On the Definition of Modular E-LOTOS *

Draft of 1996/12

Mihaela Sighireanu and Hubert Garavel

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

LOTOS has only a limited form of modules, which encapsulates data types and operations, but not processes. Moreover, this mechanism does not support abstraction: every object declared in a module is exported outside. These deficiencies make LOTOS difficult to use, and cause problems for users and tool implementors alike. A critical evaluation of LOTOS data types from the user point of view can be found, for instance, in [Mun91].

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, interfaces and generic modules are very useful.

For example, a simple router of packets containing a data field and an address field must be defined in standard LOTOS:

specification Router [in, left, right] : **noexit** :=

type Natural **is**

...

endtype

type Data **is**

formalsorts data

endtype

```
type Packet is Data
    sorts
        packet, dest
    opns
        mkpacket : dest, data  →  packet
        getdest : packet  →  dest
        getdata : packet  →  data
        L : -> dest
        R : -> dest
     eqns forall p:packet, de:dest, da:data
         ofsort packet
             mkpacket (getdest (p), getdata (p)) = p ;
         ofsort dest
             getdest (mkpacket (de, da)) = de ;
         ofsort data
             getdata (mkpacket (de, da)) = da ;
endtype
 type NatPacket is Packet actualizedby Natural
 using
        natpacket for packet
endtype
behaviour Router [in, left, right]
   where
   process Router [in, left, right] : noexit :=
        in ? p : natpacket; (* packet cannot be used here ! *)
        (
          [getdest (p) = L]  →  left!getdata(p); Router [in, left, right]
        [] [getdest (p) = R]  →  right!getdata(p); Router [in, left, right]
        )
   endproc
endspec
```

This definition suffers from some problems of readability for non-LOTOS experts as mentioned in [JL96]. Moreover, the Router process specification cannot be instantiated for some (different) types of data because the behaviour part of standard LOTOS must refer only fully instantiated types.

This can be compared with the equivalent module declaration in the module language presented here (the base language is one of [JL96]):

interface Data **is**

  type data

endint

generic Router (D : Data) **is**

  type dest **is** L | R **endtype**

  type packet **is** <de → dest, da → D.data> **endtype**

  gate In : packet, Left : data, Right : data

  process Router [in : packet, left : data, right : data] : **noexit is**

    ...

  endproc

endgen

module NatRouter **is** Router (Natural) **endmod**

open NatRouter

behaviour main : **noexit is** Router [In, Left, Right]

Note that:

- Abstract data types, as 'data', can be declared into interfaces.
- Generic modules are parameterized by interfaces, in a functional style.
- Gates can be declared in modules and generic modules.
- Process and behaviours can be declared in modules and generic modules; no fully instantiated types are needed to specify behaviours.
- The dot notation is used to access identifiers of the parameters of generic modules. However, to avoid a clumsy notation, modules can be opened to access their objects directly.
- Behaviour constants can be declared as well as value constants. One may declare a "main" behaviour or a "main" value.
- Here the types "packet" is named records.

In this paper we present a solution for the module system of E-LOTOS with a base language as defined in [Que96b]. To attain it, we present a complete abstract syntax, some derived forms (as syntactic sugar), and a static semantics. The flattening function should be provided.

The abstract syntax contains: module declarations, module expressions, declarations, generic module declarations, interface declarations, interface expressions, and interface specification. Some means for static sub-typing, equality types, and finite types are provided.

The static semantics of this language is formally defined. It is based on judgments such as $\mathcal{B} \vdash$ *mod-dec* $\Rightarrow \mathcal{C}$ meaning 'in the context $\mathcal{B}$ of declarations, the module definition *mod-dec* is well formed and gives the context $\mathcal{C}$'. For example:

$$\mathbf{int} \Rightarrow \texttt{type} \quad \vdash \quad (\mathbf{module}\ \text{OnePoint}\ \mathbf{is}\ \mathbf{type}\ M\ \mathbf{is}\ < x : int, y : int >) \\ \Rightarrow \text{OnePoint} \Rightarrow \{M \Rightarrow \mathbf{type}, M \Rightarrow (x : \mathbf{int}, y : \mathbf{int}) \rightarrow M\}$$

means 'in a context where the type **int** is declared as a type, then the OnePoint module definition is bound to a context (signature) where the type M is declared, and the constructor of this type is also declared'. The static semantics includes:

- Abstract data types, equality types (private), finite types, and static sub-typing of types.
- Modules declaration and module expressions.
- Generic modules parameterized by interfaces, and generic module instantiation.
- Interface declarations and expressions.
- Enrichment, sub-typing, and matching relation between interfaces and modules semantic objects.
- Renaming of modules and interfaces.

The module language described in this paper is based on previous discussions for module language [JGL+95] and on previous proposals for LOTOS with a module system [SG96].

2 Basic concepts

We provide here a short introduction to this proposal by giving example of how it responds to issues formulated in the questionnaire of [Que96a, Section 7.2].

2.1 Naming

The domain of names is structured, so the module system requires **an extensions to the base language**: identifiers such as constants and functions can be specified as coming from a particular module, by using the "dotted" notation for identifiers, for example:

```
function even (x:Natural.Nat) : Boolean.Bool :=
  x Natural.mod Natural.2 Boolean.= Natural.0
endfun
```

As can be seen from the above example it is very **clumsy** to explicitly name the module that every identifier comes from. So a often we will wish to **open** a module, so its contents can be used without naming the module every time. For example:

```
open Natural
open Boolean
function even (x:Nat) : Bool :=
  x mod 2 = 0
endfun
```

How long there are no nested modules in the language, the prefix of the identifiers has at the most the length one.

2.2 Specification structuring

A specification in modular-E-LOTOS is given as a sequence of base language *declarations*, *module* declarations, *generic modules* declarations, and *interface* declarations. So we provides separate declarations of module interfaces and the module implementation.

Modules Modules are sequences of *declarations* of types, constructors, processes, functions, gates, exceptions, value constants, behaviour constants, and equations. For types, functions, processes, values constants, and behaviour constants the user has to provide an implementations. Modules can be declared using *module declarations*, whose simplest form is:

module *mod-id* **is** *dec* **endmod**

where *mod-id* is a module identifier and *dec* is a (base language) declaration, enriched with gate, exceptions, and constant declarations. For example, the one-point domain has the following declaration:

```
module OnePoint is
  type M is zero () endtype ;
  value 0 : M is zero () ;
  function + (x: M, y: M) : M is
    case
      !x::zero () and !y::zero ()  →  0
    endcase
  endfun
endmod
```

It is worth noticing that, unlike SML module system, 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 and it is not obvious whether the extra complexity is necessary.

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. For example, the data-flow processes can be specified as follows:

```
module DataFlow is
  open 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
endmod
```

Interfaces Intuitively, an *interface* is a module export. Whereas a module expression declares a module, an interface expression specifies a module (or rather a class of modules). For example, the data-flow module has the interface:

interface DataFlow **is**

type In

type Out

process Flow [in : In, out : Out] : **noexit**

endint

An interface is not the type of any particular module, but rather of a whole class of modules, namely all the modules that *matches* the interface (see Section 3.2.6 for a formal definition). For example, the interface DataFlow can be matched by any module which has at least a type In, a type Out, and a process with gates of type In and Out and with functionality noexit.

In the language we accept equations to be specified for a given module or interface. For example, the Monoid interface is:

interface Monoid **is**

type M

value 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

endint

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.

Generic Modules Genericity is a useful tool to construct specifications and for code reuse. Here we provide a mean for genericity using generic modules. Generic modules allow standard libraries of components to be built up, and supports code reuse of both components and 'glue'. The simplest declaration of a generic modules is:

generic *gen-id* (*mod-id* : *int-id*) **is** *dec* **endgen**

where *gen-id* is a generic module identifier, *mod-id* is a identifier for a formal module matching the interface *int-id*, and *dec* are declarations of the base language.

It is worth noticing that generic modules cannot be parameterized by generic modules.

So, as well as specifying the exports of a module, interfaces are also used to specify the parameters of a generic module. For example, a generic lists module can be implemented as using monomorphic lists as follows:

interface List **is**

open Monoid

type E

function inj (x: E) : M

```
endint
interface EqType is
  type E private (* type with equality and assignment, see next section *)
endint
generic GenericList (E : EqType) : List is
  open E
  type M  is nil () | cons (x: E, s: M) endtype
  function + (s1: M, s2: M) : M  is
    case
      s1::nil () -> s2
    | s1::cons(x, xs) -> cons (x, + (xs, ys))
  endfun
  function inj (x : E) : M is
    cons (x, nil())
  endfun
endgen
```

This module can be used as follows:

```
module ListNat : List is GenericList (Natural)  endmod
module ListBool : List is GenericList (Boolean) endmod
```

However, to avoid explicit instantiations, for lists, arrays, sets, some rich syntax is suitable 2.

2.3 Abstraction, hiding

Abstraction is present at two levels: abstract data types, and local declarations in modules and interfaces.

In interfaces, an abstract data types can be specified by:

```
type S [any]
```

the attribute "**any**" which is used by default. This abstract data types may be implemented by a lot of concrete (or manifest) data types. For example, if we declare:

```
interface Set  is
  include Boolean
  type Element
  type Set
  value empty : Set
  function insert (x: Element, s: Set) : Set
  function delete (x: Element, s: Set) : Set
  function member (x: Element, s: Set) : Bool
endint
```

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

An issue of abstract data types (ADT) is type equality. Equality is an important concept in LOTOS, since it is used implicitly by synchronization. So far, all types allow equality, but the module system 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. The advantages of not making abstract data types equality types are:

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

Private types (as "eqtype" in SML and "private" types in Ada) provide a mean to specify in interfaces when an abstract data type has an equality function. They are subtypes of "**any**" ADT. For example, the element type for generic list presented in the previous section is an equality type:

interface EqType **is**

type E **private**

endint

The parallel over values in the base language, "**par** $V : S \,||\, B$" in [Que96b], need finite data types for S. If a concrete type is given, some algorithms can check statically if the type is finite. Abstract data types cannot be checked because it is no longer possible to see its internal representation. So, we provide a mean to specify explicitly this, using "**finite**" attribute for type interfaces. This declaration must be verified statically, at instantiation of the ADT. The finite data types are a subtype of ADT.

Static sub-typing is provided by means of abstract data types declared as subtypes of constructed types. For example, the declaration of the following ADT:

type GenHeader **subtypeof**

Req (S: address, D: address, **etc**)

| Res (S: address, D: address, **etc**)

| **any**

endtype

has as subtype the following concrete type:

type GenHeader **subtypeof**

Req (S: address, D: address, I: info)

| Res (S: address, D: address, I: info)

| Can (S: address, I: info)

endtype

The difference with the built-in sub typing of anonymous records proposed in the base language is the possibility to statically type check and an efficient implementation.

The second level of abstraction is at the module level. There are several means to endure abstraction and hiding:

1. For interfaces, the "**local**" declaration hides some parts used in the in the specification to provide binding depending on context. For example, another specification of Set interface is:

   interface Set **is**

```
include Boolean
local
  type Element
in
  type Set
  value empty : Set
  function insert (x: Element, s: Set) : Set
  function delete (x: Element, s: Set) : Set
  function member (x: Element, s: Set) : Bool
endloc
endint
```

In this way, the type Element will be bound to those present in the current environment.

2. For modules, it is possible to constrain an exiting module by an interface, provided the module matches the interface, and the effect is to obtain a special view of the module where only components specified in the interface are visible. Consider for example the following specification:

```
interface VIEW1 is
  type S
  value x: S
  value y: S
endint
interface VIEW2 is
  type S
  type pairS is <x: S, y: S> endtype
endint
 module A is
  type S is ACK () | REQ () endtype
  value x: S is ACK ()
  value y: S is REQ ()
  type pairS is <x: S, y: S> endtype
endmod
module A1 : VIEW1 is A endmod
module A2 : VIEW2 is A endmod
```

As a result of constraint of A1 by VIEW1, only components specified in VIEW1 are accessible via A1. Hence neither A1.pair nor A1.ACK is accessible.

2.4 Composition of modules and interfaces

There are several means to compose modules and interfaces:

- Inclusion (or importation) of interfaces into interfaces using "**include** *int-id*$_1, \cdots,$ *int-id*$_n$"clause.
- Opening of modules in interfaces, modules, and generic modules using "**open** *mod-id*$_1, \cdots,$ *mod-id*$_n$".
- Renaming of interfaces and modules, whose the simpler form is:

  module mod-id' **is** mod-id (S is S', C is C', ...) **endmod**

- Instantiation of generic modules, whose the simpler form is:

  module mod-id **is** gen-id (mod-id' : int-exp) **endmod**

Interface inclusion Interfaces can include other interfaces. For example, the interface for pre-orders extends that for partial orders with one equation:

```
interface PreOrder is
  include 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
endint
interface PartialOrder is
  include PreOrder
  eqns forall x,y: T
    (x <= y andthen y <= x) => x = y ;
  endeqns
endint
```

If name clashed arise, the last name included is considered.

Module opening Interfaces can also open some module using the "**open**" construct. For example the monomorphic list interface is:

interface List **is**

open Monoid

type E

function inj (x: E) : M

endint

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

module NatMonoid : Monoid **is**

open Natural

type M **is** Nat **endtype**

endmod

Renaming Renaming is used to give an unique name to objects to avoid name clashes. The solution proposed in this paper is compatible with ACTONE solution for type renaming. For example, obtaining a module of integer lists with type ListNat is as follows:

module ListNat **is** GenericList (Natural) (List **is** ListNat) **endmod**

Instantiation Instantiation provide module generation by using already declared generic modules. Instantiation is more general than renaming, and one can view renaming a s an instantiation. For example, the previous example can be wrote as follows:

interface ListNat **is**

open NatMonoid

type ListNat

function inj (x: Nat) : ListNat

endint

generic RenameList (L : List) : ListNat **is**

open NatMonoid

type ListNat **is** L.M **endtype**

function inj (x: Nat) : ListNat **is** L.inj (x) **endfun**

endgen

module ListNat : ListNat **is** RenameList (GenericList(Natural)) **endmod**

2.5 Equational specifications

A decision taken at the Paris meeting was to allow equational specifications in interfaces, for example in the 'Monoid' interface:

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.

An extension of the solution proposed here is presented in [GM96]. It allows 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).

2.6 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 the base language: types, functions, processes, and modules may be external. For example, one could give an external implementation of the Monoid module by declaring:

module ExtMonoid : Monoid **is external endmod**

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

2.7 Compatibility with ACTONE

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

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:

interface Monoid **is**
  type M
  value 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

endint

module Monoid : Monoid **is external endmod**

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

1. ActOne allows overloading, as long as the sort of any expression can be determined statically,
2. 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**),
3. the relationship between generic modules and ActOne parameterized types and type renaming should be clarified.

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.

2.8 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 the relationship with them should be clarified.

For each module:

- We should give a signature, a module, and (where necessary) the dynamic semantics for the 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 will be the subject of further discussions.
- We should specify whether the module will be defined genericity.
- 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 present paper, the built-in data types and the rich term syntax are not described in detail. In the given examples, the implementation part are often omitted and only signatures (interfaces) are provided. A detailed specification should be provided maybe base on existing proposals, for example [GS96] [Pec94].

We propose the following *predefined types*: the type “int’, the type “real”, the type “bool”, the type “char”, the type “string”, the type “void”. All this types can be declared in a Standard module wich can be pervasive.

Also we propose a set of *constructed type* with their rich term syntax: enumerated types, subrange types, record types, sets, and lists.

2.9 Static semantics

Modules In static semantics, a module is a context. Modules can be declared using *module declarations*, whose simplest form is

module *mod-id* **is** *dec* **endmod**

where *mod-is* is a module identifier and *dec* is a (base language) declaration. The context associated with the module identifier *mod-id* is generated by the declarations in *dec*. For example, the declaration

module OrdItem **is**

type item **is** int

function leq (i: item, j: item) : bool **is** i<=j **endfun**

endmod

elaborates to the following environment

```
OrdItem -> { item => int,
             leq => [](i: item, j: item)[] -> exit (bool) }
```

In this example we suppose that integers are pervasive.

Interfaces Intuitively, an *interface* is a module type. Whereas a module expression declares a module, an interface expression specifies a module (or rather a class of modules). An interface is represented semantically by a context bound to the interface identifier. For example, the Monoid interface declaration below

interface Monoid **is**

type M

value 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

endint

elaborates to the following environment

```
Monoid -> { M => type,
            0 => M,
            + => [](M,M)[] -> exit (M) }
```

Over interfaces bindings and modules bindings we define some relations as enrichment, instantiation, and matching.

Intuitively, an interface binding $\mathcal{C}$ enriches another binding $\mathcal{C}'$, written $\mathcal{C} \succ \mathcal{C}'$, if all objects of $\mathcal{C}'$ are in $\mathcal{C}$with the same profile or types, and $\mathcal{C}$ contains other "specific" objects.

Intuitively, a module binding $\mathcal{C}^2$ is *an instance of* an interface binding $\mathcal{C}^1$, written $\mathcal{C}^1 \geq \mathcal{C}^2$, if the module provide an implementation of all objects declared into the interface with respect of sub-typing,

and only of this objects.

Intuitively, a module binding $\mathcal{C}^2$ *matches* an interface binding $\mathcal{C}^1$, if the module is an enrichment of another module with is an instance of the interface.

Generic modules The static semantics object corresponding to a generic module is an application from a record of module bindings to a module binding. For example, the result of elaborating a generic module declaration

generic *gen-id* (*mod-id* : *int-id*) **is** *dec* **endgen**

is

gen-id $\Rightarrow$ (*mod-id* : $\mathcal{C}$) $\rightarrow$ $\mathcal{C}$'

where $\mathcal{C}$ is the binding (type) of the interface identifier *int-id*, and $\mathcal{C}'$ is the result of elaborating *dec*.

The elaboration of a generic module instantiation consists to check first that the actual module parameters match the domain of the binding (here (*mod-id* : $\mathcal{C}$)), and then deriving the result from $\mathcal{C}'$, by instantiation of the formal parameters used in the generic module body.

Syntactic restrictions The main restrictions imposed by the static semantics are:

- The class of identifiers are disjoint, i.e., there are no clash between identifiers bound for modules, generic modules, and interfaces.
- No module binding, generic module binding, or interface binding may describe the same identifier twice.

Closure Restrictions The semantics presented requires no restrictions on reference to non-local identifiers. For example, it allows an interface expression to refer to external interface identifiers and to external module identifiers; it also allows a generic module and a renaming morphism to refer to external identifiers of any kinds.

However, for implementation purposes, one may want to impose the following restrictions on reference of identifiers (ignoring references to identifiers bound to the base environment, which may occur anywhere):

- In any interface binding *int-id* **is** *int-exp*, the only non-local references in *int-exp* are to interface identifiers (by "'**include**" clause). For example, in the following monomorphic list declaration:

  interface List **is**
  include Monoid
  type E
  function inj (x: E) : M
  endint

  the only one non-local reference is to interface identifier 'Monoid' and to its type 'M'.

- In any generic module binding *mod-id* (*mod-id* : *int-exp*)* : *int-exp'* **is** *mod-exp*, the only non-local references are to interface identifiers, except that *int-exp'* may refer to *mod-id* and its components. For example, in the following generic module declaration:

```
generic GenericList (E : EqType) : List is
  open E
  type M  is nil () | cons (x: E, s: M) endtype
  function + (s1: M, s2: M) : M  is
    case
      s1::nil () -> s2
    | s1::cons(x, xs) -> cons (x, + (xs, ys))
  endfun
  function inj (x : E) : M is
    cons (x, nil())
  endfun
endgen
```

the only non-local declarations are to interface identifier 'EqType' and to "parameter" identifier E.

3 Overview

3.1 Syntax

For module language there are further reserved words, terminals (identifier classes), non-terminals, and derived forms; comments and lexical analysis are the same as for the base language.

3.1.1 Reserved Words

The following are the additional reserved words used in Modules:

```
module      endmod   generic   endgen
interface   endint   include   open
behaviour   value    gate      exception
eqns        finite   private   subtypeof
```

3.1.2 Identifiers

Due to the dotted notation, the identifiers of the base language can be qualified by the name of the definition module. For each class X of identifiers, belonging to SCon, Var, Typ, Con, Proc, Fun, Gat, and Exc, there is a class longX of *long identifiers*; if X ranges over X then *longx* ranges over longX. The syntax of these long identifiers is given by the following:

The additional classes of *terminals* for Modules are:

identifier domain	*meaning*	*abbreviations*
ModId	module identifiers	*mod-id*
GenId	generics identifiers	*gen-id*
IntId	interface identifiers	*int-id*

As in the base language, we consider all identifier classes to be disjoint.

longx	::= *x*		*identifier*
	| *mod-id.x*		*qualified identifier*

3.1.3 Abstract Grammar of Modules

The additional classes of *non-terminals* for Modules are:

symbol domain	*meaning*	*abbreviations*
Spec	specifications	*spec*
TopDec	top-level declarations	*top-dec*
ModDec	module-level declaration	*mod-dec*
ModBind	module binding	*mod-bind*
ModExp	module expressions	*mod-exp*
RecModExp	record module expressions	*RME*
GenDec	generics declaration	*gen-dec*
GenBind	generics binding	*gen-bind*
IntDec	interface declarations	*int-dec*
IntBind	interface binding	*int-bind*
IntExp	interface expressions	*int-exp*
RecIntExp	record interface expressions	*RIE*
IntSpec	interface specification	*int-spec*
Attrib	type attribute declaration	*attrib*
EqnDec	equations declaration	*eqn-dec*

In the grammars, non-primitive constructs (which are defined in terms of syntactic sugar for primitive) are marked "$\star$". These grammars omit any **end**-keywords, which should be included in the concrete grammar.

Specification

spec	::= *top-dec* ; [*spec*]		*specification*	(spec1)

Top-level declaration

top-dec	::= *dec*		*declaration*	(TD1)
	| *mod-dec*		*module*	(TD2)
	| *gen-dec*		*generic module*	(TD3)
	| *int-dec*		*interface*	(TD4)

Declaration

dec	::= *D*		*base language declarations*	(dec1)

|	**gate** G $[(RT)]$ $(,\ G\ [(RT)])^*$	*gate declarations*	(dec2)
|	**exception** X $[(RT])$ $(,\ X\ [(RT)])^*$	*exception declaration*	(dec3)
|	**behaviour** BV **:** **exit** (RT) **is** B	*constant behaviour declaration*	(dec4)
|	**value** V **:** S **is** E	*constant value declaration*	(dec5)
|	**open** *mod-id*$_1 \cdots$ *mod-id*$_n$	*open declaration* ($n \geq 1$)	(dec6)
|		*empty declaration*	(dec7)
|	*dec*$_1$ *dec*$_2$	*sequential declaration*	(dec8)

Module declaration

mod-dec	::=	**module** *mod-bind*	*single*	(MD1)
	|		*empty*	(MD2)
	|	*mod-dec*$_1$ *mod-dec*$_2$	*sequential*	(MD3)

Module binding

mod-bind	::=	*mod-id* [**:** *int-exp*] **is** *mod-exp*	*module binding*	(MB1)
	|	*mod-id* **:** *int-exp* **is external**	*extern module*	(MB2)

Module expression

mod-exp	::=	*dec*	*block declaration*	(ME1)
	|	*mod-id*	*alias module*	(ME2)
	|	*gen-id* (RME)	*actualization*	(ME3)
	|	*mod-exp* $((S$ **is** $S' \mid C$ **is** $C' \mid \Pi$ **is** $\Pi' \mid V$ **is** $V' \mid$ BV **is** $BV' \mid G$ **is** $G' \mid X$ **is** $X')^*)$	*renaming*	(ME4)

Record module expression

RME	::=	*mod-id* $\Rightarrow$ *mod-id*[**:** *int-exp*]	*single*	(RME1)
	|		*empty*	(RME2)
	|	RME_1, RME_2	*disjoint union*	(RME3)
	$\star$ |	*mod-id*[**:** *int-exp*] $(,$ *mod-id*[**:** *int-exp*]$)^*$	*tuple*	(RME4)

Generic module declaration

gen-dec	::=	**generic** *gen-bind*	*single*	(MD1)
	|		*empty*	(MD2)
	|	*gen-dec*$_1$ *gen-dec*$_2$	*sequential*	(MD3)

Generic module binding

gen-bind	::=	*gen-id* (*RIE*)[: *int-exp*] **is** *mod-exp*	*generic module binding*	(MB1)

Interface declaration

int-dec	::=	**interface** *int-bind*	*single*	(ID1)
	|		*empty*	(ID2)
	|	$int\text{-}dec_1$ $int\text{-}dec_2$	*sequential*	(ID3)

Interface binding

int-bind	::=	*int-id* **is** *int-exp*	*interface binding*	(IB1)

Interface expressions

int-exp	::=	*int-spec*	*interface specification*	(IE1)
	|	*int-id*	*interface identifier*	(IE2)
	|	*int-exp* ((S **is** S' | C **is** C' | Π **is** Π' | V **is** V' | BV **is** BV' | G **is** G' | X **is** X')*)	*renaming*	(IE3)

Record interface expression

RIE	::=	*mod-id* : *int-exp*	*single*	(RIE1)
	|		*empty*	(RIE2)
	|	RIE_1, RIE_2	*disjoint union*	(RIE3)

Interface specification

int-spec	::=	**type** S [*attrib*]	*abstract data type*	(IS1)
	|	**type** S **is** C[(*RT*)] (|C[(*RT*)])*	*constructed type*	(IS2)
	|	**process** Π [[[G[(*RT*)](, G[(*RT*)])*]]][(*IP*)][: **exit** (*T*)] [**raises** [[[X[(*RT*)](, X[(*RT*)])*]]]]	*process*	(IS3)
	|	**gate** G[(*RT*)] (, G[(*RT*)])*	*gate*	(IS4)
	|	**exception** X[(*RT*)] (, X[(*RT*)])*	*exception*	(IS5)
	|	**behaviour** *BV* : **exit** (*RT*)	*constant behaviour*	(IS6)
	|	**value** V : S	*constant value*	(IS7)
	|	**eqns** *eqn-dec*	*equations*	(IS8)
	|	**local** $int\text{-}exp_1$ **in** $int\text{-}exp_2$	*local*	(IS9)
	|	**open** $mod\text{-}id_1 \cdots mod\text{-}id_n$	*open* $n \geq 1$	(IS10)

	include $int\text{-}id_1 \cdots int\text{-}id_n$	$include\ n \geq 1$	(IS11)
|		*empty*	(IS12)
|	$int\text{-}spec_1 \quad int\text{-}spec_2$	*sequential*	(IS13)

Abstract type attribute

attrib	::=	**any**	*by default abstract type*	(A1)
	|	**finite**	*finite*	(A2)
	|	**private**	*with = and :=*	(A3)
	|	**subtypeof** $C[(RT)]$ $(\mid C[(RT)])^*$ [| **any**]	*subtype*	(A4)

Equation declaration

eqn-dec	::=	**forall** $RT \ \rightarrow \ E \ (;\ E)^*$ [**and** *eqn-dec*]	(EQ1)

3.1.4 Derived Forms

Module expressions

$gen\text{-}id(dec) \quad \equiv_{def} \quad gen\text{-}id(\$1 \Rightarrow dec)$

Generic module binding

$$\left(\begin{array}{l} gen\text{-}id(int\text{-}spec) \\ \quad [: int\text{-}exp] \textbf{ is } mod\text{-}exp \end{array} \right) \quad =_{def} \quad \left(\begin{array}{l} gen\text{-}id(mod\text{-}id \ : \ int\text{-}spec) \\ \quad [: int\text{-}exp] \textbf{ is open } mod\text{-}id\, mod\text{-}exp \end{array} \right)$$

3.2 Static Semantics for Modules

3.2.1 Semantic objects

The context for the base language (see [Que96b, Section 2.2]) is enriched with the following bindings for behaviour values:

$\mathcal{C}$	::=	base language context	
	|	BV $\Rightarrow$ **exit** (RT)	*behaviour constant*(C_c10)

For modules we define another context which use those of the base language, and it is defined by the following grammar:

$\mathcal{B}$	::=	$\mathcal{C}$	*context of the base language*	($\mathcal{B}$1)
	|	$mod\text{-}id \Rightarrow \mathcal{C}$	*module*	($\mathcal{B}$2)
	|	$gen\text{-}id \Rightarrow (mod\text{-}id \Rightarrow \mathcal{C})^* \rightarrow \mathcal{C}$	*generics*	($\mathcal{B}$3)

$\mid$ *int-id* $\Rightarrow \mathcal{C}$	*interface*	($\mathcal{B}4$)
$\mid$	*empty*	($\mathcal{B}5$)
$\mid \mathcal{B},\ \mathcal{B}$	*disjoint union*	($\mathcal{B}6$)
		($\mathcal{B}7$)

The difference between module bindings and generic modules co-domain context, and interface bindings are the presence of abstract data types, as detailed below in the Section 3.2.3.

The operations on contexts and module types are similar to those defined in [Que96b, Section 2.2]: "," is the disjunct composition, "+" context over-riding, "$\ominus$" is the domain restriction, and "$\odot$" is the match composition (see definitions below).

> When A and B are sets, $\mathrm{Fin}(A)$ denotes the set of finite subsets of A, and $A \xrightarrow{\text{fin}} B$ 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.
>
> - $f + g$ with domain $\mathrm{Dom}(f) \cup \mathrm{Dom}(g)$ and values:
>
> $$(f+g)(a) = \begin{cases} f(a) & \textbf{if } a \notin \mathrm{Dom}(g) \\ g(a) & \textbf{otherwise} \end{cases}$$
>
> This operator is called f *modified by* (or *overridden*) g.

3.2.2 Cycle freedom

In our semantic definition we have not cycle of modules names; that is, there is no sequence

$$\textit{mod-id}_0 \rightarrow \cdots \rightarrow \textit{mod-id}_k = \textit{mod-id}_0 \quad (k > 0)$$

of modules names, where the " $\rightarrow$ " relation is the use relation by "**open**" construct.

3.2.3 Static Sub-typing

A concrete data type S is *an instance (subtype) of* an abstract one S', written $S \sqsubseteq S'$, if:

1. S' is declared as "**any**" type.
2. If S' is declared as a "**finite**" type, and S is finite.
3. If S' is declared as a "**private**" type, and S admits equality.
4. If S' is declared as a constructed type with $Con' = \{C' \Rightarrow (RT) \rightarrow S'\}$ the set of constructors, Con is the set of constructors of S, and Con is an instance of Con', i.e.,

$$(\forall C'\ (RT') \in Con')(\exists C\ (RT) \in Con)\ s.t.\ (C' = C)\ \ \wedge\ \ (RT \sqsubseteq RT')$$

Th type sub-typing can be extended to constructors, process, gate, exception, value variable, and behaviour variables in a similar manner, as follows:

A value variable context $V \Rightarrow S$ is an instance of another value variable declaration $V' \Rightarrow S'$, written $V \Rightarrow S \sqsubseteq V' \Rightarrow S'$, if:

1. $V = V'$
2. $S \sqsubseteq S'$

Similar for other declarations.

3.2.4 Interface Instantiation

A context can be seen as a tuple $(\mathcal{C}_S, \mathcal{C}_C, \mathcal{C}_\Pi, \mathcal{C}_G, \mathcal{C}_X, \mathcal{C}_V, \mathcal{C}_{BV})$ of contexts for sorts, constructors, processes, gates, exceptions, value variables, and behaviour variables.

In this case, a module binding $\mathcal{C}^2$ is *an instance of* an interface binding $\mathcal{C}^1$, written $\mathcal{C}^1 \geq \mathcal{C}^2$, if

1. $\mathrm{Dom}(\mathcal{C}^1_S) = \mathrm{Dom}(\mathcal{C}^2_S)$, and $\mathcal{C}^1(S) \sqsubseteq \mathcal{C}^2(S)$ for all $S \in \mathrm{Dom}(\mathcal{C}^2_S)$
2. $\mathrm{Dom}(\mathcal{C}^1_C) = \mathrm{Dom}(\mathcal{C}^2_C)$, and $\mathcal{C}^1(C) \sqsubseteq \mathcal{C}^2(C)$ for all $C \in \mathrm{Dom}(\mathcal{C}^2_C)$
3. $\mathrm{Dom}(\mathcal{C}^1_\Pi) = \mathrm{Dom}(\mathcal{C}^2_\Pi)$, and $\mathcal{C}^1(\Pi) \sqsubseteq \mathcal{C}^2(\Pi)$ for all $\Pi \in \mathrm{Dom}(\mathcal{C}^2_\Pi)$
4. $\mathrm{Dom}(\mathcal{C}^1_G) = \mathrm{Dom}(\mathcal{C}^2_G)$, and $\mathcal{C}^1(G) \sqsubseteq \mathcal{C}^2(G)$ for all $G \in \mathrm{Dom}(\mathcal{C}^2_G)$
5. $\mathrm{Dom}(\mathcal{C}^1_X) = \mathrm{Dom}(\mathcal{C}^2_X)$, and $\mathcal{C}^1(X) \sqsubseteq \mathcal{C}^2(X)$ for all $X \in \mathrm{Dom}(\mathcal{C}^2_X)$
6. $\mathrm{Dom}(\mathcal{C}^1_V) = \mathrm{Dom}(\mathcal{C}^2_V)$, and $\mathcal{C}^1(V) \sqsubseteq \mathcal{C}^2(V)$ for all $V \in \mathrm{Dom}(\mathcal{C}^2_V)$
7. $\mathrm{Dom}(\mathcal{C}^1_{BV}) = \mathrm{Dom}(\mathcal{C}^2_{BV})$, and $\mathcal{C}^1(BV) \sqsubseteq \mathcal{C}^2(BV)$ for all $BV \in \mathrm{Dom}(\mathcal{C}^2_{BV})$

Intuitively, the instantiation is substitution of the abstract types of the interfaces with concrete types of modules with respect to sort sub-typing.

The instantiation of a interface binding by another interface binding can be extended naturally from the relation above by allowing sub-typing between constructed abstract data types.

3.2.5 Enrichment

A context $\mathcal{C}^1$ *enriches* another context $\mathcal{C}^2$, written $\mathcal{C}^1 \succ \mathcal{C}^2$, if

1. $\mathrm{Dom}(\mathcal{C}^1_S) \supseteq \mathrm{Dom}(\mathcal{C}^2_S)$, and $\mathcal{C}^1(S) = \mathcal{C}^2(S)$ for all $S \in \mathrm{Dom}(\mathcal{C}^2_S)$
2. $\mathrm{Dom}(\mathcal{C}^1_C) \supseteq \mathrm{Dom}(\mathcal{C}^2_C)$, and $\mathcal{C}^1(C) = \mathcal{C}^2(C)$ for all $C \in \mathrm{Dom}(\mathcal{C}^2_C)$
3. $\mathrm{Dom}(\mathcal{C}^1_\Pi) \supseteq \mathrm{Dom}(\mathcal{C}^2_\Pi)$, and $\mathcal{C}^1(\Pi) = \mathcal{C}^2(\Pi)$ for all $\Pi \in \mathrm{Dom}(\mathcal{C}^2_\Pi)$
4. $\mathrm{Dom}(\mathcal{C}^1_G) \supseteq \mathrm{Dom}(\mathcal{C}^2_G)$, and $\mathcal{C}^1(G) = \mathcal{C}^2(G)$ for all $G \in \mathrm{Dom}(\mathcal{C}^2_G)$
5. $\mathrm{Dom}(\mathcal{C}^1_X) \supseteq \mathrm{Dom}(\mathcal{C}^2_X)$, and $\mathcal{C}^1(X) = \mathcal{C}^2(X)$ for all $X \in \mathrm{Dom}(\mathcal{C}^2_X)$
6. $\mathrm{Dom}(\mathcal{C}^1_V) \supseteq \mathrm{Dom}(\mathcal{C}^2_V)$, and $\mathcal{C}^1(V) = \mathcal{C}^2(V)$ for all $V \in \mathrm{Dom}(\mathcal{C}^2_V)$
7. $\mathrm{Dom}(\mathcal{C}^1_{BV}) \supseteq \mathrm{Dom}(\mathcal{C}^2_{BV})$, and $\mathcal{C}^1(BV) = \mathcal{C}^2(BV)$ for all $BV \in \mathrm{Dom}(\mathcal{C}^2_{BV})$

3.2.6 Interface Matching

A module binding $\mathcal{C}_2$ *matches* an interface binding $\mathcal{C}_1$ if there exists a module binding $\mathcal{C}'$ such that $\mathcal{C}_1 \geq \mathcal{C}' \prec \mathcal{C}_2$. This matching is a combination of instantiation and enrichment.

An interface binding $\mathcal{C}_2$ *matches* another interface binding $\mathcal{C}_1$, written $\mathcal{C}_2 \succ \mathcal{C}_1$, if for all module binding $\mathcal{C}$, if $\mathcal{C}$ matches $\mathcal{C}_2$ then $\mathcal{C}$ matches $\mathcal{C}_1$.

3.2.7 Inference rules

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*
$\mathcal{B} \vdash$ *spec* $\Rightarrow \mathcal{B}'$	The specification *spec* is well formed and gives the context $\mathcal{B}'$.
$\mathcal{B} \vdash$ *top-dec* $\Rightarrow \mathcal{B}'$	The top-level declaration *top-dec* is well formed and gives the context $\mathcal{B}'$.
$\mathcal{B} \vdash$ *dec* $\Rightarrow \mathcal{C}$	The declaration *dec* is well formed and gives the context $\mathcal{C}$.
$\mathcal{B} \vdash$ *mod-dec* $\Rightarrow \mathcal{B}'$	The module declaration *mod-dec* is well formed and gives the context $\mathcal{B}'$.
$\mathcal{B} \vdash$ *mod-bind* $\Rightarrow \mathcal{B}'$	The module binding *mod-bind* is well formed and and gives the context $\mathcal{B}'$.
$\mathcal{B} \vdash$ *mod-exp* $\Rightarrow \mathcal{C}$	The module expression *mod-exp* is well formed and has the type $\mathcal{C}$.
$\mathcal{B} \vdash$ *RME* $\Rightarrow \mathcal{B}'$	The record module expression *RME* is well formed and gives the context $\mathcal{B}'$.
$\mathcal{B} \vdash$ *gen-dec* $\Rightarrow \mathcal{B}'$	The generic module declaration *gen-dec* is well formed and gives the context $\mathcal{B}'$.
$\mathcal{B} \vdash$ *gen-bind* $\Rightarrow \mathcal{B}'$	The generic module binding *gen-bind* is well formed and has gives the context $\mathcal{B}'$.
$\mathcal{B} \vdash$ *int-dec* $\Rightarrow \mathcal{B}'$	The interface declaration *int-dec* is well formed and gives the context $\mathcal{B}'$.
$\mathcal{B} \vdash$ *int-bind* $\Rightarrow \mathcal{B}'$	The interface block *int-bind* is well formed and gives the context $\mathcal{B}'$.
$\mathcal{B} \vdash$ *int-exp* $\Rightarrow \mathcal{C}$	The interface expression *int-exp* is well formed and has the type $\mathcal{C}$.
$\mathcal{B} \vdash$ *RIE* $\Rightarrow \mathcal{B}'$	The record interface expression *RIE* is well formed and gives the context $\mathcal{B}'$.
$\mathcal{B} \vdash$ *int-spec* $\Rightarrow \mathcal{C}$	The interface specification *int-spec* is well formed and has the type $\mathcal{C}$.
$\mathcal{B} \vdash$ *eqns-dec* $\Rightarrow$ `ok`	The equation description *eqns-dec* is well formed.

4 Specification

4.1 Rationale

An E-LOTOS specification is a sequence of top level declarations. The result of a such specification is the evaluation of an entry point specified by the user, after the elaboration of the top-level declarations.

4.2 Syntax

spec ::= *top-dec* ; [*spec*] *specification* (spec1)

4.3 Static semantics

$\boxed{\mathcal{B} \vdash spec \Rightarrow \mathcal{B}'}$

$$\frac{\mathcal{B} \vdash top\text{-}dec \Rightarrow \mathcal{B}' \quad \mathcal{B} \vdash spec \Rightarrow \mathcal{B}''}{\mathcal{B} \vdash top\text{-}dec \ ; \ spec \Rightarrow \mathcal{B}' + \mathcal{B}''}$$

5 Top-level declaration

5.1 Rationale

The top-level declarations may be any declaration of the base language (enriched with the declaration of gates, exceptions, constant values and constant behaviours), any sequence (maybe empty) of module declarations, generic module declarations and interfaces declarations.

5.2 Syntax

top-dec	::=	*dec*	*declaration*	(TD1)
	|	*mod-dec*	*module*	(TD2)
	|	*gen-dec*	*generic module*	(TD3)
	|	*int-dec*	*interface*	(TD4)

5.3 Static semantics

$\boxed{\mathcal{B} \vdash top\text{-}dec \Rightarrow \mathcal{B}'}$

$$\frac{\mathcal{B} \vdash dec \Rightarrow \mathcal{B}'}{\mathcal{B} \vdash top\text{-}dec \Rightarrow \mathcal{B}'}$$

$$\frac{\mathcal{B} \vdash mod\text{-}dec \Rightarrow \mathcal{B}'}{\mathcal{B} \vdash top\text{-}dec \Rightarrow \mathcal{B}'}$$

$$\frac{\mathcal{B} \vdash gen\text{-}dec \Rightarrow \mathcal{B}'}{\mathcal{B} \vdash top\text{-}dec \Rightarrow \mathcal{B}'}$$

$$\frac{\mathcal{B} \vdash gen\text{-}dec \Rightarrow \mathcal{B}'}{\mathcal{B} \vdash top\text{-}dec \Rightarrow \mathcal{B}'}$$

6 Declaration

6.1 Rationale

The declarations of the base language are enriched with declarations for gates, exceptions, behaviour constants, value constants, and open clauses. Those declarations may appear at the top-level or in the module expressions, as a *generative* module expression (i.e., an expression which creates objects explicitly, by declaration).

The list of gate and exceptions declaration may be seen as a global declaration for a specification or for a module. Their visibility is the all specification and the all module, except subsequent re-declarations.

6.2 Syntax

dec	::=	D	*base language declarations*	(dec1)
	|	**gate** G $[(RT)]$ $(,\ G\ [(RT)])^*$	*gate declarations*	(dec2)
	|	**exception** X $[(RT])$ $(,\ X\ [(RT)])^*$	*exception declaration*	(dec3)
	|	**behaviour** BV **: exit** (RT) **is** B	*constant behaviour declaration*	(dec4)
	|	**value** V **:** S **is** E	*constant value declaration*	(dec5)
	|	**open** *mod-id*$_1 \cdots$ *mod-id*$_n$	*open declaration* $(n \geq 1)$	(dec6)
	|		*empty declaration*	(dec7)
	|	$dec_1 \quad dec_2$	*sequential declaration*	(dec8)

6.3 Static semantics

$\boxed{\mathcal{B} \vdash dec \Rightarrow \mathcal{C}}$

$$\frac{\mathcal{B} \vdash D \Rightarrow \mathcal{C}}{\mathcal{B} \vdash D \Rightarrow \mathcal{C}}$$

Comment: We suppose provided the above rule by the base language static semantics.

$$\frac{\mathcal{B} \vdash RT_1 \Rightarrow \mathbf{type} \quad \cdots \quad \mathcal{B} \vdash RT_n \Rightarrow \mathbf{type}}{\mathcal{B} \vdash \mathbf{gate}\ G_1\ (RT_1), \cdots, G_n\ (RT_n) \Rightarrow G_1 \Rightarrow \mathbf{gate}(RT_1), \cdots, G_n \Rightarrow \mathbf{gate}(RT_n)}$$

$$\frac{\mathcal{B} \vdash RT_1 \Rightarrow \mathbf{type} \quad \cdots \quad \mathcal{B} \vdash RT_n \Rightarrow \mathbf{type}}{\mathcal{B} \vdash \mathbf{exception}\ X_1\ (RT_1), \cdots, X_n\ (RT_n) \Rightarrow X_1 \Rightarrow \mathbf{exn}(RT_1), \cdots, X_n \Rightarrow \mathbf{exn}(RT_n)}$$

$$\frac{\mathcal{B} \vdash B \Rightarrow \mathbf{exit}(RT)}{\mathcal{B} \vdash \mathbf{behaviour}\ BV\ \mathbf{:}\ \mathbf{exit}\ (RT)\ \mathbf{is}\ B \Rightarrow BV \Rightarrow \mathbf{exit}\ (RT)}$$

$$\frac{\mathcal{B} \vdash E \Rightarrow \mathbf{exit}(S') \qquad S \sqsubseteq S'}{\mathcal{B} \vdash \mathbf{value}\ V\ \mathbf{:}\ S\ \mathbf{is}\ E \Rightarrow V \Rightarrow S}$$

$$\frac{\mathcal{B} \vdash \textit{mod-id}_1 \Rightarrow \mathcal{C}_1 \quad \cdots \quad \mathcal{B} \vdash \textit{mod-id}_n \Rightarrow \mathcal{C}_n}{\mathcal{B} \vdash \mathbf{open}\ \textit{mod-id}_1 \cdots \textit{mod-id}_n \Rightarrow \mathcal{C}_1 + \cdots + \mathcal{C}_n}$$

$$\frac{}{\mathcal{B} \vdash \Rightarrow \{\}}$$

$$\frac{\mathcal{B} \vdash dec_1 \Rightarrow \mathcal{C}_1 \qquad \mathcal{B} \vdash dec_2 \Rightarrow \mathcal{C}_2}{\mathcal{B} \vdash dec_1 \quad dec_2 \Rightarrow \mathcal{C}_1 + \mathcal{C}_2}$$

7 Module declaration

7.1 Rationale

Modules describe implementations of types, processes, functions. They can be constrained at a given view, or to have a by default view. Module declarations could be grouped or interfered with interface and generic module declarations.

7.2 Syntax

mod-dec	::=	**module** *mod-bind*	*single*	(MD1)
	|		*empty*	(MD2)
	|	$mod\text{-}dec_1$ $mod\text{-}dec_2$	*sequential*	(MD3)

7.3 Static semantics

$$\boxed{\mathcal{B} \vdash mod\text{-}dec \Rightarrow \mathcal{B}'}$$

$$\frac{\mathcal{B} \vdash mod\text{-}bind \Rightarrow \mathcal{B}'}{\mathcal{B} \vdash \textbf{module}\ mod\text{-}bind \Rightarrow \mathcal{B}'}$$

$$\frac{}{\mathcal{B} \vdash \Rightarrow \{\}}$$

$$\frac{\mathcal{B} \vdash mod\text{-}dec_1 \Rightarrow \mathcal{B}_1 \quad \cdots \quad \mathcal{B} \vdash mod\text{-}dec_2 \Rightarrow \mathcal{B}_2}{\mathcal{B} \vdash mod\text{-}dec_1 \quad mod\text{-}dec_2 \Rightarrow \mathcal{B}_1, \mathcal{B}_2}$$

Comment: The "," composition of contexts ensures that the re-declaration of a module will raise an error.

8 Module binding

8.1 Rationale

A module binding allow to associate to a given module identifier the set of objects that it implements. There are two forms of bindings: either specified using the base language, or an external specification. The dependent module bindings cannot be grouped as in SML. We note that cycle freedom constraint forbid the recursive declarations of modules.

8.2 Syntax

mod-bind	::=	*mod-id* [: *int-exp*] **is** *mod-exp*	*module binding*	(MB1)
	|	*mod-id* : *int-exp* **is external**	*extern module*	(MB2)

8.3 Static semantics

$\boxed{\mathcal{B} \vdash \textit{mod-bind} \Rightarrow \mathcal{B}}$

$$\frac{\begin{array}{lll}\mathcal{B} & \vdash & \textit{mod-exp} \Rightarrow \mathcal{C} \\ \langle\mathcal{B} & \vdash & \textit{int-exp} \Rightarrow \mathcal{C}_1,\ \mathcal{C}_1 \geq \mathcal{C}' \prec \mathcal{C}\rangle\end{array}}{\begin{array}{lll}\mathcal{B} & \vdash & \textit{mod-id}\ \langle : \textit{int-exp}\rangle\ \textbf{is}\ \textit{mod-exp} \Rightarrow \textit{mod-id} \Rightarrow \mathcal{C}\langle'\rangle\end{array}}$$

Comment: If present, *int-exp* has the effect of restricting the view which *mod-id* provides by *mod-exp*. If not present, the context provided is $\mathcal{C}$.

$$\frac{\begin{array}{lll}\mathcal{B} & \vdash & \textit{int-exp} \Rightarrow \mathcal{C}\end{array}}{\begin{array}{lll}\mathcal{B} & \vdash & \textit{mod-id} : \textit{int-exp}\ \textbf{is external} \Rightarrow \textit{mod-id} \Rightarrow \mathcal{C}\end{array}}$$

Comment: The presence of *int-exp* is compulsory because no other informations are provided for the module.

9 Module expression

9.1 Rationale

Module expressions describe the body of a module. They can be an explicit (generative) declaration, an aliasing to an (already declared) module, an instantiation of a given module, or an renaming. Renaming can be applied to a module expression.

9.2 Syntax

mod-exp	::=	*dec*	*block declaration*	(ME1)
	|	*mod-id*	*alias module*	(ME2)
	|	*gen-id* (*RME*)	*actualization*	(ME3)
	|	*mod-exp* $((S$ **is** $S' \mid C$ **is** $C' \mid \Pi$ **is** $\Pi' \mid V$ **is** $V' \mid$ BV **is** $BV' \mid G$ **is** $G' \mid X$ **is** $X')^*)$	*renaming*	(ME4)

9.3 Static semantics

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

$$\frac{\begin{array}{lll}\mathcal{B} & \vdash & \textit{dec} \Rightarrow \mathcal{C}\end{array}}{\begin{array}{lll}\mathcal{B} & \vdash & \textit{dec} \Rightarrow \mathcal{C}\end{array}}$$

$$\frac{\begin{array}{lll}\mathcal{B} & \vdash & \textit{mod-id} \Rightarrow \mathcal{C}\end{array}}{\begin{array}{lll}\mathcal{B} & \vdash & \textit{mod-id} \Rightarrow \mathcal{C}\end{array}}$$

Comment: The view of the alias module can be restricted by specifying an interface for the aliasing module.

$$\frac{\begin{array}{lll}\mathcal{B} & \vdash & \textit{RME} \Rightarrow \mathcal{B} \\ \mathcal{B} & \vdash & \textit{gen-id} \Rightarrow (\textit{mod-id} \Rightarrow \mathcal{C})^* \rightarrow \mathcal{C} \\ & & \mathcal{B} \succ (\textit{mod-id} \Rightarrow \mathcal{C})^*\end{array}}{\begin{array}{lll}\mathcal{B} & \vdash & \textit{gen-id}\ (\textit{RME}) \Rightarrow \mathcal{C}\end{array}}$$

Comment: The formal parameters of the generic module $(\textit{mod-id} \Rightarrow \mathcal{C})^*$ are a context of module bindings. The actual parameters *RME* are elaborated to a collection of module bindings, $\mathcal{B}$. The side condition ensures that the actual module bindings enrich the formal module bindings (see definition in Section 3.2.5).

$$\frac{\begin{array}{ll}\mathcal{B} & \vdash \quad \textit{mod-exp} \Rightarrow \mathcal{C} \\ & \mathcal{C}' = \sigma(\mathcal{C}) \quad \sigma = (S \textbf{ is } S' \mid C \textbf{ is } C' \mid \Pi \textbf{ is } \Pi' \mid V \textbf{ is } V' \mid BV \textbf{ is } BV' \mid G \textbf{ is } G' \mid X \textbf{ is } X')^*\end{array}}{\mathcal{B} \quad \vdash \quad \textit{mod-exp } ((S \textbf{ is } S' \mid C \textbf{ is } C' \mid \Pi \textbf{ is } \Pi' \mid V \textbf{ is } V' \mid BV \textbf{ is } BV' \mid G \textbf{ is } G' \mid X \textbf{ is } X')^*) \Rightarrow \mathcal{C}'}$$

Comment: The renaming generates a substitution morphism, σ; it is applied to the context associated with the binding of *mod-id*.

10 Record module expression

10.1 Rationale

the Record module expression are the actual parameters of the generic module instantiation. There are two forms of actual parameters: records specify explicitly the formal parameter name, the order of specification of the actual parameter is not fixed; tuples are based on the declaration order of the generic module parameters. These two form cannot be merged.

10.2 Syntax

RME	::=	*mod-id* $\Rightarrow$ *mod-id*[: *int-exp*]	*single*	(RME1)
	|		*empty*	(RME2)
	|	RME_1, RME_2	*disjoint union*	(RME3)
	|	*mod-id*[: *int-exp*] (, *mod-id*[: *int-exp*])*	*tuple*	(RME4)

10.3 Static semantics

$$\boxed{\mathcal{B} \vdash \textit{RME} \Rightarrow \mathcal{B}'}$$

$$\frac{\begin{array}{ll}\mathcal{B} & \vdash \quad \textit{mod-id}_2 \Rightarrow \mathcal{C}_2 \\ \langle\mathcal{B} & \vdash \quad \textit{int-exp}_2 \Rightarrow \mathcal{C}_2' \quad \mathcal{C}_2' \geq \mathcal{C}_2\rangle\end{array}}{\mathcal{B} \quad \vdash \quad \textit{mod-id}_1 \Rightarrow \textit{mod-id}_2\langle\textbf{:}\ \textit{int-exp}_2\rangle \Rightarrow \textit{mod-id}_1 \Rightarrow \mathcal{C}_2\langle'\rangle}$$

Comment: If present, $\textit{int-exp}_2$ has the effect to restrict the view which the actual module parameter $\textit{mod-id}_2$ provides. The side condition ensure that the view $\mathcal{C}_2'$ is consistent with the module by default view $\mathcal{C}_2$. The notation $\mathcal{C}_2\langle'\rangle$ means $\mathcal{C}_2'$ if $\textit{int-exp}_2$ is present, and $\mathcal{C}_2$ if not.

$$\frac{}{\mathcal{B} \quad \vdash \quad \Rightarrow \{\}}$$

$$\frac{\begin{array}{ll}\mathcal{B} & \vdash \quad RME_1 \Rightarrow \mathcal{B}_1 \\ \mathcal{B} & \vdash \quad RME_2 \Rightarrow \mathcal{B}_2\end{array}}{\mathcal{B} \quad \vdash \quad RME_1, RME_2 \Rightarrow \mathcal{B}_1, \mathcal{B}_2}$$

Comments:

- It is worth noticing that the composition of the contexts resulting from the elaboration of records is a disjoint one. So we allow formal parameters to be actualized only one time.
- The tuple of module expressions is a syntactic sugar for the records, as for the base language tuples. $mod\text{-}id_1[: int\text{-}exp_1], \cdots, mod\text{-}id_n[: int\text{-}exp_n] \quad =_{def} \quad \$1 \Rightarrow mod\text{-}id_1[: int\text{-}exp_1], \cdots, \$n \Rightarrow mod\text{-}id_n[: int\text{-}exp_n$

11 Generic module declaration

11.1 Rationale

Generic modules provide means for code reuse. They describe implementations of types, processes, functions parameterized by other types, functions, processes. They can be constrained at a given view, or to have a by default view. Generic module declarations could be grouped or interfered with interface and module declarations.

11.2 Syntax

gen-dec	::=	**generic** *gen-bind*	*single*	(MD1)
	|		*empty*	(MD2)
	|	$gen\text{-}dec_1 \quad gen\text{-}dec_2$	*sequential*	(MD3)

11.3 Static semantics

$\boxed{\mathcal{B} \vdash gen\text{-}dec \Rightarrow \mathcal{B}'}$

$$\frac{\mathcal{B} \vdash gen\text{-}bind \Rightarrow \mathcal{B}'}{\mathcal{B} \vdash \textbf{generic } gen\text{-}bind \Rightarrow \mathcal{B}'}$$

$$\frac{}{\mathcal{B} \vdash \Rightarrow \{\}}$$

$$\frac{\mathcal{B} \vdash gen\text{-}dec_1 \Rightarrow \mathcal{B}_1 \quad \cdots \quad \mathcal{B} \vdash gen\text{-}dec_2 \Rightarrow \mathcal{B}_2}{\mathcal{B} \vdash gen\text{-}dec_1 \quad gen\text{-}dec_2 \Rightarrow \mathcal{B}_1, \mathcal{B}_2}$$

Comment: The "," composition of contexts ensures that the re-declaration of a generic module will raise an error.

12 Generic Module Binding

12.1 Rationale

A generic module binding allow to associate to a given generic module identifier the set of objects that it implements. There is only one form of binding: the body of the generic module can be declared only using the base language. There are no external generic modules. The dependent generic module bindings cannot be grouped as in SML. Generic modules can be parameterized by modules interfaces. However, generic modules cannot appear as parameters of generic modules.

12.2 Syntax

gen-bind ::= *gen-id* (*RIE*)[: *int-exp*] **is** *mod-exp* — *generic module binding* (MB1)

12.3 Static semantics

$\boxed{\mathcal{B} \vdash \textit{gen-bind} \Rightarrow \mathcal{B}'}$

$$\frac{\begin{array}{rcl} \mathcal{B} & \vdash & \textit{RIE} \Rightarrow \mathcal{B}' \\ \mathcal{B}+\mathcal{B}' & \vdash & \textit{mod-exp} \Rightarrow \mathcal{C} \\ \langle \mathcal{B}+\mathcal{B}' & \vdash & \textit{int-exp} \Rightarrow \mathcal{C}' \qquad \mathcal{C}' \geq \mathcal{C}'' \prec \mathcal{C} \rangle \end{array}}{\begin{array}{rcl} \mathcal{B} & \vdash & \textit{gen-id}\ (\textit{RIE})\langle : \textit{int-exp} \rangle\ \textbf{is}\ \textit{mod-exp} \Rightarrow \textit{gen-id} \Rightarrow \mathcal{B}' \rightarrow\ \mathcal{C}'\langle' \rangle \end{array}}$$

Comment: The elaboration of the body of the generic module *mod-exp* is made in presence of the formal parameters context $\mathcal{B}'$. Similarly for the view of the generic module *int-exp*, if it exists. The side condition ensures that the view is matched by the body of the generic module.

13 Interface declaration

13.1 Rationale

Interfaces describe external views for types, processes, functions. OInly types can have an implementation specification into interfaces. Interface declarations could be grouped or interfered with modules and generic module declarations.

13.2 Syntax

int-dec ::= **interface** *int-bind* — *single* (ID1)

| — *empty* (ID2)

| *int-dec*$_1$ *int-dec*$_2$ — *sequential* (ID3)

13.3 Static semantics

$\boxed{\mathcal{B} \vdash \textit{int-dec} \Rightarrow \mathcal{B}'}$

$$\frac{\mathcal{B} \vdash \textit{int-bind} \Rightarrow \mathcal{B}'}{\mathcal{B} \vdash \textbf{interface}\ \textit{int-bind} \Rightarrow \mathcal{B}'}$$

$$\frac{}{\mathcal{B} \vdash \Rightarrow \{\}}$$

$$\frac{\mathcal{B} \vdash \textit{int-dec}_1 \Rightarrow \mathcal{B}_1 \qquad \mathcal{B} \vdash \textit{int-dec}_2 \Rightarrow \mathcal{B}_2}{\mathcal{B} \vdash \textit{int-dec}_1\ \ \textit{int-dec}_2 \Rightarrow \mathcal{B}_1, \mathcal{B}_2}$$

Comment: The "," composition of contexts ensures that the re-declaration of an interface will raise an error.

14 Interface binding

14.1 Rationale

An interface binding allow to associate to a given interface identifier the set of objects that it specifies. No external interfaces can be specified. The dependent module bindings cannot be grouped as in SML. We note that cycle freedom constraint forbid the recursive declarations of modules.

14.2 Syntax

int-bind	::=	*int-id* **is** *int-exp*	*interface binding*	(IB1)

14.3 Static semantics

$\boxed{\mathcal{B} \vdash \textit{int-bind} \Rightarrow \mathcal{B}'}$

$$\frac{\mathcal{B} \vdash \textit{int-exp} \Rightarrow \mathcal{C}}{\mathcal{B} \vdash \textit{int-id}\ \textbf{is}\ \textit{int-exp} \Rightarrow \textit{int-id} \Rightarrow \mathcal{C}}$$

15 Interface expressions

15.1 Rationale

Interface expressions describe the body of an interface. They can be an explicit (generative) declaration, an aliasing to an (already declared) interface, or an renaming. Renaming can be applied to an interface expression.

15.2 Syntax

int-exp	::=	*int-spec*	*interface specification*	(IE1)
	|	*int-id*	*interface identifier*	(IE2)
	|	*int-exp* $((S\ \textbf{is}\ S' \mid C\ \textbf{is}\ C' \mid \Pi\ \textbf{is}\ \Pi' \mid V\ \textbf{is}\ V' \mid$ $BV\ \textbf{is}\ BV' \mid G\ \textbf{is}\ G' \mid X\ \textbf{is}\ X')^*)$	*renaming*	(IE3)

15.3 Static semantics

$\boxed{\mathcal{B} \vdash \textit{int-exp} \Rightarrow \mathcal{C}}$

$$\frac{\mathcal{B} \vdash \textit{int-spec} \Rightarrow \mathcal{C}}{\mathcal{B} \vdash \textit{int-spec} \Rightarrow \mathcal{C}}$$

$$\frac{\mathcal{B} \vdash \textit{int-id} \Rightarrow \mathcal{C}}{\mathcal{B} \vdash \textit{int-id} \Rightarrow \mathcal{C}}$$

$$\frac{\mathcal{B} \vdash \textit{int-exp} \Rightarrow \mathcal{C} \quad \mathcal{C}' = \sigma(\mathcal{C}) \quad \sigma = (S \textbf{ is } S' \mid C \textbf{ is } C' \mid \Pi \textbf{ is } \Pi' \mid V \textbf{ is } V' \mid BV \textbf{ is } BV' \mid G \textbf{ is } G' \mid X \textbf{ is } X')^*}{\mathcal{B} \vdash \textit{int-exp } (\ldots)^* \Rightarrow \mathcal{C}'}$$

16 Record interface expression

16.1 Rationale

Record interface expression are used to specify the parameters of the generic modules. They consist in a set of formal module identifiers constraint to an interface expression.

16.2 Syntax

RIE	::=	*mod-id* : *int-exp*	*single*	(RIE1)
	|		*empty*	(RIE2)
	|	RIE_1, RIE_2	*disjoint union*	(RIE3)

16.3 Static semantics

$\boxed{\mathcal{B} \vdash RIE \Rightarrow \mathcal{B}'}$

$$\frac{\mathcal{B} \vdash \textit{mod-id} \Rightarrow \mathcal{C} \quad \langle \mathcal{B} \vdash \textit{int-exp} \Rightarrow \mathcal{C}' \quad \mathcal{C}' \geq \mathcal{C}'' \prec \mathcal{C}' \rangle}{\mathcal{B} \vdash \textit{mod-id}\colon \textit{int-exp} \Rightarrow \textit{mod-id} \Rightarrow \mathcal{C}''}$$

$$\frac{}{\mathcal{B} \vdash \Rightarrow \{\}}$$

$$\frac{\mathcal{B} \vdash RIE_1 \Rightarrow \mathcal{B}_1 \quad \mathcal{B} \vdash RIE_2 \Rightarrow \mathcal{B}_2}{\mathcal{B} \vdash RIE_1, RIE_2 \Rightarrow \mathcal{B}_1, \mathcal{B}_2}$$

17 Interface specification

17.1 Rationale

The interface specifications correspond to a generative description of an interface. can be specified: types, process (and functions as a syntactic sugar), gates, exceptions, constants, equations. The view of an interface can be reduced by using the local declaration. To use other interfaces or modules, inclusion of interfaces and opening of modules is allowed.

17.2 Syntax

int-spec	::=	**type** S [*attrib*]	*abstract data type*	(IS1)
	|	**type** S **is** C[(RT)] (|C[(RT)])*	*constructed type*	(IS2)

|	**process** Π [[$G[(RT)](,G[(RT)])^*$]]][(IP)][: **exit** (T)] [**raises** [[$X[(RT)](,X[(RT)])^*$]]]	*process*	(IS3)
|	**gate** $G[(RT)]$ $(,\ G[(RT)])^*$	*gate*	(IS4)
|	**exception** $X[(RT)]$ $(,\ X[(RT)])^*$	*exception*	(IS5)
|	**behaviour** BV : **exit** (RT)	*constant behaviour*	(IS6)
|	**value** V : S	*constant value*	(IS7)
|	**eqns** *eqn-dec*	*equations*	(IS8)
|	**local** *int-exp*$_1$ **in** *int-exp*$_2$	*local*	(IS9)
|	**open** *mod-id*$_1 \cdots$ *mod-id*$_n$	*open* $n \geq 1$	(IS10)
|	**include** *int-id*$_1 \cdots$ *int-id*$_n$	*include* $n \geq 1$	(IS11)
|		*empty*	(IS12)
|	*int-spec*$_1$ *int-spec*$_2$	*sequential*	(IS13)

17.3 Static semantics

$\mathcal{B} \vdash$ *int-spec* $\Rightarrow \mathcal{C}$

$$\frac{}{\mathcal{B} \vdash \textbf{type}\ S\ \textbf{any} \Rightarrow S \Rightarrow \textbf{type}}$$

$$\frac{}{\mathcal{B} \vdash S' \sqsubseteq \textbf{type}\ S\ \textbf{any}}$$

from all constructors of S results that S is finite

$$\frac{}{\mathcal{B} \vdash (\textbf{type}\ S\ \textbf{finite}) \Rightarrow S \Rightarrow \textbf{type}}$$

S' finite

$$\frac{}{\mathcal{B} \vdash S' \sqsubseteq \textbf{type}\ S\ \textbf{finite}}$$

from all constructors of S results that S admits equality

$$\frac{}{\mathcal{B} \vdash (\textbf{type}\ S\ \textbf{private}) \Rightarrow S \Rightarrow \textbf{type}}$$

S' admits equality

$$\frac{}{\mathcal{B} \vdash S' \sqsubseteq \textbf{type}\ S\ \textbf{private}}$$

$$\frac{}{\mathcal{B} \vdash (\textbf{type}\ S\ \textbf{subtypeof}\ C[(RT)]\ (|\ C[(RT)])^*\ [|\ \textbf{any}]) \Rightarrow (S \Rightarrow \textbf{type})}$$

$\forall i \in 1..n \exists C\ (RT) \in Con(S') \quad (C_i = C) \ \wedge \ (RT \sqsubseteq RT_i)$

$$\frac{}{\mathcal{B} \vdash S' \sqsubseteq (\textbf{type}\ S\ \textbf{subtypeof}\ C_1(RT_1)\ |\ \cdots\ |\ C_n(RT_n)\ |\ \textbf{any})}$$

$$\frac{\begin{array}{l}\mathcal{B} \vdash (RT_1) \Rightarrow \textbf{type} \quad \cdots \quad \mathcal{B} \vdash (RT_m) \Rightarrow \textbf{type} \\ \mathcal{B} \vdash (RT) \Rightarrow \textbf{type} \\ \mathcal{B} \vdash (RT'_1) \Rightarrow \textbf{type} \quad \cdots \quad \mathcal{B} \vdash (RT'_n) \Rightarrow \textbf{type} \\ \mathcal{B} \vdash ((RP) \Rightarrow (RT)) \Rightarrow (RT')\end{array}}{\begin{array}{l}\mathcal{B} \vdash (\textbf{process}\ \Pi\ [G_1(RT_1), \ldots, G_m(RT_m)]\ (RP : RT) \\ \quad : \textbf{exit}(T) \textbf{raises}\ [X_1(RT'_1), \ldots, X_n(RT'_n)]) \\ \quad \Rightarrow \Pi \Rightarrow [\textbf{gate}(RT_1), \ldots, \textbf{gate}(RT_m)]\ (RT) \\ \qquad [\textbf{exn}(RT'_1), \ldots, \textbf{exn}(RT'_n)] \rightarrow \textbf{exit}(RT)\end{array}}$$

$$\frac{\mathcal{B} \vdash (RT_1) \Rightarrow \textbf{type} \quad \cdots \quad \mathcal{B} \vdash (RT_m) \Rightarrow \textbf{type}}{\mathcal{B} \vdash (\textbf{gate}\ G_1(RT_1), \ldots, G_m(RT_m)) \Rightarrow (G_1 \Rightarrow \textbf{gate}(RT_1), \ldots, G_m \Rightarrow \textbf{gate}(RT_m))}$$

Comment: The composition of contexts for gate declaration is a disjunctive composition.

$$\frac{\mathcal{B} \vdash (RT_1) \Rightarrow \textbf{type} \quad \cdots \quad \mathcal{B} \vdash (RT_m) \Rightarrow \textbf{type}}{\mathcal{B} \vdash (\textbf{exception}\ X_1(RT_1), \ldots, X_m(RT_m)) \Rightarrow (X_1 \Rightarrow \textbf{exn}(RT_1), \ldots, G_m \Rightarrow \textbf{exn}(RT_m))}$$

$$\frac{\mathcal{B} \vdash (RT) \Rightarrow \textbf{type}}{\mathcal{B} \vdash (\textbf{behaviour}\ BV\ \textbf{:}\ \textbf{exit}\ (RT)) \Rightarrow (BV \Rightarrow\ \textbf{exit}\ (RT))}$$

Comment: This feature has to be integrated within the base language.

$$\frac{\mathcal{B} \vdash S \Rightarrow \textbf{type}}{\mathcal{B} \vdash (\textbf{value}\ V\ \textbf{:}\ S) \Rightarrow (V \Rightarrow S)}$$

$$\frac{\mathcal{B} \vdash \textit{eqn-dec} \Rightarrow \texttt{ok}}{\mathcal{B} \vdash (\textbf{eqns}\ \textit{eqn-dec}) \Rightarrow \{\}}$$

$$\frac{\begin{array}{rcl}\mathcal{B} & \vdash & \textit{int-exp}_1 \Rightarrow \mathcal{C}_1 \\ \mathcal{B} + \mathcal{C}_1 & \vdash & \textit{int-exp}_2 \Rightarrow \mathcal{C}_2\end{array}}{\mathcal{B} \vdash (\textbf{local}\ \textit{int-exp}_1\ \textbf{in}\ \textit{int-exp}_2) \Rightarrow \mathcal{C}_2}$$

$$\frac{\mathcal{B} \vdash \textit{mod-id}_1 \Rightarrow \mathcal{C}_1 \quad \cdots \quad \mathcal{B} \vdash \textit{mod-id}_n \Rightarrow \mathcal{C}_n}{\mathcal{B} \vdash (\textbf{open}\ \textit{mod-id}_1 \cdots \textit{mod-id}_n) \Rightarrow \mathcal{C}_1 + \cdots + \mathcal{C}_n}$$

Comment: The composition of contexts resulting from elaboration of module identifiers is an overriding one. So clash or names is accepted, but only the last binding is kept.

$$\frac{\mathcal{B} \vdash \textit{mod-id}_1 \Rightarrow \mathcal{C}_1 \quad \cdots \quad \mathcal{B} \vdash \textit{mod-id}_n \Rightarrow \mathcal{C}_n}{\mathcal{B} \vdash (\textit{Binclude}\ \textit{int-id}_1 \cdots \textit{int-id}_n) \Rightarrow \mathcal{C}_1 + \cdots + \mathcal{C}_n}$$

Comment: The composition of contexts resulting from elaboration of interfaces is an overriding one. So clash or names is accepted, but only the last binding is kept.

$$\frac{}{\mathcal{B} \vdash \Rightarrow \{\}}$$

$$\frac{\begin{array}{rcl}\mathcal{B} & \vdash & \textit{int-spec}_1 \Rightarrow \mathcal{C}_1 \\ \mathcal{B} & \vdash & \textit{int-spec}_2 \Rightarrow \mathcal{C}_2\end{array}}{\mathcal{B} \vdash \textit{int-spec}_1 \quad \textit{int-spec}_2 \Rightarrow \mathcal{C}_1, \mathcal{C}_2}$$

18 Equation declaration

18.1 Rationale

18.2 Syntax

eqn-dec ::= **forall** *RT* → *E* (; *E*)* [**and** *eqn-dec*] (EQ1)

18.3 Static semantics

$\boxed{\mathcal{B} \vdash \textit{eqns-dec} \Rightarrow \texttt{ok}}$

$$\frac{\begin{array}{rcl} \mathcal{B} & \vdash & RT \Rightarrow \textbf{type} \\ \mathcal{B}+RT & \vdash & E_1 \Rightarrow \textbf{exit}(\textbf{bool}) \quad \cdots \quad \mathcal{B}+RT \vdash E_n \Rightarrow \textbf{exit}(\textbf{bool}) \end{array}}{\mathcal{B} \vdash \textbf{forall}\ RT\ \rightarrow\ E_1\,;\cdots;E_n \Rightarrow \texttt{ok}}$$

19 Base environment

19.0.1 Types and type constructors

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 derived 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* `ET` 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 ())))`.

20 Further Work

There are a number of features still missing from the language:

- Anonymous renaming and instantiation.
- Relationship with ActOne specifications.

We need to check a number of semantic properties for the language, for example: principal type (context here).

The flattening function must be provided if the language is accepted.

References

[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.

[GS96] Hubert Garavel and Mihaela Sighireanu. French-Romanian Integrated Proposal for the User Language of E-LOTOS. Rapport SPECTRE 96-05, VERIMAG, Grenoble, May 1996. Input document [KC3] 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.

[JGL+95] Alan Jeffrey, Hubert Garavel, Guy Leduc, Charles Pecheur, and Mihaela Sighireanu. Towards a proposal for datatypes in E-LOTOS. Annex A of ISO/IEC JTC1/SC21 N10108 Second Working Draft on Enhancements to LOTOS. Output document of the edition meeting, Ottawa (Canada), July, 20–26, 1995, October 1995.

[JL96] Alan Jeffrey and Guy Leduc. E-LOTOS Core Language. In ISO/IEC JTC1/SC21 Third Working Draft on Enhancements to LOTOS (1.21.20.2.3). Output document of the edition meeting, Kansas City, Missouri, USA, May, 12–21, 1996, October 1996.

[Mun91] Harold B. Munster. LOTOS Specification of the MAA Standard, with an Evaluation of LOTOS. NPL Report DITC 191/91, National Physical Laboratory, Teddington, Middlesex, UK, September 1991.

[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/WG1 N1173 Project 1.21.20.2.3. Output document of the Kansas City meeting, October 1996.

[SG96] Mihaela Sighireanu and Hubert Garavel. On the Definition of Modular E-LOTOS. Input document [GR2] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Grenoble, France, December 9-11, 1996, December 1996.