true` and `false`.

Operations available on *bool* are the binary conjunction and disjunction, the unary negation, comparisons (`false` $<$ `true`), and conversion to other types. Binary operations may exists in strict and non-strict (short-circuit evaluation) version.

The type "char" Values of type *char* are characters of the ASCII alphabet. Operations available on these values may be comparisons and conversion into other types.

The type "string" Values of type *string* are dynamic-length character strings. Operations available on these values may be concatenation, getting length of a string, taking a substring of a longer string, taking all of a string except for a substring, inserting a string into another one, searching a given substring in a longer string ... Strings are ordered by the lexicographic ordering. Strings may also be converted to other types.

The type "void" The type *void* comprises an unique value named `null`. This type corresponds to the "**none**" type in the core base language. It is useful to use *void* as the result type of some functions to make them procedures, and also for non-exiting process.

The *constructed type* may be:

Enumerated types A possible syntax for enumerated types is:

```
<enumerated type>       = ( <identifier+> )
<identifier+>           = <identifier+> , <identifier>
<identifier+>           = <identifier>
```

For example:

```
type day-of-week is ( Monday, Tuesday, Wednesday, Thursday,
                      Friday, Saturday, Sunday)
endtype
```

The definition of an *enumerated type* TET is translated into a base-language type and the following predefined functions:

```
function int (x: ET) : int raise UNDEFINED;

function ET (i: int) : ET raise RANGE-ERROR, UNDEFINED;

function string (x: ET) : string raise UNDEFINED;

function ET (s: string) : ET raise CONVERSION-ERROR, UNDEFINED;

function pred (x: ET) : ET raise RANGE-ERROR, UNDEFINED;

function succ (x: ET) : ET raise RANGE-ERROR, UNDEFINED;

function < (x, y: ET) : bool raise UNDEFINED;

function > (x, y: ET) : bool raise UNDEFINED;

function <= (x, y: ET) : bool raise UNDEFINED;

function => (x, y: ET) : bool raise UNDEFINED;
```

These will appear in the interface part. The implementation of the function is either automatically translated in the target language (so function are considered as external), or translated in the base language in the implementation module.

Scalar and simple types Scalar types are *int*, *bool*, *char*, and user defined *enumerated types*, and those only. They can be used in *subrange* construction types, and in the *set* constructed types[.

Simples types are scalar types plus type *real*.

Subranges A *subrange* type is defined as a subtype of an scalar type, named *host type* of the subrange type, by specifying the smallest and the largest values. Both of those values must belong to the same host type, and the former must be less or equal to the latter.

A possible syntax is:

```
<subrange>      = range <constant expr> ..  <constant expr>
```

A subrange is not a new type in the sense of the type checking: values of a subrange type are considered as values of the host type. It is only an indication that some range test must be **dynamically** performed in some situations involving subrange types: assignment of a value to a subrange-types variable, passing a value as a subrange-typed parameter in function call, etc.

The operations in subrange types are those of the host type, and the same discussion is valid as for the enumerated types.

Record types A record types correspond to the Cartesian product of component types, except that component values are accessed by a name rather than by their position.

A possible syntax for record definition type is:

```
<type decl>     = type <type id> is record <fields> endtype
<fields>        = <fields> <field>
<fields>        = <field>
<firld>         = <identifier> : <type id> ;
```

The declaration of a `RT` record type with fields `F1: S1, ..., Fn: Sn` is translated into the following one:

```
<type decl>     = type RT is
                        RT (F1: S1, \ldots{}, Fn: Sn)
                  endtype
```

and operation a selection of a field, equality and inequality comparisons are defined.

Sets A *set* type value introduces a type which is the power-set of some other *scalar* type, which is named the *basic* type.

A possible syntax is:

```
<type decl>     = type <type id> is set of <scalar type>
<scalar type>   = <type id>
```

Values of such types are subsets of the set of all values of the basic types. Any set, whatever its size, can be represented, except when system-dependent limitation arise. The implementation may use some techniques to be space and time efficient, but they cannot apply a general technique.

Operations available on set must be construction, binary operations as union, intersection and difference, comparisons, and membership test.

Lists Values of type *list* are ordered, linear lists, the element of which belong to the same type, named *element* type. There is no restriction on this type.

A possible syntax is:

```
<type decl>     = type <type id> is list of <type id>
```

Lists may be recursive types. Operations available are construction, efficient non-destructive concatenation, length, test for emptiness, and *head* and *tail*.

For each list type `LT` with element type `S`, there exists a translation in the base language as follows:

type LT **is**
    LT ()
  | LT (x: S, l: LT)
endtype
function LT (x: S) : LT;
function LT (l: LT, x: S) : LT;

A special function of construction of list may have the syntax:;

```
<expression>       = <list type name> [ <expression +> ]
```

for example `LT [V1, V2, ..., Vn]` which is equivalent to `LT (V1, LT (V2, ..., LT (Vn, LT ())))`.

A Definition of LOTOS standard data types in E-LOTOS

This Annex shows how the standard library of LOTOS [ISO88] can be expressed using the proposed User Language for E-LOTOS. We follow the following principles for this translation:

- Whenever possible, we keep the existing names used for types, functions and sorts
- The existing algebraic equations are kept and grouped, for each function definition, in an section introduced by the "**eqns**" keyword. However, equations which are not rewrite rules, and equations between constructors have been replaced with rewrite rules.
- As the modules language for E-LOTOS is not fixed yet, we have used the only proposal available yet: the so-called "LOTOSPHERE proposal", which is defined in the output document of the Yokohama SC21 meeting, July 1993. In particular, we introduced a separation between interfaces (signatures) and modules. The following table summarizes the keyword translation rules between LOTOS types and LOTOSPHERE modules.

LOTOS keywords	E-LOTOS keywords
type	**module** and **signature**
endtype	**endmod** and **endsig**
sorts	**type** ... **endtype**
opns	**function** ... **endfunc**
sortnames	**types**
opnnames	**fnnames**
renamedby	{...}
actualizedby	[...]

A.1 Boolean

```
signature Boolean is
   type Bool is
      true
   | false
   endtype
   function not (x: Bool) : Bool
   function _and_ (x: Bool, y: Bool) : Bool
   function _or_ (x: Bool, y: Bool) : Bool
   function _xor_ (x: Bool, y: Bool) : Bool
   function _implies_ (x: Bool, y: Bool) : Bool
   function _iff_ (x: Bool, y: Bool) : Bool
   function _eq_ (x: Bool, y: Bool) : Bool reqs equality (eq)
   function _ne_ (x: Bool, y: Bool) : Bool
endsig Boolean
module Boolean is
   type Bool is
      true
   | false
   endtype
   function not (x: Bool) : Bool is
      case x in
         true -> false
      | false -> true
      endcase
   eqns
        not(true) = false;
        not(false) = true;
```

```
endfunc
function _and_ (x: Bool, y: Bool) : Bool is
   case y in
      true -> x
   |  false -> false
   endcase
eqns forall x: Bool
     x and true = x;
     x and false = false;
endfunc
function _or_  (x: Bool, y: Bool) : Bool is
   case y in
      true -> true
   |  false -> x
   endcase
eqns forall x: Bool
     x or true = true;
     x or false = x;
endfunc
function _xor_ (x: Bool, y: Bool) : Bool is x and not(y) or (y and not(x))
eqns forall x, y: Bool
     x xor y = x and not(y) or (y and not(x));
endfunc
function _implies_ (x: Bool, y: Bool) : Bool is y or not(x)
eqns forall x, y: Bool
     x implies y = y or not(x);
endfunc
function _iff_ (x: Bool, y: Bool) : Bool is x implies y and (y implies x)
eqns forall x, y: Bool
     x iff y = x implies y and (y implies x);
endfunc
function _eq_  (x: Bool, y: Bool) : Bool reqs equality (eq)
eqns forall x, y: Bool
     x eq y = x iff y;
function _ne_ (x: Bool, y: Bool) : Bool is x xor y
eqns forall x: Bool
     x ne y = x xor y;
```

endfunc

endmod Boolean

A.2 FBoolean

signature FBoolean **is**

type FBool

function true : FBool

function not (x: FBool) : FBool

eqns forall x: FBool

not(not(x)) = x;

endsig FBoolean

A.3 Element

signature Element **is**

FBoolean +

type Element

function _eq_ (x: Element, y: Element) : FBool **reqs equality** (eq)

function _ne_ (x: Element, y: Element) : FBool

eqns forall x, y: Element

x ne y = not(x eq y);

endsig Element

A.4 Set

signature Set **is**

formal Element

import Boolean + NaturalNumber

type Set

function _eq_ (s: Set, t: Set) : Bool **reqs equality** (eq)

function _ne_ (s: Set, t: Set) : Bool

function {} : Set

function Insert (x: Element, s: Set) : Set

function Remove (x: Element, s: Set) : Set

function _IsIn_ (x: Element, s: Set) : Bool

function _NotIn_ (x: Element, s: Set) : Bool

function _Union_ (s: Set, t: Set) : Set

function _Ints_ (s: Set, t: Set) : Set

```
    function _Minus_ (s: Set, t: Set) : Set
    function _Includes_ (s: Set, t: Set) : Bool
    function _IsSubsetOf_ (s: Set, t: Set) : Bool
    function Card (s: Set) : Nat
endsig Set
module Set is
    formal Element
    import Boolean + NaturalNumber
    type Set is
        {}
    |   Add (e: Element, s: Set)
    endtype
    function _eq_ (s: Set, t: Set) : Bool
    eqns  s eq t = s Includes t and (t Includes s);
    endfunc
    function _ne_  (s: Set, t: Set) : Bool is not(s eq t)
    eqns  forall s, t: Set
        s ne t = not(s eq t)
    endfunc
    function Insert (x: Element, s: Set) : Set is
        case s in
          {} -> Add (x, {})
        | Add (e:= y: Element, s:= t: Set) when x eq y -> Add (x, t)
        | Add (e:= y: Element, s:= t: Set) when x ne y -> Add (y, Insert (x, t))
        endcase
    eqns forall x, y: Element, s: Set
        Insert (x, {}) = Add (x, {});
        x eq y => Insert (x, Add (y, s)) = Add (x, s);
        x ne y => Insert(x, Add (y, s)) = Add (y, Insert (x, s));
    endfunc
    function Remove (x: Element, s: Set) : Set is
        case s in
          {} -> {}
        | Add (e:= y: Element, s:= t: Set) when x eq y -> Remove (x, t)
        | Add (e:= y: Element, s:= t: Set) when x ne y -> Add (y, Remove (x, t))
        endcase
    eqns forall x, y: Element, s: Set
```

```
    Remove (x, {}) = {};
    x eq y => Remove (x, Add (y, s)) = Remove (x, s);
    x ne y => Remove (x, Add (y, s)) = Add (y, Remove (x, s));
endfunc
function _IsIn_ (x: Element, s: Set) : Bool is
    case s in
      {} -> false
    | Add (e:= y: Element, s:= t: Set) when x eq y -> true
    | Add (e:= y: Element, s:= t: Set) when x ne y -> x IsIn t
    endcase
eqns forall x, y: Element, s: Set
    x IsIn {} = false;
    x eq y => x IsIn Add (y, s) = true;
    x ne y => x IsIn Add (y, s) = x IsIn s;
endfunc
function _NotIn_ (e: Element, s: Set) : Bool is not(x IsIn s)
eqns forall x: Element, s: Set
    x NotIn s = not (x IsIn s);
endfunc
function _Union_ (s: Set, t: Set) : Set is
    case s in
      {} -> t
    | Add (e:= x: Element, s:= s1: Set) -> Add (x, s1 Union t)
    endcase
eqns forall x: Element, s, t: Set
    {} Union s = s;
    Add (x, s) Union t = Add (x, s Union t);
endfunc
function _Ints_ (s: Set, t: Set) : Set is
    case s in
      {} -> {}
    | Add (e:= x: Element, s:= s1: Set) when x IsIn t -> Add (x, s1 Ints t)
    | Add (e:= y: Element, s:= s1: Set) when x NotIn t -> s1 Ints t
    endcase
eqns forall x: Element, s, t: Set
    {} Ints s = {};
    x IsIn t => Add (x, s) Ints t = Add (x, s Ints t);
```

```
    x NotIn t => Add (x, s) Ints t = s Ints t;
  endfunc
  function _Minus_ (s: Set, t: Set) : Set is
    case t in
      {} -> s
    | Add (e:= x: Element, s:= t1: Set) -> Remove (x, s) Minus t1
    endcase
  eqns forall x: Element, s, t: Set
    s Minus {} = s;
    s Minus Add (x, t) = Remove (x, s) Minus t
  endfunc
  function _Includes_ (s: Set, t: Set) : Bool is
    case t in
      {} -> true
    | Add (e:= x: Element, s:= t1: Set) -> x IsIn s and (s Includes t1)
    endcase
  eqns forall x: Element, s, t: Set
    s Includes {} = true;
    s Includes Add (x, t) = x IsIn s and (s Includes t);
  endfunc
  function _IsSubsetOf_ (s: Set, t: Set) : Bool is t Includes s
  eqns forall s, t: Set
    s IsSubsetOf t = t Includes s;
  endfunc
  function Card (s: Set) : Nat is
    case s in
      {} -> 0
    | Add (s:= s1: Set, ...) -> Succ (Card (s1))
    endcase
  eqns forall x: Element, s: Set
    Card ({}) = 0;
    Card (Add (x, s)) = Succ (Card (s))
  endfunc
endmod Set
```

A.5 BasicNonEmptyString

```
signature BasicNonEmptyString is
   formal Element
   type NonEmptyString is
      String (e: Element)
   |  _+_ (e: Element, nes: NonEmptyString)
   endtype
endsig BasicNonEmptyString
module BasicNonEmptyString is
   formal Element
   type NonEmptyString is
      String (e: Element)
   |  _+_ (e: Element, nes: NonEmptyString)
   endtype
endmod BasicNonEmptyString
```

A.6 RicherNonEmptyString

```
signature RicherNonEmptyString is
   import BasicNonEmptyString + NaturalNumber
   function _+_ (s: NonEmptyString, x: Element) : NonEmptyString
   function _++_ (s: NonEmptyString, t: NonEmptyString) : NonEmptyString
   function Reverse (s: NonEmptyString) : NonEmptyString
   function Length (s: NonEmptyString) : Nat
endsig RicherNonEmptyString
module RicherNonEmptyString is
   import BasicNonEmptyString + NaturalNumber
   function _+_ (s: NonEmptyString, x: Element) : NonEmptyString is
      case s in
        String(e:= y: Element) -> y + String(x)
      | (e:= y: Element) + (nes:= t: NonEmptyString) -> y + (t + x)
      endcase
   eqns forall x, y: Element, s: NonEmptyString
      String(x) + y = x + String(y);
        x + s + y = x + (s + y);
   endfunc
   function _++_ (s: NonEmptyString, t: NonEmptyString) : NonEmptyString is
```

```
    case s in
      String(e:= x: Element) -> x + t
    | (e:= x: Element) + (nes:= s1: NonEmptyString) -> x + (s1 ++ t)
    endcase
  eqns forall x: Element, s, t: NonEmptyString
    String(x) ++ s = x + s;
      x + s ++ t = x + (s ++ t);
  endfunc
  function Reverse (s: NonEmptyString) : NonEmptyString is
    case s in
      String(e:= x: Element) -> String (x)
    | (e:= x: Element) + (nes:= s1: NonEmptyString) -> Reverse (s1) + x
    endcase
  eqns forall x: Element, s: NonEmptyString
    Reverse(String(x)) = String(x);
      Reverse(x + s) = Reverse(s) + x;
  endfunc
  function Length (s: NonEmptyString) : Nat is
    case s in
      String(e:= x: Element) -> Succ (0)
    | (e:= x: Element) + (nes:= s1: NonEmptyString) -> Succ (Length (s1))
    endcase
  eqns  forall x: Element, s: NonEmptyString
      Length(String(x)) = Succ(0);
      Length(x + s) = Succ(Length(s))
  endfunc
endmod RicherNonEmptyString
```

A.7 NonEmptyString

```
signature NonEmptyString is
  import RicherNonEmptyString + Boolean
  function _eq_ (s: NonEmptyString, t: NonEmptyString) : Bool reqs equality (eq)
  function _ne_ (s: NonEmptyString, t: NonEmptyString) : Bool
endsig NonEmptyString
module NonEmptyString is
  import RicherNonEmptyString + Boolean
```

```
  function _eq_ (s: NonEmptyString, t: NonEmptyString) : Bool is
    case
      s :: String (e:=x: Element) and t :: String (e:=y: Element)
          when x eq y -> true
    | s :: String (e:=x: Element) and t :: String (e:=y: Element)
          when x ne y -> false
    | (s :: String (...) and t :: ... + ...) or
      (s :: ... + ... and t :: String (...)) -> false
    | s :: (e:= x: Element) + (nes:= s1: NonEmptyString) and
      t :: (e:= y: Element) + (nes:= t1: NonEmptyString)
          when x eq y -> s1 eq t1
    | s :: (e:= x: Element) + (nes:= s1: NonEmptyString) and
      t :: (e:= y: Element) + (nes:= t1: NonEmptyString)
          when x ne y -> false
    endcase
  eqns forall s, t : NonEmptyString, x, y : Element
      x eq y => String(x) eq String(y) = true;
      x ne y => String(x) eq String(y) = false;
      String(x) eq (y + s) = false;
      x + s eq String(y) = false;
      x eq y => x + s eq (y + t) = s eq t;
      x ne y => x + s eq (y + t) = false;
  endfunc
  function _ne_ (s: NonEmptyString, t: NonEmptyString) : Bool is not(s eq t)
  eqns forall s, t : NonEmptyString
      s ne t = not(s eq t)
  endfunc
endmod NonEmptyString
```

A.8 String

```
signature String
  formal Element
  import NaturalNumber + Boolean
  type String is
    <>
  |  _+_ (e: Element, s: String)
```

```
    endtype
    function String (x: Element) : String
    function _+_ (s: String, x: Element) : String
    function _++_ (s: String, t: String) : String
    function Reverse (s: String) : String
    function Length (s: String) : Nat
    function _eq_ (s: String, t: String) : Bool reqs equality (eq)
    function _ne_ (s: NonEmptyString, t: NonEmptyString) : Bool
endsig String
module String
    formal Element
    import NaturalNumber + Boolean
    type String is
        <>
    |   _+_ (e: Element, s: String)
    endtype
    function String (x: Element) : String is x + <>
    eqns forall x: Element
        String (x) = x + <>
    endfunc
    function _+_ (s: String, x: Element) : String is
        case s in
          <> -> x + <>
        | (e:= y: Element) + (s:= s1: String) -> y + (s1 + x)
        endcase
    eqns forall x, y: Element, s: String
        <> + y = x + <>;
          x + s + y = x + (s + y);
    endfunc
    function _++_ (s: String, t: String) : String is
        case s in
          <> -> t
        | (e:= x: Element) + (s:= s1: String) -> x + (s1 ++ t)
        endcase
    eqns forall x: Element, s, t: String
        <> ++ s = s;
          x + s ++ t = x + (s ++ t);
```

```
endfunc
function Reverse (s: String) : String is
   case s in
     <> -> <>
   | (e:= x: Element) + (s:= s1: String) -> Reverse (s1) + x
   endcase
eqns forall x: Element, s: String
   Reverse(<>) = <>;
     Reverse(x + s) = Reverse(s) + x;
endfunc
function Length (s: String) : Nat is
   case s in
     <> -> 0
   | (e:= x: Element) + (s:= s1: String) -> Succ (Length (s1))
   endcase
eqns forall x: Element, s: String
     Length(<>) = 0;
     Length(x + s) = Succ(Length(s))
endfunc
function _eq_ (s: String, t: String) : Bool is
   case
     s :: <> and t :: <> -> true
   | (s :: <> and t :: ... + ...) or
     (s :: ... + ... and t :: <>) -> false
   | s :: (e:= x: Element) + (s:= s1: String) and
     t :: (e:= y: Element) + (s:= t1: String) when x eq y -> s1 eq t1
   | s :: (e:= x: Element) + (s:= s1: String) and
     t :: (e:= y: Element) + (s:= t1: String) when x ne y -> false
   endcase
eqns forall s, t : String, x, y : Element
     <> eq <> = true;
     <> eq (x + s) = false;
     x + s eq <> = false;
     x eq y => x + s eq (y + t) = s eq t;
     x ne y => x + s eq (y + t) = false;
endfunc
function _ne_ (s: String, t: String) : Bool is not(s eq t)
```

```
    eqns forall s, t : String
        s ne t = not(s eq t)
    endfunc
endmod String
```

A.9 BasicNaturalNumber

```
signature BasicNaturalNumber is
    type Nat is
      0
    | Succ (N: Nat)
    endtype
    function _+_ (m: Nat, n: Nat) : Nat
    function _*_ (m: Nat, n: Nat) : Nat
    function _**_ (m: Nat, n: Nat) : Nat
endsig BasicNaturalNumber
module BasicNaturalNumber is
    type Nat is
      0
    | Succ (N: Nat)
    endtype
    function _+_ (m: Nat, n: Nat) : Nat is
      case n in
        0 -> m
      | Succ (N:=n1 : Nat) -> Succ (m) + n
      endcase
    eqns forall m, n : Nat
        m + 0 = m;
        m + Succ(n) = Succ(m) + n;
    endfunc
    function _*_ (m: Nat, n: Nat) : Nat is
      case n in
        0 -> 0
      | Succ (N:=n1 : Nat) -> m + (m * n1)
      endcase
    eqns forall m, n : Nat
        m * 0 = 0;
```

```
        m * Succ(n) = m + (m * n);
    endfunc
    function _**_ (m: Nat, n: Nat) : Nat is
        case n in
          0 -> 0
        | Succ (N:=n1 : Nat) -> m * (m ** n1)
        endcase
    eqns forall  m, n : Nat
          m ** 0 = Succ(0);
          m ** Succ(n) = m * (m ** n)
    endfunc
endmod BasicNaturalNumber
```

A.10 NaturalNumber

```
signature NaturalNumber is
    import BasicNaturalNumber + Boolean
    function _eq_ (m: Nat, n: Nat) : Bool reqs equality (eq)
    function _ne_ (m: Nat, n: Nat) : Bool
    function _lt_ (m: Nat, n: Nat) : Bool
    function _le_ (m: Nat, n: Nat) : Bool
    function _ge_ (m: Nat, n: Nat) : Bool
    function _gt_ (m: Nat, n: Nat) : Bool
endsig NaturalNumber
module NaturalNumber is
    import BasicNaturalNumber + Boolean
    function _eq_ (m: Nat, n: Nat) : Bool is
        case
          m :: 0 and n :: 0 -> true
        | (m :: 0 and n :: Succ (...)) or
          (m :: Succ (...) and n :: 0) -> false
        |  m :: Succ (N:=m1: Nat) and n :: Succ (N:=n1: Nat) -> m1 eq n1
        endcase
    eqns forall m, n : Nat
          0 eq 0 = true;
          0 eq Succ(m) = false;
          Succ(m) eq 0 = false;
```

```
        Succ(m) eq Succ(n) = m eq n;
    endfunc
    function _ne_ (m: Nat, n: Nat) : Bool is not (m eq n)
    eqns forall m, n : Nat
        m ne n = not(m eq n);
    endfunc
    function _lt_ (m: Nat, n: Nat) : Bool is
      case
        (m :: 0 or m :: Succ (...)) and n :: 0 -> false
      | m :: 0 and n :: Succ (...) -> true
      | m :: Succ (N:=m1: Nat) and n :: Succ (N:=n1: Nat) -> m1 lt n1
      endcase
    eqns forall m, n : Nat
        0 lt 0 = false;
        0 lt Succ(n) = true;
        Succ(n) lt 0 = false;
        Succ(m) lt Succ(n) = m lt n;
    endfunc
    function _le_ (m: Nat, n: Nat) : Bool is m lt n or (m eq n)
    eqns forall m, n : Nat
        m le n = m lt n or (m eq n);
    endfunc
    function _ge_ (m: Nat, n: Nat) : Bool is not(m lt n)
    eqns forall m, n : Nat
        m ge n = not(m lt n);
    endfunc
    function _gt_ (m: Nat, n: Nat) : Bool is not(m le n)
    eqns forall m, n : Nat
        m gt n = not(m le n);
    endfunc
endmod NaturalNumber
```

A.11 NatRepresentations

```
signature NatRepresentations is
    import HexNatRepr + DecNatRepr + OctNatRepr + BitNatRepr
endsig
```

A.12 HexNatRepr

```
signature HexNatRepr is
   import HexString
   function NatNum (hs: HexString) : Nat
endsig
module HexNatRepr is
   import HexString
   function NatNum (hs: HexString) : Nat is
      case hs in
        Hex (h:=h1: HexDigit) -> NatNum (h1)
      | (h:=h1: HexDigit) + (hs:=hs1: HexString) ->
            NatNum(h1) * (Succ(NatNum(F)) ** Length(hs1)) + NatNum(hs1)
      endcase
   eqns forall hs : HexString , h : HexDigit
   NatNum(Hex(h)) = NatNum (h);
   NatNum(h + hs) = NatNum(h) * (Succ(NatNum(F)) ** Length(hs)) + NatNum(hs)
endmod
```

A.13 HexString

```
signature HexString is
   NonEmptyString [HexDigit :: types HexDigit   for Element
                               Bool      for FBool
                               HexString  for NonEmptyString
                   constructors Hex for String
                   labels h  for e
                          hs for s
                ]
endsig
module HexString is
   NonEmptyString [HexDigit :: types HexDigit   for Element
                               Bool      for FBool
                               HexString  for NonEmptyString
                   constructors Hex for String
                   labels h  for e
                          hs for s
                ]
```

endmod

A.14 HexDigit

```
signature HexDigit is
   import Boolean + NaturalNumber
   type HexDigit is
      0| 1| 2| 3| 4| 5| 6| 7| 8| 9| A| B| C| D| E| F
   endtype
   function _eq_ (x: HexDigit, y: HexDigit) : Bool reqs equality (eq)
   function _ne_ (x: HexDigit, y: HexDigit) : Bool
   function _lt_ (x: HexDigit, y: HexDigit) : Bool
   function _le_ (x: HexDigit, y: HexDigit) : Bool
   function _ge_ (x: HexDigit, y: HexDigit) : Bool
   function _gt_ (x: HexDigit, y: HexDigit) : Bool
   function NatNum (x: HexDigit) : Nat
endsig HexDigit
module HexDigit is
   import Boolean + NaturalNumber
   type HexDigit is
      0| 1| 2| 3| 4| 5| 6| 7| 8| 9| A| B| C| D| E| F
   endtype
   function _eq_ (x: HexDigit, y: HexDigit) : Bool is NatNum(x) eq NatNum(y)
   eqns forall x, y : HexDigit
        x eq y = NatNum(x) eq NatNum(y);
   endfunc
   function _ne_ (x: HexDigit, y: HexDigit) : Bool is NatNum(x) ne NatNum(y)
   eqns forall x, y : HexDigit
        x ne y = NatNum(x) ne NatNum(y);
   endfunc
   function _lt_ (x: HexDigit, y: HexDigit) : Bool is NatNum(x) lt NatNum(y)
   eqns forall x, y : HexDigit
        x lt y = NatNum(x) lt NatNum(y);
   endfunc
   function _le_ (x: HexDigit, y: HexDigit) : Bool is NatNum(x) le NatNum(y)
   eqns forall x, y : HexDigit
        x le y = NatNum(x) le NatNum(y);
```

```
endfunc
function _ge_ (x: HexDigit, y: HexDigit) : Bool is NatNum(x) ge NatNum(y)
eqns forall x, y : HexDigit
    x ge y = NatNum(x) ge NatNum(y);
endfunc
function _gt_ (x: HexDigit, y: HexDigit) : Bool is NatNum(x) gt NatNum(y)
eqns forall x, y : HexDigit
    x gt y = NatNum(x) gt NatNum(y);
endfunc
function NatNum (x: HexDigit) : Nat is
  case x in
    0 -> 0
  | 1 -> Succ(NatNum(0))
  | 2 -> Succ(NatNum(1))
  | 3 -> Succ(NatNum(2))
  | 4 -> Succ(NatNum(3))
  | 5 -> Succ(NatNum(4))
  | 6 -> Succ(NatNum(5))
  | 7 -> Succ(NatNum(6))
  | 8 -> Succ(NatNum(7))
  | 9 -> Succ(NatNum(8))
  | A -> Succ(NatNum(9))
  | B -> Succ(NatNum(A))
  | C -> Succ(NatNum(B))
  | D -> Succ(NatNum(C))
  | E -> Succ(NatNum(D))
  | F -> Succ(NatNum(E))
  endcase
eqns
    NatNum(0) = 0;
    NatNum(1) = Succ(NatNum(0));
    NatNum(2) = Succ(NatNum(1));
    NatNum(3) = Succ(NatNum(2));
    NatNum(4) = Succ(NatNum(3));
    NatNum(5) = Succ(NatNum(4));
    NatNum(6) = Succ(NatNum(5));
    NatNum(7) = Succ(NatNum(6));
```

```
        NatNum(8) = Succ(NatNum(7));
        NatNum(9) = Succ(NatNum(8));
        NatNum(A) = Succ(NatNum(9));
        NatNum(B) = Succ(NatNum(A));
        NatNum(C) = Succ(NatNum(B));
        NatNum(D) = Succ(NatNum(C));
        NatNum(E) = Succ(NatNum(D));
        NatNum(F) = Succ(NatNum(E))
    endfunc
endmod HexDigit
```

A.15 DecNatRepr

```
signature DecNatRepr is
    import NaturalNumber + DecString
    function NatNum (ds: DecString) : Nat
endsig
module DecNatRepr is
    import NaturalNumber + DecString
    function NatNum (ds: DecString) : Nat is
        case ds in
          Dec (d:=d1: DecString) -> NatNum (d1)
        | (d:=d1: DecDigit) + (ds:=ds1: DecString) ->
              NatNum(d1) * (Succ(NatNum(9)) ** Length(ds1)) + NatNum(ds1)
        endcase
    eqns forall ds : DecString , d : DecDigit
    NatNum(Dec(d)) = NatNum(d);
    NatNum(d + ds) = NatNum(d) * (Succ(NatNum(9)) ** Length(ds)) + NatNum(ds)
    endfunc
endmod
```

A.16 DecString

```
signature DecString is
    NonEmptyString [ DecDigit :: types DecDigit   for Element
                                    Bool      for FBool
                                    DecString  for NonEmptyString
                               constructors Dec for String
```

```
                    labels d  for e
                           ds for s
          ]
endsig
module DecString is
   NonEmptyString [ DecDigit :: types DecDigit   for Element
                                 Bool      for FBool
                                 DecString  for NonEmptyString
                    constructors Dec for String
                    labels d  for e
                           ds for s
          ]
endmod
```

A.17 DecDigit

```
signature DecDigit is
   import NaturalNumber + Boolean
   type DecDigit is
       0| 1| 2| 3| 4| 5| 6| 7| 8| 9
   endtype
   function _eq_ (x: DecDigit, y: DecDigit) : Bool reqs equality (eq)
   function _ne_ (x: DecDigit, y: DecDigit) : Bool
   function _lt_ (x: DecDigit, y: DecDigit) : Bool
   function _le_ (x: DecDigit, y: DecDigit) : Bool
   function _ge_ (x: DecDigit, y: DecDigit) : Bool
   function _gt_ (x: DecDigit, y: DecDigit) : Bool
   function NatNum  (x: DecDigit) : Nat
endsig
module DecDigit is
   import NaturalNumber + Boolean
   type DecDigit is
       0| 1| 2| 3| 4| 5| 6| 7| 8| 9
   endtype
   function _eq_ (x: DecDigit, y: DecDigit) : Bool is NatNum(x) eq NatNum(y)
   eqns forall x, y : DecDigit
       x eq y = NatNum(x) eq NatNum(y);
```

```
endfunc
function _ne_ (x: DecDigit, y: DecDigit) : Bool is NatNum(x) ne NatNum(y)
eqns forall x, y : DecDigit
    x ne y = NatNum(x) ne NatNum(y);
endfunc
function _lt_ (x: DecDigit, y: DecDigit) : Bool is NatNum(x) lt NatNum(y)
eqns forall x, y : DecDigit
    x lt y = NatNum(x) lt NatNum(y);
endfunc
function _le_ (x: DecDigit, y: DecDigit) : Bool is NatNum(x) le NatNum(y)
eqns forall x, y : DecDigit
    x le y = NatNum(x) le NatNum(y);
endfunc
function _ge_ (x: DecDigit, y: DecDigit) : Bool is NatNum(x) ge NatNum(y)
eqns forall x, y : DecDigit
    x ge y = NatNum(x) ge NatNum(y);
endfunc
function _gt_ (x: DecDigit, y: DecDigit) : Bool is NatNum(x) gt NatNum(y)
eqns forall x, y : DecDigit
    x gt y = NatNum(x) gt NatNum(y);
endfunc
function NatNum  (x: DecDigit) : Nat is
   case x in
     0 -> 0
   | 1 -> Succ(NatNum(0))
   | 2 -> Succ(NatNum(1))
   | 3 -> Succ(NatNum(2))
   | 4 -> Succ(NatNum(3))
   | 5 -> Succ(NatNum(4))
   | 6 -> Succ(NatNum(5))
   | 7 -> Succ(NatNum(6))
   | 8 -> Succ(NatNum(7))
   | 9 -> Succ(NatNum(8))
   endcase
eqns forall x, y : DecDigit
    NatNum(0) = 0;
    NatNum(1) = Succ(NatNum(0));
```

```
      NatNum(2) = Succ(NatNum(1));
      NatNum(3) = Succ(NatNum(2));
      NatNum(4) = Succ(NatNum(3));
      NatNum(5) = Succ(NatNum(4));
      NatNum(6) = Succ(NatNum(5));
      NatNum(7) = Succ(NatNum(6));
      NatNum(8) = Succ(NatNum(7));
      NatNum(9) = Succ(NatNum(8));
   endfunc
endmod
```

A.18 OctNatRepr

```
signature OctNatRepr is
   import OctString
   function NatNum (os: OctString) : Nat
endsig
module OctNatRepr is
   import OctString
   function NatNum (os: OctString) : Nat is
      case os in
        Oct(o:=o1: OctDigit) -> NatNum(o1)
      | (o:=o1: OctDigit) + (os:=os1: OctString) ->
            NatNum(o1) * (Succ(NatNum(7)) ** Length(os1)) + NatNum(os1)
      endcase
   eqns forall os : OctString, o : OctDigit
   NatNum(Oct(o)) = NatNum(o);
   NatNum(o + os) = NatNum(o) * (Succ(NatNum(7)) ** Length(os)) + NatNum(os)
   endfunc
endmod
```

A.19 OctString

```
signature OctString is
   NonEmptyString [OctDigit :: types OctDigit  for Element
                               Bool      for FBool
                               OctString for NonEmptyString
                   constructors Oct  for String
```

```
                    labels o  for e
                           os for s
            ]
endsig
module OctString is
   NonEmptyString [OctDigit :: types OctDigit  for Element
                               Bool     for FBool
                               OctString for NonEmptyString
                    constructors Oct for String
                    labels o  for e
                           os for s
            ]
endmod
```

A.20 OctDigit

```
signature OctDigit is
   import NaturalNumber + Boolean
   type OctDigit is
      0| 1| 2| 3| 4| 5| 6| 7
   endtype
   function _eq_ (x: OctDigit, y: OctDigit) : Bool reqs equality (eq)
   function _ne_ (x: OctDigit, y: OctDigit) : Bool
   function _lt_ (x: OctDigit, y: OctDigit) : Bool
   function _le_ (x: OctDigit, y: OctDigit) : Bool
   function _ge_ (x: OctDigit, y: OctDigit) : Bool
   function _gt_ (x: OctDigit, y: OctDigit) : Bool
   function NatNum (x: OctDigit) : Nat
endsig OctDigit
module OctDigit is
   import NaturalNumber + Boolean
   type OctDigit is
      0| 1| 2| 3| 4| 5| 6| 7
   endtype
   function _eq_ (x: OctDigit, y: OctDigit) : Bool is NatNum(x) eq NatNum(y)
   eqns forall x, y : OctDigit
        x eq y = NatNum(x) eq NatNum(y);
```

```
endfunc
function _ne_ (x: OctDigit, y: OctDigit) : Bool is NatNum(x) ne NatNum(y)
eqns forall x, y : OctDigit
    x ne y = NatNum(x) ne NatNum(y);
endfunc
function _lt_ (x: OctDigit, y: OctDigit) : Bool is NatNum(x) lt NatNum(y)
eqns forall x, y : OctDigit
    x lt y = NatNum(x) lt NatNum(y);
endfunc
function _le_ (x: OctDigit, y: OctDigit) : Bool is NatNum(x) le NatNum(y)
eqns forall x, y : OctDigit
    x le y = NatNum(x) le NatNum(y);
endfunc
function _ge_ (x: OctDigit, y: OctDigit) : Bool is NatNum(x) ge NatNum(y)
eqns forall x, y : OctDigit
    x ge y = NatNum(x) ge NatNum(y);
endfunc
function _gt_ (x: OctDigit, y: OctDigit) : Bool is NatNum(x) gt NatNum(y)
eqns forall x, y : OctDigit
    x gt y = NatNum(x) gt NatNum(y)
endfunc
function NatNum (x: OctDigit) : Nat is
  case x in
    0 -> 0
  | 1 -> Succ(NatNum(0))
  | 2 -> Succ(NatNum(1))
  | 3 -> Succ(NatNum(2))
  | 4 -> Succ(NatNum(3))
  | 5 -> Succ(NatNum(4))
  | 6 -> Succ(NatNum(5))
  | 7 -> Succ(NatNum(6))
  endcase
eqns forall x: OctDigit
    NatNum(0) = 0;
    NatNum(1) = Succ(NatNum(0));
    NatNum(2) = Succ(NatNum(1));
    NatNum(3) = Succ(NatNum(2));
```

```
        NatNum(4) = Succ(NatNum(3));
        NatNum(5) = Succ(NatNum(4));
        NatNum(6) = Succ(NatNum(5));
        NatNum(7) = Succ(NatNum(6))
    endfunc
endmod OctDigit
```

A.21 BitNatRepr

```
signature BitNatRepr is
    import BitString
    function NatNum (bs: BitString) : Nat
endsig
module BitNatREpr is
    import BitString
    function NatNum (bs: BitString) : Nat is
        case bs in
          Bit (b:=b1: Bit) -> NatNum (b)
        | (b:=b1: Bit) + (bs:=bs1: BitString) ->
              NatNum(b1) * (Succ(NatNum(1)) ** Length(bs1)) + NatNum(bs1)
        endcase
    eqns forall  bs : BitString, b : Bit
    NatNum(Bit(b)) = NatNum(b);
    NatNum(b + bs) = NatNum(b) * (Succ(NatNum(1)) ** Length(bs)) + NatNum(bs)
    endfunc
endmod
```

A.22 BitString

```
signature BitString is
    NonEmptyString [ Bit :: types Bit        for Element
                            Bool       for FBool
                            BitString  for NonEmptyString
                     constructors Bit for String
                     labels b  for e
                            bs for s
                ]
endsig
```

```
module BitString is
   NonEmptyString [ Bit :: types Bit        for Element
                            Bool       for FBool
                            BitString  for NonEmptyString
                    constructors Bit for String
                    labels b  for e
                           bs for s
                 ]
endmod
```

A.23 Bit

```
signature Bit is
   import NaturalNumber + Boolean
   type Bit is
      0| 1
   endtype
   function _eq_ (x: Bit, y: Bit) : Bool reqs equality (eq)
   function _ne_ (x: Bit, y: Bit) : Bool
   function _lt_ (x: Bit, y: Bit) : Bool
   function _le_ (x: Bit, y: Bit) : Bool
   function _ge_ (x: Bit, y: Bit) : Bool
   function _gt_ (x: Bit, y: Bit) : Bool
   function NatNum (x: Bit) : Nat
endsig Bit
module Bit is
   import NaturalNumber + Boolean
   type Bit is
      0| 1| 2| 3| 4| 5| 6| 7
   endtype
   function _eq_ (x: Bit, y: Bit) : Bool is NatNum(x) eq NatNum(y)
   eqns forall x, y : Bit
        x eq y = NatNum(x) eq NatNum(y);
   endfunc
   function _ne_ (x: Bit, y: Bit) : Bool is NatNum(x) ne NatNum(y)
   eqns forall x, y : Bit
        x ne y = NatNum(x) ne NatNum(y);
```

```
    endfunc
    function _lt_ (x: Bit, y: Bit) : Bool is NatNum(x) lt NatNum(y)
    eqns forall x, y : Bit
        x lt y = NatNum(x) lt NatNum(y);
    endfunc
    function _le_ (x: Bit, y: Bit) : Bool is NatNum(x) le NatNum(y)
    eqns forall x, y : Bit
        x le y = NatNum(x) le NatNum(y);
    endfunc
    function _ge_ (x: Bit, y: Bit) : Bool is NatNum(x) ge NatNum(y)
    eqns forall x, y : Bit
        x ge y = NatNum(x) ge NatNum(y);
    endfunc
    function _gt_ (x: Bit, y: Bit) : Bool is NatNum(x) gt NatNum(y)
    eqns forall x, y : Bit
        x gt y = NatNum(x) gt NatNum(y)
    endfunc
    function NatNum (x: Bit) : Nat is
      case x in
        0 -> 0
      | 1 -> Succ(NatNum(0))
      endcase
    eqns forall x: Bit
        NatNum(0) = 0;
        NatNum(1) = Succ(NatNum(0));
    endfunc
endmod Bit
```

A.24 Octet

```
signature Octet is
    import Bit + Boolean
    type Octet is
      Octet (b1: Bit, b2: Bit, b3: Bit, b4: Bit, b5: Bit, b6: Bit, b7: Bit, b8:Bit)
    endtype
    function Bit1 (o: Octet) : Bit
    function Bit2 (o: Octet) : Bit
```

```
    function Bit3 (o: Octet) : Bit
    function Bit4 (o: Octet) : Bit
    function Bit5 (o: Octet) : Bit
    function Bit6 (o: Octet) : Bit
    function Bit7 (o: Octet) : Bit
    function Bit8 (o: Octet) : Bit
    function _eq_ (x: Octet, y: Octet) : Bool reqs equality (eq)
    function _ne_ (x: Octet, y: Octet) : Bool
endsig
module Octet is
    import Bit + Boolean
    type Octet is
        Octet (b1: Bit, b2: Bit, b3: Bit, b4: Bit, b5: Bit, b6: Bit, b7: Bit, b8:Bit)
    endtype
    function Bit1 (o: Octet) : Bit is o.b1
    eqns forall b1, b2, b3, b4, b5, b6, b7, b8 : Bit
        Bit1(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) = b1;
    endfunc
    function Bit2 (o: Octet) : Bit is o.b2
    eqns forall b1, b2, b3, b4, b5, b6, b7, b8 : Bit
        Bit1(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) = b2;
    endfunc
    function Bit3 (o: Octet) : Bit is o.b3
    eqns forall b1, b2, b3, b4, b5, b6, b7, b8 : Bit
        Bit1(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) = b3;
    endfunc
    function Bit4 (o: Octet) : Bit is o.b4
    eqns forall b1, b2, b3, b4, b5, b6, b7, b8 : Bit
        Bit1(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) = b4;
    endfunc
    function Bit5 (o: Octet) : Bit is o.b5
    eqns forall b1, b2, b3, b4, b5, b6, b7, b8 : Bit
        Bit1(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) = b5;
    endfunc
    function Bit6 (o: Octet) : Bit is o.b6
    eqns forall b1, b2, b3, b4, b5, b6, b7, b8 : Bit
        Bit1(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) = b6;
```

```
    endfunc
    function Bit7 (o: Octet) : Bit is o.b7
    eqns forall b1, b2, b3, b4, b5, b6, b7, b8 : Bit
        Bit1(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) = b7;
    endfunc
    function Bit8 (o: Octet) : Bit is o.b8
    eqns forall b1, b2, b3, b4, b5, b6, b7, b8 : Bit
        Bit1(Octet(b1, b2, b3, b4, b5, b6, b7, b8)) = b8;
    endfunc
    function _eq_ (x: Octet, y: Octet) : Bool
    eqns forall x, y : Octet
       x eq y = Bit1(x) eq Bit1(y) and (Bit2(x) eq Bit2(y)) and
            (Bit3(x) eq Bit3(y)) and (Bit4(x) eq Bit4(y)) and
            (Bit5(x) eq Bit5(y)) and (Bit6(x) eq Bit6(y)) and
            (Bit7(x) eq Bit7(y)) and (Bit8(x) eq Bit8(y));
    endfunc
    function _ne_ (x: Octet, y: Octet) : Bool is not(x eq y)
    eqns forall x, y : Octet
        x ne y = not(x eq y)
    endfunc
endmod Octet
```

A.25 OctetString

```
signature OctetString is
   String [ Octet :: types Octet       for Element
                       Bool        for FBool
                       OctetString for String
                  funnames Octet for String
          ]
endsig
module OctetString is
   String [ Octet :: types Octet       for Element
                       Bool        for FBool
                       OctetString for String
                  labels o  for e
                         os for s
```

funnames Octet **for** String

]

endmod

B Translation of LOTOSPHERE data types in E-LOTOS

This Annex indicates how some of the data types of the so-called "LOTOSPHERE proposal" (output document of the Yokohama SC21 meeting, July 1993) can be expressed using the proposed User Language for E-LOTOS. It is worth noticing that:

- The LOTOSPHERE proposal only defines interfaces (signatures) rather than the implementation of modules themselves.
- In this Annex, we stress the differences (i.e., enhancements) between LOTOS standard library and the LOTOSPHERE proposal. We do not repeat thoses definitions which are similar as those given in Annex A.
- The LOTOSPHERE proposal uses "`==`", "`!=`", "`<`", "`<=`", "`>`", and "`>=`" where the LOTOS library uses "`eq`", "`ne`", "`lt`", "`le`", "`gt`", and "`ge`". In E-LOTOS, we might wish to allow both notation to ensure both upward compatibility and user-friendliness.

B.1 BOOL

Same as in Annex A, except that "`implies`" and "`iff`" are respectively noted "`==>`" and "`<==>`".

B.2 NAT

In the LOTOSPHERE proposal, natural numbers are a built-in type, not defined by "`0`" and "`Succ`" constructors. Natural numbers have an arbitrary, unlimited precision. They can be written using decimal notation (e.g., "`123`"), but also in binary, octal, or hexadecimal notations (e.g., "`bin (100101101)`", "`oct (455)`", "`hex (12d)`".

A unary function "`pred`" and a substraction operator are available, which raise an underflow error if a negative number is to be returned. Similarly, division and modulo operators are available, which can raise a division-by-zero error.

signature NAT **is**

type Nat **is** <decimal digit sequence> **endtype**

function bin (<binary digit sequence>) : Nat

function oct (<octal digit sequence>) : Nat

function hex (<hexadecimal digit sequence>) : Nat

function succ (Nat) : Nat

function pred [UNDERFLOW] : Nat

function _+_ (Nat, Nat) : Nat

function _-_ [UNDERFLOW] (Nat, Nat) : Nat

function _*_ (Nat, Nat) : Nat

function _/_ [ZERO_DVISION] (Nat, Nat) : Nat

function rem [ZERO_DVISION] (Nat, Nat) : Nat

function pow (Nat, Nat) : Nat

function max (Nat, Nat) : Nat

function min (Nat, Nat) : Nat

function _>_ (Nat, Nat) : Bool

function _>=_ (Nat, Nat) : Bool

function _<_ (Nat, Nat) : Bool

function _<=_ (Nat, Nat) : Bool

function _==_ (Nat, Nat) : Bool **function** _!=_ (Nat, Nat) : Bool

endsig NAT

B.3 INT

The `INT` type represented signed integers of arbitrary precision. As the `NAT` type, it is a built-in type, not defined by constructors. Values are denoted by signed decimal numbers.

signature INT **is**

type Int **is**

<decimal digit sequence>

| -<decimal digit sequence>

endtype

function bin (<binary digit sequence>) : Int

function oct (<octal digit sequence>) : Int

function hex (<hexadecimal digit sequence>) : Int

function succ (Int) : Int

function pred (Int) : Int

function minus (Int) : Int

function _+_ (Int, Int) : Int

function _-_ (Int, Int) : Int

function _*_ (Int, Int) : Int

function _/_ [ZERO_DIVISION] (Int, Int) : Int

function rem [ZERO_DVISION] (Int, Int) : Int

function pow (Int, Int) : Int

function max (Int, Int) : Int

function min (Int, Int) : Int

function abs (Int) : Int

function _>_ (Int, Int) : Bool

function _>=_ (Int, Int) : Bool

function _<_ (Int, Int) : Bool

function _<=_ (Int, Int) : Bool

function _==_ (Int, Int) : Bool **function** _!=_ (Int, Int) : Bool

endsig INT

B.4 FRAC

The "FRAC" type represents rational numbers. We present its interface and provide an implementation using the User Language for E-LOTOS:

signature FRAC **is**

import INT

type Frac **is**

frac (num: Int, den: Int) **where** den > 0

endtype

function _+_ (Frac, Frac) : Frac

function _-_ (Frac, Frac) : Frac

function _*_ (Frac, Frac) : Frac

function _/_ [ZERO_DVISION] (Frac, Frac) : Frac

function pow (Frac, Int) : Frac

function max (Frac, Frac) : Frac

function min (Frac, Frac) : Frac

function abs (Frac) : Frac

function round (x: Frac) : Frac

```
(* this function returns the nearest integral value to x, except for
   halfway cases, which are rounded to the integral value larger in
   magnitude *)
```

function ceil (x: Frac) : Frac

```
(* this function returns the least integral value greater than or equal to x *)
```

function floor (x: Frac) : Frac

```
(* this function returns the greatest value less than or equal to x *)
```

function _>_ (Frac, Frac) : Bool

function _>=_ (Frac, Frac) : Bool

function _<_ (Frac, Frac) : Bool

function _<=_ (Frac, Frac) : Bool

function _==_ (Frac, Frac) : Bool **function** _!=_ (Frac, Frac) : Bool

endsig FRAC

A tentative implementation is given below:

```
module FRAC is
   import INT
   function gd (m: Int, n: Int) is
      case
      n :: 0 -> m
      |  otherwise -> gd (n, m rem n)
      endcase
   endfunc
   function reduce (f: Frac) is
      let gd: Nat = gd (abs (f.num), abs (f.den)) in frac (f.num / gd, f.den / gd)
   endfunc
    function minus (f: Frac) : Frac is
      frac (minus (f.num), f.den)
    endfunc
   function _+_ (f1: Frac, f2: Frac) : Frac is
      reduce (frac (f1.num * f2.den + f2.num * f1.den, f1.den * f2.den))
   endfunc
   function _-_ (f1: Frac, f2: Frac) : Frac is
      reduce (frac (f1.num * f2.den - f2.num * f1.den, f1.den * f2.den))
   endfunc
   function _*_ (f1: Frac, f2: Frac) : Frac is
      reduce (frac (f1.num * f2.num, f1.den * f2.den))
   endfunc
   function _/_ [ZERO_DVISION] (f1: Frac, f2: Frac) : Frac is
      case f2 in
         frac (num:=0, ...) -> raise ZERO_DVISION
      |  otherwise -> reduce (frac (f1.num * f2.den, f2.num * f1.den))
      endcase
   endfunc
   function pow (f: Frac, p: Int) : Frac is
      case p in
         0 -> frac (1, 1)
      |  when p lt 0 -> pow (f, p + 1) / f
      |  when p gt 0 -> pow (f, p - 1) * f
      endcase
   endfunc
```

```
    function max (f1: Frac, f2: Frac) : Frac is
        if f1 >= f2 then f1 else f2
    endfunc
    function min (f1: Frac, f2: Frac) : Frac is
        if f1 >= f2 then f2 else f1
    endfunc
    function abs (f: Frac) : Frac is frac (abs (f.num), f.den) endfunc
    function round (x: Frac) : Frac is
        frac ((x.num / x.den) +
                (if (x.num rem x.den) ge (x.den / 2) then 1 else 0),
                1)
    endfunc
    function ceil  (x: Frac) : Frac is
        frac (  (x.num / x.den) +
                (if (x.num rem x.den) ge 0 then 1 else 0),
                1)
    endfunc
    function floor (x: Frac) : Frac is
        frac ((x.num / x.den), 1)
    endfunc
    function _>_  (f1: Frac, f2: Frac) : Bool is
        (f1.num * f2.den - f2.num * f1.den) > 0
    endfunc
    function _>=_ (f1: Frac, f2: Frac) : Bool is
        (f1.num * f2.den - f2.num * f1.den) >= 0
    endfunc
    function _<_  (f1: Frac, f2: Frac) : Bool is
        (f1.num * f2.den - f2.num * f1.den) < 0
    endfunc
    function _<=_ (f1: Frac, f2: Frac) : Bool is
        (f1.num * f2.den - f2.num * f1.den) <= 0
    endfunc
    function _==_ (f1: Frac, f2: Frac) : Bool is
        (f1.num * f2.den - f2.num * f1.den) == 0
    endfunc
    function _!=_ (f1: Frac, f2: Frac) : Bool is not (f1 == f2) endfunc
endmod FRAC
```

B.5 FLOAT

The "`Float`" type is a built-in type for floating-point numbers, together with the associated functions.

signature FLOAT **is**

type Float **is**

[+|-]<dds>.[<dds>] [E[+|-]<dds>]

endtype

function _+_ (Float, Float) : Float

function _-_ (Float, Float) : Float

function _*_ (Float, Float) : Float

function _/_ [ZERO_DIVISION] (Float, Float) : Float

function max (Float, Float) : Float

function min (Float, Float) : Float

function pow (Float, Float) : Float

function abs (Float) : Float

function sqrt (Float) : Float

function exp (Float) : Float `(* e^x *)`

function log (Float) : Float `(* log_10 x *)`

function sin (Float) : Float

function cos (Float) : Float

function tan (Float) : Float

function asin (Float) : Float

function acos (Float) : Float

function atan (Float) : Float

function sinh (Float) : Float

function cosh (Float) : Float

function tanh (Float) : Float

function asinh (Float) : Float

function acosh (Float) : Float

function atanh (Float) : Float

function pi : Float

function e : Float

function round (x: Float) : Float

```
(* this function returns the nearest integral value to x, except for
   halfway cases, which are rounded to the integral value larger in
   magnitude *)
```

function ceil (x: Float) : Float

```
(* this function returns the least integral value greater than or equal to x *)
```

function floor (x: Float) : Float

`(* this function returns the greatest value less than or equal to x *)`

function _>_ (Float, Float) : Bool

function _>=_ (Float, Float) : Bool

function _<_ (Float, Float) : Bool

function _<=_ (Float, Float) : Bool

function _==_ (Float, Float) : Bool **function** _!=_ (Float, Float) : Bool

endsig FLOAT

B.6 CHAR

signature CHAR **is**

import NAT + BOOLEAN

type Char **is** ...

endtype

function toNat (Char) : Nat

function toChar (Nat) : Char

function tolower (Char) : Char

function toupper (Char) : Char

function isalpha (Char) : Bool

function isdigit (Char) : Bool

function isxdigit (Char) : Bool

function islower (Char) : Bool

function isupper (Char) : Bool

function isalnum (Char) : Bool

function _>_ (Char, Char) : Bool

function _>=_ (Char, Char) : Bool

function _<_ (Char, Char) : Bool

function _<=_ (Char, Char) : Bool

function _==_ (Char, Char) : Bool **function** _!=_ (Char, Char) : Bool

endsig CHAR

B.7 STRING

signature STRING **is**

import NAT + CHAR + BOOLEAN

type string **is** ...

endtype

function length (string) : Nat

function concat (string, string) : string

function prefix (string, Nat) : string

function suffix (string, Nat) : string

function substr (string, Nat, Nat) : string

function index (string, string) : Nat `(* search from left *)`

function rindex (string, string) : Nat `(* search from right *)`

function nth (string, Nat) : Char

function _>_ (string, string) : Bool

function _>=_ (string, string) : Bool

function _<_ (string, string) : Bool

function _<=_ (string, string) : Bool

function _==_ (string, string) : Bool **function** _!=_ (string, string) : Bool

function toString (Char) : String

endsig STRING

References

[BFG+91] Ahmed Bouajjani, Jean-Claude Fernandez, Susanne Graf, Carlos Rodríguez, and Joseph Sifakis. Safety for Branching Time Semantics. In *Proceedings of 18th ICALP*. Springer Verlag, July 1991.

[GM96] Hubert Garavel and Radu Mateescu. French-Romanian Proposal for Capture of Requirements and Expression of Properties in E-LOTOS Modules. Rapport SPECTRE 96-04, VERIMAG, Grenoble, May 1996. Input document [KC4] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Kansas City, Missouri, USA, May, 12–21, 1996.

[ISO88] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[Mou92] Laurent Mounier. *Méthodes de vérification de spécifications comportementales : étude et mise en œuvre*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), January 1992.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The defintion of Standard ML*. MIT Press, 1990.

[Par81] David Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, March 1981.

[Pec94] Charles Pecheur. A proposal for data types for E-LOTOS. Technical Report, University of Liège, October 1994. Annex H of ISO/IEC JTC1/SC21/WG1 N1349 Working Draft on Enhancements to LOTOS.

[Que96a] Juan Quemada, editor. Revised Working Draft on Enhancements to LOTOS (V2). ISO/IEC JTC1/SC21/WG7 N10108 Project 1.21.20.2.3. Output document of the Ottawa meeting, January 1996.

[Que96b] Juan Quemada, editor. Revised Working Draft on Enhancements to LOTOS (V4). ISO/IEC JTC1/SC21/WG7 N1173 Project 1.21.20.2.3. Output document of the Kansas City meeting, October 1996.

[vGW89] R. J. van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989. Also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.

[Wir83] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, Berlin, 1983.