Towards a proposal for datatypes in E-LOTOS

Source: Belgium,[*] France,[†] Romania,[‡] United Kingdom[§]

Output document of ISO/IEC JTC1/SC21/WG7/1.21.20.2.3
'Enhancements to LOTOS'

Ottawa meeting, July 1995

Abstract

This document makes proposals for the data language of Enhanced LOTOS. It describes a core language, plus a module system and standard libraries. Possible approaches to the dynamic and static semantics of the language are sketched out. The relationship of the model to the behavioural part of LOTOS and the other E-LOTOS enhancements are discussed.

Contents

[*] Represented by Guy Leduc and Charles Pecheur (University of Liège).
[†] Represented by Hubert Garavel (Inria Rhône-Alpes, Verimag).
[‡] Represented by Mihaela Sighireanu (RSI).
[§] Represented by Alan Jeffrey (University of Sussex)

1 Summary

1.1 Objective

The work item on enhancements to LOTOS is developing an extension to the existing LOTOS specification language [10] for data. This paper makes proposals for the structure of such a language, and discusses the major outstanding design decisions.

1.2 Coverage

This paper discusses the data language and module extensions for E-LOTOS. Relationships are given with the behavioural part of LOTOS, and the other proposed extensions.

1.3 Application

The new data language will make it easier to describe the data part of protocols, and ease the production of tools for verification and implementation of protocols.

1.4 Status

Output document of the Ottawa meeting of the ISO working group on LOTOS enhancements, held in July 1995.

1.5 Standards

IS.8807 LOTOS standard.

2 Introduction

Experience with using LOTOS as currently standardized by ISO has raised a number of criticisms from users who have to deal with the data part of LOTOS (based on ACT ONE) to specify real protocols. The most common problems are:

1. the data part definition is too verbose,
2. Even the most basic data types have to be redefined again and again, and the standard library is not effective in cutting down this effort.

3. Fundamental data type specifications may be "corrupted"; even if taken from the standard library, types may be extended inconsistently, thus collapsing their meanings (this is known as the "persistency" issue).
4. Equational semantics can be tricky; in practice, most people use rewriting rules, and a pattern-matching style that is inconsistent when interpreted as equations.

   These difficulties affect tool support, as different tools may produce different results depending on the strategy used to turn equations into rewrite rules. Equivalence for an arbitrary algebra is inherently undecidable, and so it is impossible to provide sound and complete simulators for existing LOTOS.
5. 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.

These deficiencies make LOTOS is 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 [24]

The decision of the Paris interim meeting (January 1995) was to replace the datatypes language of full LOTOS with:

1. a pure strict (in the sense of call-by-value) functional language with user-defined recursive datatypes, and
2. an equational specification language for functions.

This should be considered as a single two-level language rather than two distinct languages sitting side by side. The first level is a 'functional' language suitable for full execution, while the second level is an 'equational' level suitable for abstraction and much in the spirit of ACT ONE.

The reason for choosing a functional language was that most LOTOS designers were already using a subset of ACT ONE in a functional style by discriminating the operations into constructors and functions. There is a large body of work on semantics for functional languages, stretching back to the λ-calculus. There is also experience of providing tool support such as type checkers and code generators for such languages.

At the Ottawa meeting (July 1995), the working group decided to break the data language into four parts:

1. a small functional language, described in Section 5,
2. shorthand notation, described in Section 6,
3. a modules system, described in Section 7 and
4. a base environment, described in Section 8.

In Section 3 we mention research work for improving the data type and module features of LOTOS, as well as existing functional languages from which ideas could be borrowed when designing E-LOTOS.

In Section 4 we define guidelines for the syntactic and semantic definition of E-LOTOS.

In Section 9 we discuss how the data language relates to the LOTOS behavioural language, and in Section 10 we discuss how the data language relates to the other E-LOTOS proposed enhancements.

The present document does not attempt to design such a language, but lists some of the design decisions which have to be made, and their implications. An abstract syntax for the core functional language is provided, together with some possible shorthand notations. The concrete syntax and semantics of this language is *not* described in this document.

3 Related work

3.1 Overview

Since 1988, there have been several proposals for enhancing LOTOS datatypes and modules. In this section, we briefly present these proposals, and we also introduce some functional programming languages which can be of interest for the design of E-LOTOS.

3.2 The PhD Thesis of Ed Brinksma

Ed Brinksma served as the Editor of the ISO 8807 standard. In his PhD thesis [5] entitled "*On the design of Extended LOTOS*", he proposes several extensions to LOTOS, both for the behaviour and data part.

As far as the data part is concerned, Ed Brinskma's proposal still remains in ACT-ONE's initial algebra framework: functions are defined equationally; the domain of a sort is defined by the set of its ground terms; there is no separation between constructor and defined functions, etc.

Therefore, it is clear that these proposals regarding datatypes are not aligned with the recent decisions of the E-LOTOS Committee, who decided to shift towards a functional language.

Ed Brinksma also introduces a system of modules for E-LOTOS. In his proposal, a module contains definitions of types and processes. A top-level E-LOTOS specification is a module itself (with a different keyword: "**specification**" instead of "**module**").

Modules can also be composed together, using "combinators", to obtain new modules. The type combinators that exist in LOTOS are extended to modules:

- "synthesis": modules can be imported in other modules, leading to an hierarchical organization of specifications;
- "renaming" of types, sorts, operations, and processes;
- "parameterization" and "actualization" of formal sorts, formal operations, and formal equations.

Two different forms of abstraction are available for modules:

- the objects declared in a part of a module following the "**where**" or "**use**" keywords are local to this module and are not exported outside;
- a "hiding" combinator exists for modules, which allows to remove certain types, sorts, operations, and processes from the list of objects exported by a module.

Combinators can be applied in arbitrary order, leading to algebraic "module expressions". This is not possible in LOTOS, because it is necessary to introduce a new module name each time a combinator (renaming or actualization) is applied.

The static semantics of modules is operational: it is defined as a function which computes, for each modular expression, the list of types, sorts, operations and processes declared in the module expression. This function is defined by induction on the set of module combinators.

Ed Brinksma's proposal seeks upward compatibility with standard LOTOS. However, it presents some drawbacks, as some "undesirable" features of LOTOS definition are retained:

- The notion of LOTOS "type" is kept as is, leading to a redundancy with the module concept, as there are two different levels of modularity: types and modules.
- Moreover, these two levels are not symmetric: there is no "hiding" combinator for types; the "synthesis" combinator for modules is called "combination" for types.
- Modules cannot be parameterized by formal processes.
- The possibility of defining types and processes locally to the body of a process is retained (which is probably unnecessary if modules provide abstraction facilities).

3.3 The LOTOSPHERE proposal

The LOTOSPHERE proposal was produced in the framework of ESPRIT project LOTOSPHERE [4, 3] and input to the E-LOTOS Committee during the Yokohama meeting (November 1993) [29].

As far as datatypes are concerned, this proposal remains in the algebraic framework, but introduces substantial changes with respect to ACT ONE: it makes a distinction between constructors and defined functions and it allows functions and constructors to be defined partially. However, in this proposal, functions are still defined by equations and the semantics is still based on initial (partial) algebras, which is not compatible with the functional approach taken by the E-LOTOS Committee.

Regarding modules, the LOTOSPHERE proposal extends Ed Brinksma's approach in many ways:

- (ACT-ONE) types and (LOTOSPHERE) modules are merged into a single notion of module. Upward compatibility is ensured in the sense that LOTOS types can be translated directly into modules (simply be changing the keywords "**type**...**endtype**" into "**mod**...**endmod**").
- The LOTOSPHERE proposal draws a clear separation between interfaces (called *descriptions*) and implementations (called *modules*). This separation, which exists in many programming languages (e.g., Modula-2, Ada, etc.), was also advocated by Jose Manas [14].
- Modules can contain definitions of sorts (with or without predefined equality), operations (constructors, functions, and predicates[1]), equations, and processes.
- Descriptions can contain declarations of sorts, operations, and processes. For operations and processes, only the profiles[2] are declared.

[1] A predicate is a special notation for a function returning a boolean result.

[2] The profile of an operation is the list of sorts of its argument and the sort of its result. The profile of a process is the list of its gate parameters, the list of the sorts of its value parameters and its functionality.

- For simplicity purpose, descriptions and modules cannot be nested: it is not possible to declare a description or module inside another description or module.
- Descriptions can be combined together using combinators such as: import, union, removal (i.e., hiding), and renaming.
- Similar combinators also exist for modules: import, union, and renaming. Two forms of abstraction are available for modules: selective export (the module specifies which objects are visible outside) and restriction (the contents of a module are "filtered" through a description: only the objects declared in the description will be accessible).
- It is possible to attach a description to a given module, and several modules can implement the same description. But this is not mandatory. If a module has no description attached, its description will be considered to be implicit and synthesized automatically. This is useful for programming "in-the-small". This also allows to filter a given module through different descriptions, thus providing different levels of abstraction ("views") of the same module.
- As regards parameterization, modules can be parameterized by descriptions. Therefore, they can be parameterized with (formal) sorts, operations, and processes. Generic modules can later be instantiated with actual arguments.

As in Ed Brinksma's proposal, the semantics of modules is operational and based on functions mapping module expressions and description expressions to to (structured) collection of sorts, operations, and processes. Renaming morphisms are used to handle renaming and actualization, and to generate unique names for local objects.

Although the LOTOSPHERE proposal covers many suitable requirements for modular LOTOS, it could be improved in several ways:

- The distinction between constructors and functions is not explicit in descriptions, which allows non-persistent specifications to be written, and also creates problems in pattern-matching expressions (where constructors have to be known by the compiler). The modification proposed in [26] solves this problem.
- The proposal allows renamings to be explicitly named. This introduces a third class of objects (apart from descriptions and modules), which has no clear practical justification. Moreover, the reusability of renamings is weak: it is very unlikely that the same renaming can be applied to several modules.

In [36, 30], a comprehensive library of predefined datatypes for E-LOTOS, including booleans, natural numbers, (generic) sequences, finite sets, arrays, maps, records, disjoint unions, etc., is proposed.

For all these types, ad hoc syntactic constructs (such as "**settype**", "**recordtype**" and "**uniontype**") are provided. These shorthand notations are nothing but macro-definitions that expand to the datatype and module language defined in the LOTOSPHERE proposal, according to a mapping function.

3.4 The work of Charles Pecheur

In his proposal for E-LOTOS [26], Charles Pecheur introduces both:

- a pure functional model based on a recursively-defined datatypes and recursive functions (with a distinction between constructors and functions),
- a "rich term syntax", which allows concise and readable shorthand notations to be used for numbers, sequences, selectors, modifiers, etc.

Charles Pecheur also developed the APERO 2.0 tool[3], which enhances LOTOS with a collection of language facilities (library types, syntactic extensions and shorthand notations). These facilities are automatically translated into standard LOTOS, which allows to take advantage of existing LOTOS tools.

3.5 The Opal language

Opal [27, 8, 38] is a strongly typed, higher order, strict functional language, which integrates concepts borrowed from functional programming and algebraic specifications.

In Opal, datatypes are defined as recursive types generated by free constructors: the general concept for defining types is the sum of products. Records, unions, lists, enumerated types, etc. are obtained as particular cases of this general mechanism. A standard library exists, which encompasses boolean, natural and integer numbers, real numbers, characters, products, unions, sequences, strings, sets, bags, maps, arrays, and graph-like data structure.

In Opal, functions are defined by combining operations such as constant denotations, variables, pattern-matching case statements, tupling, λ abstraction, function application, local declarations ("**let**" statements), etc.

Opal also provides a modules system, which supports both "in the small" and "in the large" programming. An Opal program is a collection of modules (called "structures") connected together by import relations . Modules consist of two parts: a visible signature and an implementation (with a one-to-one relation between them). The implementation part of a module can be written in another language.

Various abstraction facilities are available. A module can have local objects, defined only in its implementation part and not declared in its signature. When importing a module, one can restrict the list of imported sorts and functions (this allows to define multiple views of a module). Also, it is possible not to export the free constructors of exported sorts (in this way, a sort may export only its name or also his implementation). Unfortunately, it is not possible to have two different implementations of the same signature.

Opal modules do not support renaming: it is not possible to give different names to the same sort or function.

Modules can be parameterized with a list of formal sorts and functions. However, it is not possible to group these formal objects into a signature, possibly leading to very long lists of formal parameters[4]. Opal has an algebraic flavour by allowing parameterization and overloading, rather than the polymorphism found in ML.

Opal combines functional concepts together with a specification language. Modules can contain algebraic equations expressing properties. These equations are either exported or hidden.

[3] APERO is a loose acronym for "*Act-one PrE-pROcessor*"

[4] The same problem was identified in Ada 83 and addressed in Ada 95.

According to the Opal team, an efficient Opal-to-C compiler has been developed: the generated code is claimed to be of comparable efficiency than hand-written C code and “in any case [...] much faster than that of traditional functional languages. There are orders of magnitudes between the execution times of Opal and e.g. ML or HOPE” [34, 38].

3.6 The SML language

Standard ML (SML) [20, 21] is a strict, impure functional language, with a strong polymorphic type system, and a parameterized typed module system. SML has many compilers, including Edinburgh ML, New Jersey ML (SML/NJ) and Poplog ML (PML). There are also a number of ML variants in existence, such as Caml and Lazy ML. SML has a number of textbooks(such as [25, 40]) used for undergraduate teaching. SML is one of the few ‘real world’ languages to have a formal language definition [20], including a static and dynamic semantics.

We should not forget that ML and Opal have been designed as a programming language whereas E-LOTOS should remain a specification language. This does not only mean keeping compatibility with ACT ONE (with some form of **eqns** declaration) but also that some design choices in the core functional part of E-LOTOS be made carefully.

Note: There is currently a project to develop a revised SML standard, called ‘SML 2000’. This project is covering some material which is relevant to the E-LOTOS project, in particular subtyping for records. This project should provide a source of ideas which could be incorporated into the E-LOTOS data language.

4 Rationale

4.1 Overview

In this section, we discuss general requirements which should be satisfied by the definition of the E-LOTOS language. These requirements concern the syntax and static semantics features. They should apply not only to the data language and modules system, but also to the whole language E-LOTOS.

4.2 Syntax

The syntax of LOTOS, as given in [10], is formally defined by a context-free BNF grammar, which can be — possibly with some adaptations — directly passed to compiler-generators for LALR(1) grammars.

The syntax of E-LOTOS should also be formally defined and have similar “good” properties, so that the correctness of the grammar can be checked automatically by compiler tools, and so that compiler front-ends for E-LOTOS can be produced easily.

The decision of which syntax to adopt is largely a matter of taste. However, several criteria should be followed:

1. Uniformity among all constructs
2. Compatibility with current LOTOS syntax

3. Simplicity, which means that redundant constructs should be avoided (Occam's razor paradigm) unless really needed, e.g., for software engineering reasons
4. Technical properties of the syntax: the grammar should be context-free and unambiguous (so as to allow deterministic parsing), and even LR(*k*) or LALR(*k*) if possible (to allow efficient parsing).

As regards the integration of the datatype and behaviour parts, two possibilities are retained:

(A) either the syntaxes for these two parts will be unified,

(B) or these syntaxes will be kept distinct by integrating a functional language in E-LOTOS (e.g., SML) to allow for easier code reuse from both worlds.

The solution (B) is likely to cause problems. For example, mixing the LOTOS and SML syntaxes would collide on several aspects:

- Some legal identifiers in LOTOS are keywords in SML and vice-versa
- Some keywords exist in both LOTOS and SML, but at different places and with a different intended meaning: '**of**', ':', '=>', etc.
- The definition of a token is different (for example '[]' is two tokens in SML and one token in LOTOS).
- More importantly, plugging the full grammar of SML expressions as LOTOS value denotations would make the LOTOS grammar ambiguous. For example, in the behaviour 'g !a [b]; B', the fragment '[b]' can be parsed either as the one-element list to which function 'a' is applied or as a guard over the action.
- SML allows functions to be declared "infix" in declarations, and such declarations are subject to scope. Precedence and associativity can also be declared. For this reason, SML cannot be parsed statically.
- Both LOTOS and SML allow complex nestings of operators, and use different methods (such as explicit '**end**' keywords or bracketing) to resolve ambiguities.

In order to maintain compatibility with LOTOS syntax, it will not be possible to allow all valid first-order SML in E-LOTOS.

It seems that solution (A) would be preferable. However, in the event that solution (B) would be retained, it was agreed during the Ottawa meeting to resolve syntactic conflicts between LOTOS and the datatype language in favor of LOTOS.

4.3 Static semantics

The role of the static semantics for a language is to analyze decidable properties of programs. The most common of these is type checking, which (in the case of strongly typed languages such as LOTOS) ensures that no 'run-time' type errors will be found.

The static semantics of LOTOS given in [10] covers the following issues:

Identifier binding: binding of process-identifiers, gate-identifiers in the behaviour part; binding of type-identifier in the data part; binding of sort-identifiers, operation-identifiers, and variable-identifier in both the behaviour and data parts.

Computation of type signatures, to determine the set of sorts, operations and equations contained in a type.

Type-checking of value-expressions, together with operation overloading treatment, in both the behaviour and data parts.

Computation of functionality: to determine (approximately) whether a behaviour-expression can terminate with an "**exit**" instruction.

This static semantics is not defined formally, but semi-formally, using a combination of mathematics (sets, tuples, functions), abstract syntax trees, functions defined by induction on syntax trees, fixed-points, logical constraints, and natural language.

Therefore, the static semantics of LOTOS is sometimes ambiguous and it was not possible to check automatically its consistency (using computer tools). Consequently, a number of errors have been detected (see [23] for a 22-pages report).

We propose to adopt a formally defined static semantics for the E-LOTOS data language.

We need to ensure that the static semantics we provide is decidable, and that it is reasonably efficient in practice, and that it has good formal properties. For instance, the "subject reduction" property might be a desirable one.

It would also be desirable to have a static semantics specified in such a way that it can be checked automatically (at least, partially) by computer tools.

There are at least two possible approaches to specify the static semantics of E-LOTOS:

- The first is to provide a static semantics based on type judgements, on the typed λ-calculi, and on the Curry-Howard isomorphism. To see in detail how this can be given for a language similar to that defined here, see the SML language definition [20].
- The second is to provide a more operational static semantics, for example based on attribute grammars, which can be efficiently implemented.

Hopefully these two approaches are not contradictory, although we will have to decide which will be in the E-LOTOS standard.

5 Base language

5.1 Overview

In this section, we discuss the some of the major design choices for the datatypes part of E-LOTOS. Modules are not discussed, since they will be dealt with in a further section.

We then sketch how this language can be given a formal static and dynamic semantics.

5.2 Language design

The reader interested in the Committee's discussion about the datatype part of E-LOTOS may consult the minutes of the ISO E-LOTOS meetings held in Madrid (January 1994), Southampton (July 1994), Paris (January 1995), and Ottawa (July 1995). The minutes of the COST-247 meeting held in Warsaw (June 1995) are also of interest.

The following subsections list the "high level" design decisions taken by the E-LOTOS Committee, possibly with explanatory notes. These notes should eventually grown into a commentary on the E-LOTOS standard.

Open issues are also presented and technical arguments are provided in order to measure the potential impact of future design decisions.

5.2.1 Scope of the New Work Item

According to the scope of the new work item, the datatype enhancements should provide "a more user-friendly notation for datatype descriptions". "This will consider:

- Built-in types (e.g., booleans, characters, bits, integers, enumeration, and possibly reals),
- Composed types (e.g., records, unions, sets, arrays, strings, character strings, bit strings),
- Partially defined functions or subtyping,
- Constructive types,
- Interface with data types in other languages, e.g., ASN.1, SDL

All enhancements to be defined shall meet the following requirements:

- They shall maintain the existing LOTOS properties which support design and engineering procedures, i.e. for refined, behavioural compatibility, executable specifications, for test derivation and proper tool support.
- The enhancements to LOTOS shall progress the definition of the textual version of the language together with the graphical version (G-LOTOS).
- The enhancements to LOTOS shall ensure compatibility with the LOTOS base standard."

5.2.2 "Impure" features

We agreed that the data language of E-LOTOS should be strongly typed, and that type-checking should be done statically, i.e., at compile-time.

We agreed not to introduce in E-LOTOS "impure" (non-functional) features that do not exist in LOTOS, such as explicit assignment to variables or functions with side-effects.

In LOTOS, the evaluation of a value expression has no effect on the environment: it does not modify the value of variables stored in memory, nor the observable behaviour of the process that evaluates the value expression.

Having side-effects in E-LOTOS would introduce all the semantic problems that arise when a shared memory is accessed by several concurrent processes (race conditions).

There are similar problems with any form of data with side-effects (for example, file access, database lookup, communication on channels).

5.2.3 Pointer types

We agreed not to introduce pointer types (also called "reference types") in E-LOTOS.

The introduction of pointer types in E-LOTOS would pose severe semantic and pragmatic problems, such as references to dead objects, concurrent access to the same object by several parallel processes, and the problems caused by polymorphic pointers.

5.2.4 Distinction between constructors and functions

Users of most specification languages have to specify two different things: *data structures*, which define the types of values handled in the specifications, and *algorithms*, which express computations involving these values.

In LOTOS, the algorithms are split in two: the *sequential* algorithms, which are specified in the data language (currently ACT-ONE), and *concurrent* algorithms, which are specified in the behavioural language.

Due to the adoption of ACT-ONE's semantics based on initial algebras, data structures and sequential algorithms are not clearly separated, which confuses many users of the language.

We believe that E-LOTOS should adopt a more pragmatic style, similar to algebraic specification languages such as PLUSS, LPG, or μCRL, and to functional programming languages such as SML, Miranda, Haskell, Opal, by distinguishing *constructors* from *defined functions*.

Quoting the rationale of LOTOSPHERE proposal [29]: "A crucial ingredient [...] is to make a clear distinction between so-called constructors and functions in data type definitions. Constructors are those operations that are needed to build up data objects, whereas functions are intended to map them to other objects, e.g., to extract information. This approach leads to a the identification of a functional sublanguage for the data related part of LOTOS, which allows a more constructive approach to the specification of data types. Using the concept of constructors, an automatic generation of operations, such as *equality predicates*, *discrimination functions*, and *projections* becomes possible, i.e., without the user's obligation to specify them."

5.2.5 Recursively defined datatypes

When describing protocols, one needs not only basic datatypes (such as bits and integers) but also sophisticated data structures, such as records, discriminated unions and lists.

Although they are used in the ISO language Estelle [9], we believe that Pascal-like types (and similarly C-like or Ada-like types) are not appropriate for E-LOTOS since:

1. Without pointer types, Pascal-like languages are unable to express dynamic data structures such as lists.
2. Discriminated unions in Pascal raise another semantic problem, when one tries to access a field that does not correspond to the current value of the discriminant. Such violations are a well-known way to subvert strong typing, thus compromising the correctness of the whole specification. Clearly, such situations should be prohibited, since they go against the functional style of the behaviour part of LOTOS, which using appropriate syntactic and static semantic restrictions, always

ensures that a variable is initialized before it is accessed. Run-time type checking is not suitable for a specification language such as LOTOS.

The best solution for avoiding these semantic and pragmatic issues whilst still allowing tool support by type checkers and code generators is to adopt *recursive types* (or in algebraic specification terminology *sorts generated by free constructors*).

There are a variety of syntaxes available for declaring such sorts. However, some common-sense requirements should be considered:

1. The constructors of a given sort should be declared together with that sort, instead of being distributed through a whole description as is currently permitted in LOTOS. Providing such scope constructs would improve the readability of E-LOTOS descriptions, and would forbid the modification of existing or pre-defined type libraries by adding new constructors or collapsing old ones.
2. It should be possible to name the arguments (i.e. formal parameters) of constructors (in particular, it should be possible to name the fields of records). This is not the case in existing LOTOS, where the arguments of an operation are only defined by their sorts. As LOTOS is supposed to be a specification language, this is a clear drawback. For example, the formal description of the ISO 8072 transport service specification contains:

   ```
   (*
     The failure probability parameters have the following 4-tuple
     structure: Failures = Prob * Prob * Prob * Prob, where the
     arguments respectively represent the failure probabilities
     termed TC establishment, transfer, TC resilience, and TC release.
   *)
   type TCFailureProbabilities is Probability
     sorts
       Failures
     opns
       Failures : Prob, Prob, Prob, Prob → Failures
       ...
   endtype
   ```

   in which the meaning of the four arguments of constructor ‘Failures’ cannot be explained but with a comment.

Keeping in mind these requirements, we can choose between (at least) two syntaxes. The first possibility is to use a syntax similar to the syntax of process definitions:

```
sort BoolList is
  nil : BoolList
  cons (hd:Bool, tl:BoolList) : BoolList
endsort
```

```
sort TCFailureProbabilities is
  Failures (TCEstablishment, Transfer, TCResilience,
         TCRelease : Prob) : TCFailureProbabilities
endsort
```

Another possibility is to use the ML-like syntax:

```
datatype BoolList :=
  nil
 | cons of { hd:Bool, tl:BoolList }
endtype
signature TCFailureProbabilities :=
  include Probability
  datatype Failures :=
    Failures of {
      TCEstablishment : Prob,
      Transfer : Prob,
      TCResilience : Prob,
      TCRelease : Prob
    }
  endtype
  ...
endsig
```

5.2.6 Recursively defined functions and non-termination

We agreed to have recursively defined functions.

One of the main differences between algebraic languages (like ACT-ONE) and functional languages (like SML) is that the initial algebra approach is not confronted to the possibility of non-termination. For example, the specification:

```
type Loop is
  sorts one
  opns loop : nat → one
  eqns forall x : int ofsort one
    loop(x) = loop(x+1);
endtype
```

specifies a sort ‘one’ with only one element; its quotient algebra has a single equivalence class containing all terms of the form: loop (0), loop (succ (0)), loop (succ (succ (0))), and so on[5]. This can be compared with the SML function:

```
fun loop x = loop (x+1);
val loop : int → ’a
```

[5] However, all LOTOS tools that implement equations as rewrite rules are likely to diverge when asked to evaluate ‘loop(0)’.

which has the semantics of never terminating.

It is by no means clear that non-termination is at all desirable in a specification language, albeit one with implementation very much in view. An alternative to non-termination is to restrict recursion so that it will always terminate. This could be done in two ways:

- Either by forcing the specifier to provide a termination proof for each recursive function definition, e.g., an induction proof on the length(s) of the function argument(s) and a subsidiary proof that the function does not increase the length of its argument(s); such an argument is too much to require of a specifier; moreover, E-LOTOS compiler should be able to verify the correctness of this proof.
- Either by embedding sufficient conditions for termination [6, 7] in the static semantics of E-LOTOS (and, consequently, in the compilers) and rejecting all programs which do not satisfy these conditions (even if they terminate).

We propose to allow general recursion, unless it can be demonstrated that there is a language with restricted recursion which is sufficiently powerful and simple to be used in specifications.

5.2.7 Partial functions and exceptions

In the initial algebra framework used by ACT-ONE, all functions are defined totally. However, experience indicates that there is a need for partially-defined functions.

Quoting the rationale of LOTOSPHERE proposal [29]: "An [...] important improvement is the possibility to define *partial* functions [...] on data types, i.e., functions [...] whose application is not defined for all data objects of a given sort. This relieves the specifier of the burden from specifying explicit "error-handling" in those cases where this is not relevant for the object of specification".

Examples of partial functions are: division by zero, taking the head of an empty list, etc. More generally, such situations occur in "case" statements, when a given value cannot be matched against a set of patterns.

As far as the dynamic semantics to partial functions (and expressions) is concerned, there are (at least) three possibilities:

1. The semantics could be left undefined, and up to implementors: in such case, a run-time error (with a diagnostic message) is likely to occur. This approach is used in functional languages such as LISP, Scheme, Opal, etc.
2. The semantics could be said to be equivalent as non-termination, i.e., an infinite loop. However, there is an essential difference between those two concepts: for a given (algebraically closed) expression, partiality is semi-decidable, whereas non-termination is undecidable.
3. An exception mechanism (similar as that of Ada, SML, or Modula-3) could be introduced. A exception is raised when an expression yields an undefined result.

With the third possibility, one has to introduce a new class of objects (exceptions) as well as syntactic constructs in the data language to declare, raise, and catch exceptions. For instance, the function that extracts the head of a list could be defined as follows (using SML-like syntax):

```
exception EmptyList
function hd (l:NatList) :=
  case l in
    cons { hd=h:Nat, tl=any:NatList } → h
   | nil → raise EmptyList
  endcase
endfun
```

Exception handlers can be used in expressions, for example:

```
hd e handle EmptyList → nil
```

Exceptions do not have side-effects, they just extend the range of values a term may terminate with.

There are some difficulties with the SML exception system, notably:

1. the type system does not specify which exceptions a function may raise (in comparison with, say, Ada and Modula-3),
2. polymorphic exceptions cause problems,
3. exceptions may propagate outside their scope, for example:

   ```
   let exception e in raise e end
   ```

   this presents problems in providing a semantics to exceptions

If exceptions are introduced in E-LOTOS, then they should be given an explicit semantics and they should be related strongly to exception handling in the behavioural language.

We propose to investigate further whether exceptions are required in describing concurrent systems. If there is a demonstrable need for exceptions, then they should be provided at both the functional and behavioural level, with a clean semantics for how exceptions are propagated from one to the other.

5.2.8 Higher-order functions

ACT-ONE only allows first-order functions, i.e., it it does not allow function expressions in the way that most functional languages do (functions are not “first-class” citizen).

This restriction is quite severe, for example it does not allow a declaration such as:

```
function filter (p : int → bool) (l : int list) : int list :=
  case l of
    nil → nil
   | cons { hd=h, tl=t } [p h] → cons { hd=h, tl=filter p t }
   | cons { hd=h, tl=t } → filter p t
   endcase
endfun
function limit (max : int) : int list → int list :=
  filter (fn d → d < max)
```

```
endfun
function small : int list → int list :=
  limit 10
endfun
```

Although introducing higher-order functions in E-LOTOS would give a greater expressiveness, it would also present some problems:

1. the semantics of higher-order concurrency is still under development; for example some the full abstraction results for first-order languages do not exist yet for higher-order languages, and
2. implementing tools for higher-order languages is harder than implementing tools for first-order ones (e.g. requiring closures).

Also, the availability of generic modules parameterized by (formal) functions greatly reduces the need for higher-order functions at the expense of some extra verbosity, for example the above higher-order functions become:

```
signature Prop :=
  type data
  fun p : data → bool
and Filt :=
  include Prop
  fun f : data list → data list
and Max :=
  const max : int
and IntFilt :=
  fun f : int list → int list
endsig
generic Filt (P : Prop) : Filt :=
  open P endopen
  function f (l : data list) : data list :=
    case l of
      nil → nil
     | cons { hd=h, tl=t } [p h] → cons { hd=h, tl=f t }
     | cons { hd=h, tl=t } → f t
    endcase
  endfun
and Less (M : Max) : Prop :=
  open M endopen
  datatype data == int endtype
  function p (x:int) := x < max endfun
endgen
module Ten : Max :=
  constant max := 10 endconst
and Small ==
  Filt(Less(Ten))
endmod
```

We believe that the semantic problems of modelling higher-order functions outweigh their practical advantages, so that E-LOTOS should only permit first-order functions.

5.2.9 Overloading

LOTOS currently supports *operation overloading*: it is possible to declare several functions with the same identifier, provided that these functions differ either by their number of arguments, the sorts of their arguments or the sort of their results.

Overloading is allowed as long as the sort of each expression can be determined statically. If necessary, the coercion operator "**of**" can be used to solve ambiguities.

```
type T is
  sorts A, B
  opns F : → A, F : → B
endtype
...
G !(F of A); G !(F of B); stop
```

The E-LOTOS Committee has discussed the issue of keeping or removing overloaded functions from E-LOTOS.

The drawbacks of overloading are[6]:

1. It can decrease the readability of programs, since it is sometimes not obvious for the reader to figure out which function is invoked (although the compiler ensures that there is no ambiguity in determining a unique function).
2. Adding a new (overloaded) function definition in an already existing program may create ambiguities, which have to be solved by introducing "**of**" coercions at appropriate places[7]. This is a problem of compositionality.
3. In the absence of overloaded functions, most functional languages provide a *principal* (or *most general*) type for every typeable expression. This property is necessary for other features, such as type inference and polymorphism.

   If overloaded functions are allowed, this property does not hold, except for expressions which do not use overloaded functions, or in which all ambiguities are properly solved using "**of**" coercions.
4. A module system allows the safe reuse of identifiers without needing overloading. For example, a Real module and an Integer module might both export a function '+', and any ambiguity is resolved by whether the Real or Integer module is imported. It is only if both modules are imported that the ambiguity has to be resolved (e.g. by decorations such as 'Real.+' and 'Integer.+').
5. Without overloading, it is possible to determine the scope of a variable syntactically, without knowing any type information. For example, the following programs are equivalent if overloading is not allowed:

```
( foo y where function foo (x:int) := x+1 endfun )
( bar y where function bar (x:int) := x+1 endfun )
```

[6] Some authors of the present Annex might disagree with some items of the following list.

[7] which are all places where the compiler signals an ambiguity

However, in the presence of overloading, they can be distinguished by the context:

```
let y:real = 0.0 in ... where
   function foo (x:real) := x+2.0 endfun
   function bar (x:real) := x+3.0 endfun
```

For this reason, most existing mathematical theories of type do not support overloading, since it breaks the property of α-conversion (which allows renamings such as the above).

The advantages of overloading are[8]:

1. Overloading is primarily intended for notational convenience. It is not obvious that removing overloading from E-LOTOS is the right way to obtain "a more user-friendly notation for datatype descriptions" (which is the goal defined in the scope of the new Work Item).
2. Overloading exists in most computer languages. Some languages (like Fortran, Algol, Pascal, C, and SML) only allow overloading for built-in operators (e.g., + on integers, and + on reals).

   Some other recent languages (such as Ada, C++, Eiffel, Opal, etc.) have made overloading uniformly available for both built-in and defined functions. This results in a greater convenience for the programmer to exploit and fewer special cases to memorize.
3. User-defined, overloaded functions do not prevent type-checking from being done at compile time (this was done for the first time in 1962 for APL). Overloading does not add much complexity to languages and compilers, since overloading treatment is usually isolated in a well-defined part of static semantics, without impact on dynamic semantics.
4. Well-known, efficient algorithms exist to perform type-checking and solve overloading simultaneously [1]. These algorithms use either two passes (as in the case of Ada) or even a single bottom-up pass [2].
5. The so-called "compositionality problem" mentioned above is not serious. First, overloading does not prevent widely-spread languages (e.g., Ada) from working satisfactorily.

   Also, this "compositionality problem" exists anyhow, even in absence of overloading. For instance, if a program that uses only the 'Int' module is modified to import also the 'Real' module, the proposed "decorations" require to replace all occurrences of '+' with either 'Int.+' or 'Real.+'. This is non-compositional as well.

   Note: the idea of labelling functions with module names explicitly is not very different from the "**of**" operator of LOTOS, except the fact that it is applied to function names rather than expressions.

   On the opposite, we can even say that overloading is more compositional, because it does not require that all functions have different names, thus reducing the risk

[8] Some authors of the present Annex might disagree with some items of the following list.

of name clashes when importing new modules. For instance, if two modules, a *fifo_queue* module and a *stack* module, both of them exporting an *is_empty* function with different profiles, are imported simultaneously in a third module, no name clash will occur, and both functions will be coexist and be accessible.

6. Overloading fits well with generic modules. Let's consider a generic FIFO queue module, exporting a sort *queue* and a large collection of functions, among which a function *is_empty : queue* →*bool*.

   With overloading, this module can be instantiated several times by simply actualizing the sort *queue*. Doing so, all the functions exported by the module will be available by default with the same identifiers (with overloaded profiles).

   Without overloading, it is also necessary to actualizing all these functions by giving them new names (e.g., is_empty_packet_queue, is_empty_message_queue, etc.), which is rather cumbersome.
7. Overloading is existing practice in LOTOS. Forbidding overloading in E-LOTOS would raise difficult compatibility issues with the current standard.. In such case, an algorithm should be provided to translate existing LOTOS descriptions into ones without overloading.
8. The "rich-term syntax" proposed by Charles Pecheur [26] relies on the existence of overloading (see Section 6.2 below).

5.2.10 Polymorphism

An extension which makes a large difference to the static semantics of the language is to allow polymorphic type declarations, such as:

```
datatype 'a List :=
  nil | cons of { hd:'a, tl:'a List }
endtype
```

This type can then be instantiated with a type, for example:

```
function sum (x:Nat List):Nat :=
  case x in
    nil → zero
    cons { hd=h:Nat, tl=t:Nat List } → h + (sum tl)
  endcase
endfunc
```

or used in polymorphic function declarations, for example:

```
function hd (x:'a List):'a :=
  case x in
    cons { hd=h:'a, tl=any:'a List } → h
   | nil → raise EmptyList
endfunc
```

Type constructors can be polymorphic in more than one type variable, for example:

```
  datatype ('a,'b) Tree :=
    leaf of 'a | node of { root:'a, left:('a,'b)Tree, right:('a,'b)Tree }
  endtype
```

Although overloading is sometimes called “ad hoc polymorphism”, it should be understood that it is not a particular case of (e.g., ML) polymorphism. Overloading and polymorphism provide distinct functionalities.

For instance, if overloading is forbidden, then specifiers wishing to define common mathematical operators such as ‘+’ on new datatypes (such as complex numbers, matrices or vectors) would have to do so inside a module, so that the compiler can distinguish between ‘Complex.+’, ‘Matrix.+’, and so on. A polymorphic + would have to apply to all types, and perform similar computations on its arguments irrespective of their type.

To quote [39]: “We should be careful not to confuse the distinct concepts of overloading and polymorphism. Overloading means that a (small) number of distinct abstractions just happen to have the same identifiers; these abstractions do not necessary have related types, nor do the necessarily perform similar operations on their arguments. Polymorphism is a property of a single abstraction that has a (large) family of related types. The abstraction operates uniformly on its arguments, whatever their types.”

Although overloading and polymorphism can be both useful, it is not suitable to have them simultaneously included in a computer language. If so, the type-checking algorithm would become more complex, as the possibilities of type-checking ambiguities would increase. This is one reason why ML does not allow overloading on user-defined functions. Another reason is that ML also allows type inference, and it is known that type-checking of expressions is NP-complete if type inference and overloading are combined.

Therefore, for the final standard, we will have to decide whether or not to include either polymorphism and type inference, or overloading. There are several possible directions:

1. allow overloading and not introduce polymorphism, as in existing LOTOS,
2. allow polymorphism, but without any overloaded operators, as in existing SML,
3. or have neither polymorphism nor overloading in the base language, and move the problem to the modules system.

The advantages of polymorphism are[9]:

1. Polymorphism is one of the most important concepts in functional languages, allowing for strong typing with much code reuse. Polymorphism has a well-understood theory and practice.
2. Algorithms for compile-time polymorphic type inference exist, and although in theory are expensive (EXP-TIME complete) in practice are tractable.
3. Polymorphism has a clean semantics, and logical basis as universal quantification in the logic of types.
4. SML polymorphism (without overloading) gives every expression a principal type, thus making specification simpler.

[9] Some authors of the present Annex might disagree with some items of the following list.

5. It supports simple scope rules, and allows specifications to be combined without worrying about clashes caused by overloading.
6. If a suitably powerful module system is used, there are no problems with reusing identifiers.

However, introducing polymorphism in E-LOTOS would have the following drawbacks[10]:

1. It would imply the suppression of overloading facilities that currently exist in LOTOS, leading to to the problems mentioned in Section 5.2.9. It would go against the goal of compatibility with LOTOS defined in the scope of the New Work Item.
2. The existing type-checking algorithm for polymorphism is significantly more complex than the one for overloading (12 pages versus 4 pages in [1], 8 pages versus 3 pages in [41]). This would increase the complexity of both E-LOTOS static semantics and E-LOTOS type-checkers.
3. Moreover, type-checking for overloading can be performed in linear-time, whereas type-checking for polymorphism is exponential. There a risk that large, polymorphism-based protocol descriptions could not be type-checked in a reasonable amount of time.
4. Apart from the languages of the ML family, polymorphism is not used in any major programming or specification language. Although interesting ML features (e.g., union types, pattern-matching) have been reused in recent languages, this is not the case with polymorphism.
5. Lack of polymorphism has not been a usual complaint heard from people actually working with LOTOS. In this respect, it is worth noticing that none of the proposals for E-LOTOS enhancements described in Section 3 suggest to enbody polymorphism as a desirable feature. The reason for this is explained in the next items.
6. The genericity features of ACT-ONE already cover most of the functionalities provided by polymorphism, by allowing the definition of sorts and functions parameterized by one or several formal sorts. The same approach can be found in many algebraic languages, as well as in functional languages like Opal.

   Although the ACT-ONE syntax for actualization is not very user-friendly, it could be improved. Such changes would rather affect the module system than the type system of the core language itself.
7. If polymorphism is included in E-LOTOS, it is feared that it will be largely redundant with genericity features. For instance, a specifier wanting to define a stack (or a fifo queue) of parameterized items will be offered two possibilities: either declaring a polymorphic stack type, or declaring a generic stack type (using ACT-ONE-like generic modules or SML-like functors).

   Having two different mechanisms for the same concept will reduce reusability, as parts of code written with one mechanism might not be reusable in a context where the other mechanism is chosen.

[10] Some authors of the present Annex might disagree with some items of the following list.

Moreover, in such event, the predefined libraries of the base environment (e.g., sets, bags, etc.) might be defined twice: a polymorphic version, or a generic version, or both, might be available.

8. The generic features provided ACT-ONE go beyond the scope of polymorphism, by allowing sorts and functions to be parameterized by formal functions. This is also possible in SML, but not in the polymorphism framework; two different mechanisms (higher-order functions and functors) are provided instead.

   In SML, the redundancy issue mentioned in the previous item is all the more present, because *three* different approaches (polymorphism, higher-order functions, and functors) are available for the same purpose (defining objects parameterized by other objects).

   On the opposite, ACT-ONE and similar languages deal with genericity in a single, uniform approach. Formal sorts and formal functions are handled symmetrically, and the type system remains simple.

5.2.11 Records

We agree that E-LOTOS should allow record types to be easily defined and dealt with. It seems that there is some consensus about having the following features available for record types:

- It should be possible to declare a record by giving the list of its fields together with their types. The syntax could be either borrowed from algebraic languages (e.g., ACT-ONE, Opal, etc.) as in:

  type Point **is** Pt (x:real, y:real) **endtype**
  type Vector **is** Vec (x:real, y:real) **endtype**
  type Identity **is** Id (FirstName:string, Name:string) **endtype**
  type Person **is** Person (Id:Identity, Age:int) **endtype**

  or the ML syntax:

  datatype Point = Pt **of** {x:real, y:real}
  datatype Vector = Vec **of** {x:real, y:real}
  datatype Identity = Id **of** {FirstName:string, Name:string}
  datatype Person = Person **of** {Id:Identity, Age:int}

  Whatever the syntax, each record declaration also introduces a constructor declaration. In the above examples ‘Pt’, ‘Vec’, ‘Id’, and ‘Person’ are constructors.

- It should be possible to write record values[11] easily, and to nest these values arbitrarily. This could be done either with an algebraic syntax:

  Pt (1.0, 2.0)
  Vec (1.0, 2.0)
  Id (“Jane”, “Doe”)
  Person (Id (“Jane”, “Doe”), 33)

[11] called *aggregates* in Ada

or with the ML syntax:

```
Pt {x=1.0, y=2.0}
Vec {x=1.0, y=2.0}
Id {FirstName=“Jane”, LastName=“Doe”}
Person {Id=Id {FirstName=“Jane”, LastName=“Doe”}, Age=33}
```

- When writing record values, it should be possible to refer to field names (possibly with some permutation):

  ```
  Pt (x := 1.0, y := 2.0)
  Vec (y := 2.0, x := 1.0)
  ```

  or:

  ```
  Pt {x = 1.0, y = 2.0}
  Vec {y = 2.0, x = 1.0}
  ```

- Projections should be available for records, either as (overloaded) functions automatically declared from the record declaration, as it is the case in Opal:

  ```
  x (Pt (1.0, 2.0)) = 1.0
  x (Vec (1.0, 2.0)) = 1.0
  Name (Id (“Jane”, “Doe”)) = “Jane”
  ```

  or using the ML notation:

  ```
  #x {x=1.0, y=2.0} = 1.0
  ```

- Besides projections, pattern-matching should be available to access records. It is not wise to require that patterns should specify every field of the record. This is often impractical (say when one field out of many is being accessed) so we may wish to allow a record wild card pattern (noted “...”).

  For example, a projection function extracting the x-coordinate of a point could be defined as follows:

  ```
  function x_coord (p:Point) : real is
    let Point (x:real, ...) = p
    in x
  endfunc
  ```

- Also, it could be useful to have (functional) record *updaters*, i.e., shorthand notations that allow to make a copy of an already-existing record value together with a selective modification of the values of some fields:

  ```
  Pt { x=1.0, y=2.0 }.{ x=0.0 } = Pt { x=0.0, y=2.0 }
  ```

For example a function to zero the head of a list might be:

```
function zerohd (l:NatList) : NatList is
  case l in
    nil → nil
   | cons r → cons (r.{ hd=zero })
  endcase
endfunc
```

However, some technical points about records are still under discussion:

- Should records be a new type constructor (solution *A*), or should they simply appear as a particular case of the general union type mechanism described in Section 5.2.5 (solution *B*)?

  Solution *A* was preferred in ML, whereas solution B[12] was adopted in algebraic languages, such as ACT-ONE and Opal.

- If solution *A* is chosen, there are still two ways to include records in the base language. The first is to treat records as 'first class citizens' (Solution A_1), the second is to restrict records to the arguments of constructors and functions (Solution A_2).

  The difference in syntax between A_1 and A_2 is small, but makes a large difference to the type system:

  - Solution A_1 allows "anonymous" records, i.e., records which are not bound to a given constructor name. For instance, with the above sample definitions, expression '{x=1.0, y=2.0}' is legal and its type is '{x:real, y:real}'; it is different from expression 'Pt {1.0, 2.0}', the type of which is 'Point'.

    Problems occur when Solution A_1 is used together with record wildcard (for pattern-matching), or together with record updating. In a monomorphic type system, these present no difficulties but in a polymorphic type system it is no longer obvious what the principal type of a function with record wildcards or record updating should be.

    For example, it is impossible to give a principal type to the following functions:

    ```
    function getage { age=x, ... } is x endfunc
    function updateage { person=p, age=x } := p.{ age=x } endfunc
    ```

    because there are infinitely many record types having fields 'age' and 'person'. The problem is similar to the one of mixing overloading and polymorphism (except that overloading only allows for a finite number of principal types for an expression).

    There are a number of existing solutions to this problem in the literature, such as bounded polymorphism, row variables, and record extension types. However, these all add complexity to the type system.

[12] in which records are simply as *sums of products* types with a single variant

- Solution A_2 is more restrictive than Solution A_1, in the sense that record values can only occur as arguments of constructors or functions.

  In this respect, Solution A_2 and Solution B are almost identical. The main difference relies in the fact that Solution B allows functions and constructors with several arguments (as in existing LOTOS), whereas Solution A_2 is based upon the ML approach, in which functions and constructors have a single argument (this argument can be of type record when it is necessary to pass several arguments to the function or constructor).

  This has a mild consequence on the way of thinking the above notations (e.g., record aggregates, projections, wildcard patterns, etc.). In Solution A_2, these notations are attached to records themselves, whereas in Solution B they are rather attached to constructor invocations.

5.3 Abstract syntax(es) for the base language

In this section, we review some features which could form the basis of an abstract syntax of the data language for E-LOTOS.

In many cases, it is mentioned that (at least) two possible solutions exist and are still under discussion. These two approaches are respectively numbered 1 and 2. The first one is motivated by the idea of keeping, as much as possible, some compatibility with existing LOTOS, whilst turning to a functional approach as advocated in [26] or in the design of Opal. The second approach is motivated by the idea of being compatible with SML.

We tried to present these two approaches with similar notations, so as to ease the comparison by highlighting similarities and differences. We also put the discussion at the level of abstract syntax, rather than concrete syntax. The keywords and other notations used in the syntaxes are not firmly decided.

At least six important concepts should appear in the abstract syntax:

- Type expressions
- Data type declarations
- Patterns
- Function declarations
- Constant declarations[13]
- Expressions (called ‘value expressions’ in the LOTOS standard)

We present them in turn in the following subsections.

In addition, we may wish to allow an exception mechanism. One such is also described below.

5.3.1 Notations

In the sequel, we will use the following notations for various classes of identifiers used in the base languages (these classes correspond to the terminal symbols of the abstract syntax).

[13] In the first approach, there is no notion of constants: as it is the case in ACT-ONE, constants are simply functions of arity zero

symbol domain	*meaning*	*abbreviations*
Var	variable identifiers	X
Fun	operations identifiers	F
Con	constructors identifiers	C
Srt	sort identifiers	S
Pty	polymorphic type identifier	Y
Lab	field identifiers	L

We will also define the following classes of non-terminal symbols:

symbol domain	*meaning*	*abbreviations*
Typ	type expressions	T
Tdec	type declarations	D_t
Fdec	function declarations	D_f
Cdec	constant declarations	D_c
Exp	expression	E
Pat	patterns	P

Note: Since we are giving an abstract syntax, we will not specify a bracketing convention. This will be provided in a concrete syntax.

5.3.2 Type expressions

There are two possible approaches for type expressions:

1. In the first approach, a type expression is nothing but a sort identifiers (possibly an identifier of a predefined sort, such as **bool**, etc.). In such case, we have:

$$T \quad ::= \quad S$$

2. In the second approach, a type expression is either a polymorphic type variable, or a record type, or an instantiated polymorphic type:

$$\begin{array}{lll} T & ::= & Y \\ & \mid & \{L_1 : T_1, ..., L_n : T_n\} \\ & \mid & T_1 ... T_n \, S \end{array}$$

   Note: n is allowed to be 0, so $\{\}$ and S are types.

   Note: in the absence of polymorphism, the rule "$T ::= Y$" should be removed, and the rule "$T := T_1 ... T_n \, S$" should be replaced by "$T ::= S$".

   Note: in the presence of polymorphic record update, we should extend the type system with one of the treatments of polymorphic records discussed in Section 5.2.11.

5.3.3 Data type declarations

Symmetrically, there are two possible approaches for data type declarations:

1. In the first approach, a data type S is declared to be either synonymous of another data type S', or the union of p constructors $C_1, ..., C_p$ (each constructor C_i having n_i arguments of respective names L^i_j and respective types T^i_j, each C_i returning a result of type S):

$$\begin{array}{lll} D_t & ::= & \textbf{sort } S \textbf{ is} \\ & & \quad S' \\ & & \textbf{endsort} \\ & | & \textbf{sort } S \textbf{ is} \\ & & \quad C_1(L^1_1 : T^1_1, ..., L^1_{n_1} : T^1_{n_1}) \\ & & \quad ... \\ & & \quad C_p(L^p_1 : T^p_1, ..., L^p_{n_p} : T^p_{n_p}) \\ & & \textbf{endsort} \end{array}$$

   Note: one shall have $p \geq 1$ (at least one constructor).

   Note: as in LOTOS, a constructor C_i may have zero arguments (i.e., $n_i = 0$). If all constructors have arity zero, then S is an enumerated type.

   Note: as in LOTOS, constructors can be declared to be infixed.

   Note: of course, type definition can be recursive, i.e., some T^i_j can be equal to S.

   Note: in Opal, it is not allowed to declare a data type S identical to another data type S'.

2. In the second approach, a type constructor S is declared to be either synonymous of another data type T, or the union of p constructors $C_1, ..., C_p$ each with argument type T_i. Types may be parameterized by m type variables $Y_1, ..., Y_m$:

$$\begin{array}{lll} D_t & ::= & \textbf{datatype } Y_1 ... Y_m \ S == \\ & & \quad T \\ & & \textbf{endtype} \\ & | & \textbf{datatype } Y_1 ... Y_m \ S := \\ & & \qquad C_1 \textbf{ of } T_1 \\ & & \quad \mid ... \\ & & \quad \mid C_p \textbf{ of } T_p \\ & & \textbf{endtype} \end{array}$$

   Note: m can be equal to zero.

   Note: p must be greater or equal than 1.

   Note: alternatives of the form "T_i' C_i T_i''" could be added to handle infix constructors.

Note: in absence of polymorphism, the word "$Y_1 ... Y_m$" should be omitted.

Note: we can elide "**of** {}", writing C for C **of** {}.

Note: we should consider how to "cluster" mutually recursive datatype declarations.

5.3.4 Patterns

Patterns are a restricted form of expressions that are used in the context of pattern-matching. The structure of patterns must determined at compile-time in order to take advantage of efficient pattern-matching compiling techniques. Therefore, patterns can only contain constructors and bound variables[14]; functions and free variables[15] may not be used in patterns (see also the note below).

The declarations of patterns are similar in both approaches:

1. In the first approach, a pattern is either a named bound variable X of type T, or an anonymous variable taking any value of type T, or a constructor C applied to a list of pattern arguments $P_1, ..., P_n$ (the labels $L_1, ..., L_n$ of the constructor fields can be explicitly mentioned, possibly with some permutation, and the list of fields can be incompletely given, which is noted "**...**"). Finally the "**of**" notation can be used to coerce the type of a pattern:

$$\begin{array}{lcl} P & ::= & X : T \\ & \mid & \textbf{any}\ T \\ & \mid & C(P_1, ..., P_n[, \textbf{...}]) \\ & \mid & C(L_1 := P_1, ..., L_n := P_n[, \textbf{...}]) \\ & \mid & P_0\ \textbf{of}\ T \end{array}$$

   Note: As in LOTOS, constructors can have zero arguments (i.e., constants). They can also be infixed.

   Note: For convenience, we could relax the constraints by allowing non-constructor functions to be used in patterns, provided that the these functions are defined as finite terms containing only constructors and variables. In a very simple example, assuming that naturals are defined with '0' and 'succ', we could allow constant functions such as:

   function two **is** succ (succ (0)) **endfunc**

   and simple functions such as:

   function plus_2 (x:Nat) **is** Succ(Succ(x)) **endfunc**

   to be used in patterns.

2. The second approach essentially differs in its treatment of records, which can be used anonymously, and type annotations, which are optional.

   Making type annotations optional can make functions more compact, for example:

[14] i.e., variables declared inside the pattern

[15] i.e., variables declared outside of the pattern

```
function sum (x:Nat List) :=
  sum’ (x,0)
where function sum’ (x:Nat List, a:Nat) : Nat :=
  case x in
    nil → a
   | cons (cell : { hd:Nat, tl:Nat List }) → sum’ (cell.tl, a + cell.hd)
   endcase
  endfun
endfun
```

compared to:

```
function sum x :=
  sum’ (x,0)
where function sum’ (x,a) :=
  case x in
    nil → a
   | cons cell → sum’ (cell.tl, a + cell.hd)
   endcase
  endfun
endfun
```

Since module signatures require explicit types, type annotations are only optional within module bodies (and are hence localized).

On the other hand, it is often seen as good software engineering practice to give full type annotations to programs for purposes of readability and ensuring program correctness. For example, in the above, the reader has to be practiced at reading tail-recursive code before it is ‘obvious’ what the types of ‘x’ and ‘a’ are.

$$\begin{array}{rcl} P & ::= & X \\ & | & \textbf{any} \\ & | & CP \\ & | & \{L_1 = P_1, \ldots, L_n = P_n[, \textbf{...}]\} \\ & | & P \textbf{ of } T \end{array}$$

Note: if we allow “$X : T$” and “**any** T” as shorthand for “X **of** T” and “**any of** T”, and use the shorthand for tuples described in Section 6.10 then this is an extension of the previous proposal.

Note: the syntax for record patterns may lead to the problems with type inference described in Section 5.2.11.

Note: a rule of the form “$P ::= P_1\ C\ P_2$” could be added to handle infix constructors.

Note: we could add the ability to write C for $C\{\}$ in patterns. This makes programs easier to read, but at the cost of adding an ambiguity between variables and constant constructors. In SML this ambiguity is resolved in favour of constructors (although this means the semantics of a term is dependent on its context).

Note: Another possible shorthand (that exists in SML) would be to replace "$L_i = P_i$" in record patterns when L_i is syntactically identical to P_i.

Note: the syntax of patterns could be extended by in several ways:

- Boolean guards could be introduced in patterns, in order to have conditional patterns,
- Functions could be allowed in patterns, as well as constructors, provided that the value expressions defining the results of these functions are themselves patterns.

5.3.5 Function declarations

Symmetrically, there are two possible approaches for function declarations:

1. Either a function F is declared to have n formal parameters with names $X_1, \ldots, X_n$ and types $T_1, \ldots, T_n$, and to return a result of type T defined by some expression E:

$$\begin{array}{lll} D_f & ::= & \textbf{function}\, F(X_1 : T_1, \ldots, X_n : T_n) : T \ \textbf{is} \\ & & \quad E \\ & & \textbf{endfunc} \end{array}$$

2. Or, following the ML approach, a function F can be declared to have arguments in some pattern P and to return a result of type T defined by some expression E:

$$\begin{array}{lll} D_f & ::= & \textbf{function}\, F P [: T] \ \textbf{is} \\ & & \quad E \\ & & \textbf{endfun} \end{array}$$

Note: the type decorations on functions are optional in this version, and should be inferred.

Note: we should consider how to extend the language with mutually recursive function declarations.

5.3.6 Constant declarations

Constant declarations allow a way to bind variables to values, which are computed from expressions. A constant declaration just binds an expression to a pattern, similar to "**let**" declarations below:

$$D_c \quad ::= \quad \textbf{constant}\, P = E \ \textbf{endconst}$$

Note: these are called "value declarations" in SML.

Note: there are two possible dynamic semantics for constant declarations: either they are evaluated at declaration time, or when they are used. The first possibility fits with the call-by-value semantics, but means that it is possible for constant declarations to diverge. The second possibility means that constant declarations are just syntactic sugar for thunks (that is functions with a dummy argument), in the same way that constructor constants are syntactic sugar for constructor functions with a dummy argument.

5.3.7 Expressions

As regards value expressions, both proposals share a number of common points. A value expression can be either a constant denotation (i.e., a number, a string...), or a variable, or a "**case**" statement, or an "**of**" coercion:

$$
\begin{array}{lll}
E & ::= & \mathit{constant\_denotation} \\
 & | & X \\
 & | & \textbf{case } E_0 \textbf{ in} \\
 & & \qquad P_1 [\textbf{when } E_1] \rightarrow E'_1 \\
 & & \quad\ | \ldots \\
 & & \quad\ | P_n [\textbf{when } E_n] \rightarrow E'_n \\
 & & \textbf{endcase} \\
 & | & E_0 \textbf{ of } T
\end{array}
$$

Note: the evaluation of the "**case**" statement is deterministic. It selects the first pattern P_i (i.e., the smallest index i) such that E_0 matches P_i and the boolean guard E_i is true. The result returned is E'_i. Variables bound in P_i are visible in E_i and E'_i. If no alternative matches, the semantics is to be defined according to the discussion in Section 5.2.7 above.

Note: It might be desirable to allow several patterns in the same **case** alternative, e.g., "$P^i_1 | \ldots | P^i_{m_i} [\textbf{when } E_i] \rightarrow E'_i$". This would allow a proper factorization of identical processings. This is only possible if, for a given i, all the patterns P^i_j declare exactly the same set of variables, and with the same types.

Note: the "**case**" construct should have a symmetric counterpart in behaviour expressions.

However, the approaches differ on the following points:

1. In the first approach, an expression can be also a constructor application or a function application:

$$
\begin{array}{lll}
E & ::= & \ldots \\
 & | & C(E_1, \ldots, E_n) \\
 & | & C(X_1 := E_1, \ldots, X_n := E_n) \\
 & | & F(E_1, \ldots, E_n) \\
 & | & F(X_1 := E_1, \ldots, X_n := E_n)
\end{array}
$$

Note: constructors and functions without arguments, or infixed, are included by these rules.

Note: also, the tests for equality (and inequality) between two expressions are included here as a special case of function application.

Note: As in Ada, the names of the formal parameters of function and constructors can be mentioned in the application.

Note: This syntax can be enriched with shorthand notations such as testers, selectors, and updaters, as explained in Section 6.8.

2. The second approach mainly differs from the first (up to record syntax and syntactic sugar for tuples) by allowing anonymous records:

$$\begin{array}{rcl} E & ::= & ... \\ & | & C\,E \\ & | & F\,E \\ & | & \{L_1 = E_1, ..., L_n = E_n\} \\ & | & E_0.\{L_1 = E_1, ..., L_n = E_n\} \end{array}$$

Note: Two rules of the form "$E ::= E_1\,C\,E_2$" and "$E ::= E_1\,F\,E_2$" should be added to handle infix constructors and infix functions.

Note: we could add the ability to write C for $C\{\}$ in expressions. This is in line with the shorthand in type declarations and patterns, although differs from SML.

5.3.8 Exceptions

Exceptions give a way of flagging run-time errors, and (optionally) handling them. It is not obvious whether the core language should include exceptions or not.

If we do decide to include them, then we need a new class of terminals 'Exn' of exception identifiers, ranged over by N, and a new class of non-terminals 'NDec' of exception declarations, ranged over by D_n.

Exception declarations just declare a new exception name:

$$D_n \quad ::= \quad \textbf{exception}\ N$$

which can then be raised inside expressions:

$$\begin{array}{rcl} E & ::= & \ldots \\ & | & \textbf{raise}\ N \end{array}$$

Optionally, they can be handled by exception handlers:

$$\begin{array}{rcl} E & ::= & \ldots \\ & | & E_1\ \textbf{handle}\ N \to E_2 \end{array}$$

Note that exception handlers introduce extra semantic complexity to the core language, and require more complex interaction with the behaviour language.

5.4 Semantics

5.4.1 Static semantics

We should provide a static semantics for the core language. This should be formally defined, and should produce a language which is guaranteed to be free of run-time type errors.

At present, the form of the type system is not completely determined. This is largely due to the presence of overloading or polymorphism, which has yet to be decided, and to the problems of record subtyping.

For instance, the two approaches for the abstract syntaxes of expressions lead to different definitions of *type equivalence*:

- In the first approach, type equivalence is based on name equivalence (two type expressions are equivalent iff they have the same names). This approach is also found in languages like LOTOS, Pascal, Ada, etc.
- In the second approach, type equivalence follows SML approach and uses a combination of name equivalence and structure equivalence.

5.4.2 Dynamic semantics

We have to define the dynamic semantics of the core language in a *closing context*, that is one which contains bindings for all the free variables, function definitions and datatype definitions.

Also, unless polymorphism is selected, the dynamic semantics should be defined only for fully-actualized value expressions, i.e., expressions that not contain “generic” features. This is already the case in ACT-ONE.

As general requirement, the dynamic semantics of the core date language should:

- be formally defined,
- be deterministic, in the sense that the same expression always evaluates the same way in the same memory context,
- be strict, i.e., based on call-by value: the arguments of a constructor or a function have to be evaluated first, before the constructor or the function is applied
- ensure that any variable is initialized before it is used (this nice property already exists in LOTOS),
- and be implementable.

Whatever the approach chosen for the abstract syntax of the data language (see Section 5.3), the dynamic semantics should be fairly simple in both cases.

The precise definition of the dynamic semantics will depend upon the decision of introducing or not exceptions in E-LOTOS (see Section 5.2.7 above), as well as the semantics chosen for non-termination (see Section 5.2.6 above).

Also the dynamic semantics can be either an operational one, or a denotational one.

In the sequel, let V be a value, i.e., a term containing only constant denotations (e.g., numbers, etc.) and constructors. Let C be a context, i.e., a “storage” function mapping identifiers to their bindings.

In an operational approach, the dynamic semantics of expressions could be possibly defined by a predicate of the form:

- '$C \vdash E \Rightarrow V$' for 'in context C, E terminates with value V', and
- '$C \vdash E \Rightarrow$ **error**' for 'in context C, the result of E is undefined', if exceptions are not introduced in E-LOTOS,

  or,

  '$C \vdash E \Rightarrow$ **raise** t' for 'in context C, e raises exception t', if exceptions are included in E-LOTOS.

For example, the dynamic semantics of an expression E reduced to a variable X should be defined by the following rule:

$$\frac{C(X) = V}{C \vdash X \Rightarrow V}$$

which means that, if variable X has value V in context C, then expression X should evaluate to V in C.

In a denotational approach, the dynamic semantics of expressions is defined as a fixed-point. The result of the evaluation of an E in a context C is noted "$[[E]]_C$". It can be either equal to:

- a value V, if the evaluation terminates normally,
- a special value "**error**" (or "**raise** t") if the evaluation is undefined,
- "$\perp$" (the bottom value of the domain) if the evaluation does not terminate.

For example, the dynamic semantics of an expression E reduced to a variable X is defined by:

$$[[X]]_C = C(V)$$

6 Shorthand notation

6.1 Overview

In this section we present some shorthands which make specifications simpler to read and write. Some of these shorthands make use of features of the base environment and the module system.

6.2 Usual abbreviations and rich-term syntax

The most frequent complaint about ACT-ONE is the lack of abbreviated notations for:

1. integer, real and rational notation,
2. characters and strings.
3. list, sets, arrays, bags, etc.

Such abbreviations should be introduced in E-LOTOS. For instance, it should be possible to write abbreviated lists of character strings as simply as in SML:

```
["abc","DE"]
```

instead of ACT ONE verbose notations:

```
cons (a_l + (b_l + (c_l + <>)), cons (d_u + (e_u + <>), nil))
```

It is not decided yet whether such notations should be available only for the built-in types defined in the base environment, or could also be generalized to user-defined types.

Charles Pecheur's proposal for rich-term syntax [26] allows abbreviated notations to be used for both built-in and user-defined types. It provides mixfix notations like lists and strings as general syntax facilities rather than bound to built-in types.

For example, the list notation '[a, b, c]' is nothing but a macro-definition that expands to conventional prefix notation:

```
cons (a, cons (b, cons (c, nil)))
```

This notation is merely syntactic sugar; its expansion is done at a purely *syntactic* level, notwithstanding the meaning and type of 'a', 'b', and 'c'. Afterwards, type-checking is applied to the expanded expression.

The notation for empty lists introduces potential type-checking ambiguities. Some form of coercion (as the "**of**" clause of LOTOS) is probably needed:

```
[] of Bool_List
[] of Int_List
[] of Real_List
```

If operation overloading is allowed, the list notation facility can be extended to any user-defined type that embodies 'nil' and 'cons' operations, whatever the meaning of 'nil' and 'cons' is:

```
[1, 2, 3] of Int_List
[1, 2, 3] of Int_Set
[1, 2, 3] of Int_Sequence
```

Generic special notations may also be used for defining different kinds of characters and strings.

They might be used to define record projections and record updaters.

Rich-term syntax features causes no more penalty for language parsing than if they were restricted to built-in types.

We propose to investigate whether a rich term syntax is useful and practical.

6.3 Infix functions

Some infix functions are required, at least for the usual arithmetic, logical and comparison operations.

The existence of infix operations causes some syntactic issues, some of which have already been mentioned in Section 4.2.

For instance, it is not possible to have both user-defined infix operators (e.g., the binary "−") and user-defined prefix operators (e.g., the unary "−") at the same time. Otherwise, the language becomes ambiguous. For instance, expression "a b c" could be parsed either as "a (b (c))" or "(a) b (c)".

In order to allow static parsing of E-LOTOS, several approaches could be taken:

- The LOTOS philosophy could be kept as is. ACT-ONE allows only user-defined infix operators, whereas prefix operators have to be declared as functions with a single argument. Therefore, “a b c” is parsed as “(a) b (c)”. Furthermore all infix operations have equal precedence and associate to the left.
- Alternatively we could adopt the Miranda or Haskell approaches of having separate lexical classes for infix operators of different precedences and associativities.

We propose to find a syntax for infix operators which is simple and independent of context.

6.4 Conditional shorthands

We propose to introduce two “pseudo-operators” boolean operators named “**andthen**” (or “**andalso**”) and “**orelse**”, respectively defined as follows:

```
X andthen Y =
      case X in
        true → Y
      | false → false
      endcase

X orelse Y =
      case X in
        true → true
      | false → Y
      endcase
```

These shorthands are similar to those that exist in Ada or SML. They differ from the built-in “**and**” and “**or**” operators because they are not strict: the second argument is not evaluated before applying “**andthen**” or “**orelse**” and it might not be evaluated at all. These shorthands are useful in expressions such as:

```
(x <> 0) andthen (1 / x < epsilon)
(queue = nil) orelse (tl queue = nil)
```

We could also consider another shorthand, noted “⇒” and defined as:

```
X ⇒ Y =
      case X in
        true → Y
      | false → true
      endcase
```

6.5 “Let” statements

We can introduce a **let** statement as a particular form of of the **case** statement:

```
let P = E in E0 =
      case E in
        P → E0
      endcase
```

The semantics is the following: expression E is evaluated and matched against pattern P. The variables declared in P are bound to appropriate values and E_0 is evaluated.

The proposed “**let**” statements generalizes the “**let**” statement that exists in LOTOS behaviour expressions: it can be obtained when all pattern P_i have the form “$X_i : T_i$”. But the proposed “**let**” statements also supports pattern-matching, as in:

```
function norm (v:vector) : real is
  let Vec (x:real, y:real) = v
  in (x*x + y*y)
endfunc
```

For example a function to compute the middle of two points might be:

```
function middle (p1, p2:Point) : Point is
  let Point (x1, y1:real) = p1, Point (x2, y2:real) = p2
  in  Point ((x1 + x2) / 2, (y1 + y2) / 2)
endfunc
```

The **let** statement can be extended to support multiple patterns. In such case, it has to be expanded into nested **case** statements:

```
let P1 = E1, ..., Pn = En in E0 =
      case E1 in
        P1 →
           ...
           case En in
             Pn → E0
           endcase
           ...
      endcase
```

However, to remain compatible with existing LOTOS, we should ensure that all the variables declared in patterns $P_1, ..., P_n$ are pairwise distinct (which is not ensured by the translation into nested **case** statements, where the innermost declarations hide the others). Provided that additional verifications are included in the static semantics phase, it is not needed to include **let** in the core language.

If the language supports anonymous records (and hence tuples using the shorthand in Section 6.10) then the translation of multiple patterns can be:

```
let $P_1$ = $E_1$, ..., $P_n$ = $E_n$ in $E_0$ =
      case ($E_1$, ..., $E_n$) in
        ($P_1$, ..., $P_n$) $\to$ $E_0$
      endcase
```

Note: In SML, the "**let**" statements allows to declare many different objects (e.g., variables, types, functions, exceptions...), the visibility of which is local to the expression E_0. A "reasonable" subset must be identified for E-LOTOS.

6.6 "Assert" statements

For specification and verification purpose, it might be useful to introduce boolean assertions expressing invariants in E-LOTOS descriptions. This could be done using a "**assert**" statement defined as follows:

```
assert $E_1, \ldots, E_n$ in $E_0$ =
    case $E_1 \wedge \ldots \wedge E_n$ in
         true $\to$ $E_0$
      endcase
```

If all the boolean assertions $E_1, \ldots, E_n$ are true, then E_0 is evaluated. Otherwise, the result is undefined (because no alternative exists in the case statement), leading either to a run-time error or to an exception, depending on the semantics chosen.

Note: In the definition of **assert**, it would be possible to use **andthen** in place of "$\wedge$".

For instance, assertions can be used to express pre-conditions on the arguments of a (partial) function:

```
function pred (x : Nat) : Nat is
  asssert x > 0 in x – 1
endfunc

function modulo (x, y: Int) : Int is
  assert y <> 0 in ...
endfunc
```

or to post-conditions on the result returned by a function:

```
function sum (x, y : Nat) : Nat is
  let result:Nat = x + y
  in assert result >= x, result >= y
  in result
endfunc
```

Note: the expression "**assert false in** *E*", where *E* is any value expression, explicitly triggers a run-time error or the raise of an exception.

Note: if "**assert** ... **in** *E*" is introduced in the data part of E-LOTOS, then a similar construct "**assert** ... **in** *B*" might be desirable for the behaviour part.

6.7 If-then-else statements

We propose to introduce an "**if-then-else**" statement of the form:

```
if $E_0$ then $E'_0$
elsif $E_1$ then $E'_1$
...
elsif $E_n$ then $E'_n$
else $E'_{n+1}$
endif
```

Note: Keyword "**fi**" could be used in place of "**endif**".

Note: One may have $n = 0$ if there is no "**elsif**" parts.

Semantically, the "**if-then-else**" statement would be defined as syntactic sugar for a "**case**" statement:

```
if $E_0$ then $E_1$ else $E_2$ endif =
             case $E_0$ in
               true $\rightarrow E_1$
             | false $\rightarrow E_2$
             endcase
```

More complex "**if-then-else**" statements, with "**elsif**" parts, are expanded as follows:

```
if $E_0$ then $E'_0$
elsif $E_1$ then $E'_1$
...
elsif $E_n$ then $E'_n$
else $E'_{n+1}$
endif
        =
        if $E_0$ then $E'_0$
        else if $E_1$ then $E'_1$
        ...
        else if $E_n$ then $E'_n$
        else $E'_{n+1}$
        endif endif ... endif
```

Note: A proposal exists to introduce a similar "**if-then-else**" statement in the behaviour part of E-LOTOS

6.8 Testers, selectors, and updaters

Tester functions could be defined automatically to check the top-level constructor of a union type. For instance, we could have for each constructor C of a given type T, a predicate "*test_C (E)*" to check whether a value E of type T has the form "$E = C(...)$". Using the first approach to abstract syntax, tester functions can be defined as special forms of **case** statements:

```
function test_C (E:T) : bool is
    case E in
      C (...) → true
    | any T → false
    endcase
endfunc
```

Similarly, *selector functions* (or projections, or lookup functions) can also be derived automatically from data type definitions. Using the second approach to static semantics, we can define the projection #LE field L from a record E:

```
case $E$ of
   { $L$=$E'$, ... } → $E'$
 endcase
```

For example:

```
function append { front, back } :=
   case front in
     nil → back
   | cons cell → cons { hd=#hd cell,
                        tl=append { front=#tl cell, back=back} }
   endcase
endfun
```

Using the first approach to static semantics, the situation is almost identical if we only only consider record types (i.e., type with a single constructor) or if we assume that all the constructors of the same type have different names for their formal parameters. In such case, for each constructor C of type T and for each formal parameter $X : T'$ of C, we can define a selector function:

```
function select_X (E:T) : T' is
    case E in
      C (X:T' ...) → X
    endcase
endfunc
```

The Opal language is less restrictive and puts no restriction on the names of constructor arguments. The situation is slightly more complex since is leads to selector functions that can be overloaded and/or may contain **case** statements with several alternatives.

Similarly, *updaters* could be defined simply for record types (or even union types). For a given constructor C, the formal arguments of which are $X_1 : T_1, ..., X_n : T_n$, we can have the following notation:

$$E.\{X_{j_1} := E_{j_1}, ..., X_{j_p} := E_{j_p}\}$$

where $\{j_1, ..., j_p\}$ is a subset of $\{1, ..., n\}$. The result of this notation is the expression E in which the fields $X_{j_1}, ..., X_{j_p}$ have been given values $E_{j_1}, ..., E_{j_p}$ respectively. This notation is equivalent to:

$$\begin{array}{l}\textbf{let}\ C(V_1 : T_1, ..., V_n : T_n) = E\ \textbf{in} \\ \quad \textbf{let}\ (\forall i \in \{1, ..., n\}) V_i' : T_i = (\textbf{if}\ i \in \{j_1, ..., j_p\}\ \textbf{then}\ E_i\ \textbf{else}\ V_i) \\ \qquad \textbf{in}\ C(V_1', ..., V_n')\end{array}$$

6.9 Other record shorthands

The remaining shorthand notations in this section are mainly concerned with the SML-like notation for the core language, as defined in Section 5.3. These shorthand notations are motivated by the idea of introducing in E-LOTOS some of the convenient features that exist in SML.

As a shorthand to avoid writing many patterns '$L = X$', we could adopt the SML abbreviation of 'L' for '$L = L$'. For example an append function can be written:

```
function append { front, back } :=
  case front in
    nil → back
  | cons { hd, tl } → cons { hd, tl=append { front=tl, back } }
  endcase
endfun
```

6.10 Tuples

Using the second approach for abstract syntax, we can include tuples as syntactic sugar for records with integer fields names, for example:

```
int * bool == { 1:int, 2:bool }
(5, true) == { 1=5, 2=true }
```

This is the same abbreviation used by SML, although other languages like Caml, Haskell or Miranda, have tuples as primitive types. Making tuples abbreviations for records simplifies the core language. The proposed abbreviations are:

$(T_1 * \cdots * T_n)$ =
  { 1:T_1, ..., n:T_n }
(E_1, ..., E_n) =
  { 1:E_1, ..., n:E_n }
(P_1, ..., P_n) =
  { 1:P_1, ..., n:P_n }

For example, the List datatype can be defined as:

```
datatype 'a List :=
  nil
| cons of ('a * 'a List)
endtype
```

and a function to append two lists is:

```
function append (f,b) :=
  case f in
     nil → b
   | cons (h,t) → cons (h, append(t,b))
  endcase
end
```

6.11 Pattern-matching function declarations

As we have seen, it is very common for a function declaration to begin with a **case** statement. We can provide an abbreviation for this, similar to the pattern-matching syntax common to many functional programming languages. For example:

```
function
    append (nil,b) := b
  | append (cons (h,t), b) := cons (h, append (t,b))
endfun
```

The proposed abbreviation is (when X is not free in E_i):

```
function
    $F\ P_1 := E_1$
  | ...
  | $F\ P_n := E_n$
endfun =
  function $F\ X$ :=
    case $X$ in
        $P_1 \rightarrow E_1$
      | ...
      | $P_n \rightarrow E_n$
    endcase
  endfun
```

7 Modules

7.1 Overview

LOTOS has no general module facility, although ACT ONE **type** declarations can be seen as modules for datatypes. There is no export hiding (everything defined in the type is visible) nor import hiding (everything in the inherited types is accessible). Formal types and actualization provide for parametric modules.

One of the tasks of the E-LOTOS work item is to investigate modularization of LOTOS specifications.

In Section 7.2, we attempt to list some questions and and approaches relevant to the design of E-LOTOS modules. The answers to these questions should highlight the design principles for modules.

In Section 7.3, we present an approach based on SML modules, with some adaptations and extensions to fit E-LOTOS requirements.

7.2 Language design

We enumerate here a (possibly incomplete) list of questions which could be a base for design decisions. This list is divided into thematic sub-lists.

7.2.1 Separation between interfaces and modules

Question Q1.1: ACT ONE types do not have the notion of interface. Do we keep this approach or do we decide to introduce a separation between "declarations"[16] and "implementations"[17]? The first approach is convenient for specifying "in the small" whereas the second one is suitable for specifying "in the large".

Question Q1.2: If the answer to Question Q1.1 is 'yes', do we allow several modules (i.e., possible implementations) for a given interface?

Question Q1.3: If the answer to Question Q1.1 is 'yes', do we allow a given module to be viewed through different interfaces?

Question Q1.4: If the answer to Question Q1.1 is 'yes', do we force the specifier to write explicitly an interface for each module, or do we provide an (optional) mean to synthesize the interface from the module's description automatically?

7.2.2 Contents of modules (and interfaces)

Question Q2.1.a: What kind of objects should be contained in a module: types (or sorts), constructors, functions, exceptions, processes, channels (or gate types), etc.?

Question Q2.1.b: If the answer to Question Q1.1 is 'yes', what kind of objects should be declared in an interface?

Question Q2.2.a: Do we allow algebraic equations to be declared in modules?

Question Q2.2.b: If the answer to Question Q1.1 is 'yes', do we allow algebraic equations to be declared in interfaces?

Question Q2.3.a: Do we allow a module to contain definitions of other modules (and possibly interfaces), i.e., do we accept nested modules and interfaces?

Question Q2.3.a: If the answer to Question Q1.1 is 'yes', do we allow an interface to contain definitions of other interfaces (and possibly modules), i.e., do we accept nested interfaces and modules?

Question Q2.4: If the answer to Question Q1.1 is 'yes', and if types may appear in interfaces, how are types declared in interfaces? Is it possible to declare the type constructors together with the type in the interface?

Question Q2.5: If the answer to Question Q1.1 is 'yes', and if constructor declarations are allowed in interfaces, which profile informations about the constructors should appear in an interface?

[16] hereafter called *interfaces* or *signatures*

[17] hereafter called *modules* or *structures*

Question Q2.6: If the answer to Question Q1.1 is 'yes', and if function declarations are allowed in interfaces, which profile informations about the function should appear in an interface?

Question Q2.7: If the answer to Question Q1.1 is 'yes', and if process declarations are allowed in interfaces, which profile informations about the process should appear in an interface?

Question Q2.8: Where are modules (and interfaces) declared? Is it allowed to declare a module (or an interface) in a value expression? In a behaviour expression? Alternatively, should an E-LOTOS description be simply a flat collection of modules (and interfaces)?

7.2.3 Composition of modules (and interfaces)

Question Q3.1.a: It should be possible to reuse existing modules to build new modules. This determines a dependency relation between modules. Should the graph of this relation be a collection of DAGs[18] (i.e., modules are combined together in a hierarchical manner) or could this graph contain circuits (i.e., mutually recursive modules are allowed).

Question Q3.1.b: If the answer to Question Q1.1 is 'yes', same question for interfaces.

Question Q3.2.a: In the sequel, we will call "module combinators" those constructs that allow to build new modules. In ACT-ONE, for instance, module combinators are: import, renaming, and actualization. The list of desirable module combinators will discussed in further questions.

In ACT-ONE, applying a combinator can only be done by defining a new module (i.e., type) and by giving a new name to this module. Should we keep this approach, or should we also allow combinators to be applied without necessarily giving a name to the result? This would lead to "module expressions" (as in Ed Brinksma's thesis and the LOTOSPHERE proposal, for example), thus giving an "algebraic" flavour to module combinators.

Question Q3.2.b: If the answer to Question Q1.1 is 'yes', same question for interfaces.

Question Q3.3.a: If the answer to Question Q3.2.a is 'yes', do we allow module combinators to be composed in arbitrary manner, or do we identify some (syntactic or semantic) restrictions?

Question Q3.3.b: If the answer to Question Q3.2.b is 'yes', same question for interfaces.

7.2.4 Abstraction, hiding

Question Q4.1.a: Should it be possible to declare objects that are local to a module and not visible outside of the module? If so, should it be an *explicit* declaration in the module (e.g., with a "**local**" keyword) or an *implicit* one (e.g., all the objects declared in a module and not declared in the interface are local to the module)?

[18] Directed Acyclic Graphs

Question Q4.1.b: If the answer to Question Q1.1 is ‘yes’, same questions for interfaces. Would it make sense to have objects local to an interface?

Question Q4.2.a: Should we have an “hiding” combinator that would restrict the list of objects exported by a module by hiding some of them? If so, what kind of objects could be hidden (types, functions, processes, etc.)?

If so, how should the hiding be specified? There are at least four possibilities: an explicit list of objects to be hidden — or not to be hidden — or an interface defining the objects to be hidden — or not to be hidden.

In particular, do we allow to use an interface in order to “filter” the list of objects exported by a module: this would allow to have different *views* of the same module.

Question Q4.2.b: If the answer to Question Q1.1 is ‘yes’, same questions for interfaces.

Question Q4.3: Besides hiding, should we have a way to redefine an object exported by a module, i.e., to replace its implementation by another one (the redefinition feature is often used in object-oriented languages)?

Question Q4.4: Are the constructors of a given type systematically exported with this type? Or is it possible to hide some or all of them?

Question Q4.5: Is the the built-in, syntactic equality function for a given type systematically exported with this type? Or is it possible to hide it?

7.2.5 Importation, union, enrichment

Question Q5.1.a: We should certainly allow modules to import other modules. Should we allow *multiple* imports (i.e., a module can import several other modules)? Should we allow *transitive* imports?

Question Q5.1.b: If the answer to Question Q1.1 is ‘yes’, same questions for interfaces.

Question Q5.2.a: As a particular case of Question Q3.1.a, should the import dependency graph be a forest (hierarchical module imports), or a collection of DAGs (hierarchical module imports, but the same module can be imported twice using different paths), or any graph (possibly with circuits, which would allow mutually recursive imports)?

Question Q5.2.b: If the answer to Question Q1.1 is ‘yes’, same questions for interfaces.

Question Q5.3.a: When importing objects from a module, is it possible to specify a subset of objects to be imported, or not to be imported (*selective import*)? If so, how should this subset be specified (by an explicit list, an interface, etc.)?

Question Q5.3.b: If the answer to Question Q1.1 is ‘yes’, same question for interfaces.

Question Q5.4.a: Should it be possible to enrich an existing module with additional objects? If so, do we ensure *persistency*, i.e., do we guarantee that the semantics of the existing module will not be “corrupted” by the objects added?

Question Q5.4.b: If the answer to Question Q1.1 is ‘yes’, same questions for interfaces.

Question Q5.5.a: Do we require that all modules are *closed*, or do we accept that a module M_1 that does not import another module M_2 can refer to objects defined in M_2?

Even if all module definitions are closed, this issue might occur when selective import is used: for instance if we decide to import a boolean function such as "**and**", but not the **bool** type itself.

Question Q5.5.b: If the answer to Question Q1.1 is 'yes', same questions for interfaces.

Question Q5.6.a: When several modules are imported together (directly or transitively), what are the rules for solving potential name clashes?

Question Q5.6.b: If the answer to Question Q1.1 is 'yes', same questions for interfaces.

7.2.6 Renaming

Question Q6.1.a: Do we keep a renaming functionality for modules, as it exists in ACT-ONE? If so, what kind of objects should it be possible to rename (types, functions, processes, etc.)?

Question Q6.1.b: If the answer to Question Q1.1 is 'yes', same questions for renaming interfaces.

7.2.7 Object designation

Question Q7.1: Where several modules are used simultaneously, there is always a risk of name clashing for the objects exported by these modules. The renaming combinator of ACT-ONE solves this problem: by renaming objects when needed, name clashes can be avoided[19].

Do we want to keep this approach, or do we want an alternative solution, in which objects names can be prefixed with the name of the module (or interface) from which they are exported (*qualified notations*)? In the latter case, how do qualified notations support nested modules and transitive imports? Is there a mean to get rid of qualified notations?

7.2.8 Genericity

Question Q8.1: Do we allow generic modules? What kind of "formal" objects can be used to parameterize modules (e.g., types, constructors, functions, processes, equations, etc.)?

Question Q8.2: If the answer to Question Q1.1 is 'yes', same questions for interfaces.

Question Q8.3: How are "formal" objects specified? Using explicit lists of formal objects, as in ACT-ONE and Opal? Or using interfaces, as in the LOTOSPHERE proposal and SML? Can formal objects be grouped in modules (or interfaces)?

Question Q8.4: Does the actualization mechanism support partial instantiation?

[19] Renaming is also useful to give abbreviations for imported objects

7.2.9 Relationship with the external environment

Question Q9.1: We agree to have externally defined objects. How do we import external objects? Which objects (modules, interfaces, types, constructors, functions, processes, etc.) can be external? Do we allow that some objects in a module could be external, whereas the others not? Can we have external formal (or polymorphic) objects?

Question Q9.2: Do we want to support separate compilation in E-LOTOS? Is the state of the art sufficiently mature for this? What are the requirements laid by separate compilation on the modules system?

7.2.10 Compatibility with ACT-ONE

Question Q10.1: Which kind of compatibility is provided for LOTOS "**type**" declarations? In particular, to which extent does the new modules language support ACT-ONE's generic type declarations?

Question Q10.2: Which kind of compatibility is provided for LOTOS "**specification**" declarations?

Question Q10.3: In the behaviour part of E-LOTOS, do we keep the possibility to declare processes and types nested in process declarations, or do we only allow "flat" process definitions and use instead the abstraction facilities provided by the modules language?

7.2.11 Semantics

Question Q11.1: ACT-ONE's type declarations are fully handled at the static semantics level. Do we agree to maintain this property, so that the semantics of E-LOTOS modules should be a concern of static semantics only, with no implication on dynamic semantics?

Question Q11.2: ACT-ONE type declarations are fully orthogonal with the behaviour part and, to a large extent, orthogonal to the data part. This is ensured by the so-called *flattening* function that translates a LOTOS description structured in types (i.e., modules) into a "flat" LOTOS description without types. Do we intend to keep the same approach, so that the module part of E-LOTOS can be orthogonal to the data and behaviour part?

Question Q11.3: The semantics of ACT-ONE types is defined by a function that, for a given type T, computes the signature of T, i.e., the set of all (formal and non-formal) objects defined in T. Should we follow a similar approach for E-LOTOS — the signature function could be defined by induction (or possibly, as a fixed-point) on the combinators of module expressions and return the set of formal, hidden, and visible objects contained in a module or an interface — or are there other approaches?

7.3 Outline of an SML-based proposal

SML has a stable and powerful modules facility, and it should be possible to allow process declarations in structures and signatures. This would provide a flexible, well-understood module system.

One of the goals of E-LOTOS is to develop a modularization system, which should allow for export and import hiding, and for 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 **val** declarations. For abstraction and code re-use, signatures and generic modules are very useful.

In this section, we do not propose a precise syntax for modules, but the examples here give the flavour of what one might look like.

7.3.1 Abstraction and signatures

One of the most important properties of a module system is to provide a mechanism for abstraction. This should allow both import and export abstraction.

The example system described here is based on the SML modules system (and on McQueen's **abstraction** declarations and Sanella's EML). This provides separate declarations of module signatures and the module implementation.

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

```
signature Monoid :=
  eqtype M
  val 0 : M
  fun + : M * M → M
  eqns forall (x,y,z) →
    (0 + x) = x ;
    (x + 0) = x ;
    ((x + y) + z) = (x + (y + z))
  endeqns
endsig
```

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

```
module OnePoint : Monoid :=
  datatype M := 0 endtype
  function 0 + 0 = 0 endfun
endmod
```

Another is to coerce the natural numbers type (assuming an appropriate 'Natural' module):

```
module NatMonoid : Monoid :=
  open Natural
  datatype M == Nat endtype
endmod
```

Even though 'NatMonoid' is implemented using the natural numbers, this cannot be used outside the module definition, for example the following will not type check:

```
NatMonoid.0 = Natural.0
```

Signatures can be extended using **include** specifications, for example a simple signature for monomorphic lists is:

```
signature List :=
  include Monoid
  eqtype E
  fun inj : E → M
endsig
```

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 :=
  eqtype E
endsig
generic GenericList (E : EqType) : List :=
  open E
  datatype M :=
    0 | cons of (E * M)
  endtype
  function
    0 + ys := ys
   | (x cons xs) + ys := x cons (xs + ys)
  endfun
  function inj x :=
    x cons 0
  endfun
endgen
```

7.3.2 Equality types

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 datatype (for example sets might be represented as binary trees).

In a higher-order functional language, there are two sources of non-equality types: higher-order types, and abstract datatypes. If the E-LOTOS data language is restricted to first-order functions, then the only source of non-equality types will be data abstraction. For example, if we declare:

```
signature Set :=
  type ’a Set
  const empty : ’a Set
  fun insert : ”a * ”a Set → ”a Set
  fun delete : ”a * ”a Set → ”a Set
  fun member : ”a * ”a Set → bool
endsig
```

then we may wish to stop sets being compared for syntactic equality with ‘=’, since this would give away implementation details.

For this reason, it may be desirable to distinguish between types which do or do not admit equality. In a polymorphic language such as ML, this distinction can be reflected by a new class of equality type variables, which can range over types which admit equality. For example, the type of ‘=’ is:

fun = : ”a * ”a → bool

The advantages of making abstract datatypes equality types are:

1. the type system is simpler, and
2. all datatypes can be used in communications between processes.

The advantages of not making abstract datatypes equality types are:

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

For some applications (for example, implementing sets using binary trees), it is useful to provide a built-in ordering on all equality types. All equality types in Caml and Miranda have a built-in ordering.

We propose to make abstract datatypes non-equality types, and not to include an ordering on equality types.

7.3.3 Nested modules

The SML module system (on which much of the material in this section is based) is unusual in that it allows modules to be nested in other modules, for example all of the standard environment from Section 8 could be wrapped into one module with a signature including:

signature Standard :=
  module Boolean : Boolean
  module Character : Character
  ...
endsig
module Standard : Standard **external endmod**

The modules can the be accessed using nested module expressions, for example the ‘Bool’ type is ‘Standard.Boolean.Bool’.

This extension makes the module system more complex (as described below) and it is not obvious whether the extra complexity is necessary.

7.3.4 Sharing

Sometimes in building complex specifications it is necessary to build module heirachies 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 :=
  type Source
  type Target
  fun f : Source → Target
endsig
generic Compose (
  F : Morphism, G : Morphism,
  sharing F.Target = G.Source
) :=
  datatype Source == F.Source endtype
  datatype Target == G.Target endtype
  function f (x:Source) : Target :=
    G.f (F.f x)
  endfun
endgen
```

How to specify sharing between modules is a tricky problem, and one which we will postpone for the moment. It is not discussed below.

7.3.5 External declarations

One of the decisions of the Paris meeting was to support external declarations, interfacing to other specification or implementation languages. In the language described below, there is only a very simple **external** syntax allowing modules and generic modules to be implemented externally.

```
module ExtMonoid : Monoid external endmod
```

We may wish to extend this to allow arbitrary declarations to be external, or to have a richer **pragma** language. The syntax for **pragma**s should be as compatible as possible with existing annotations used by LOTOS tools such as CAESAR and TOPO.

Any module declared to be **external** has no formal dynamic semantics. In the E-LOTOS standard, we should give the dynamic semantics for most of the externally declared modules in the base environment (the exceptions are modules such as ‘Float’ which is too system-specific to model formally).

7.3.6 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) →
  (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 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).

7.3.7 Processes and modules

One of the requirements of the E-LOTOS work is to develop a module system for process declarations. This should be compatible with the module system used for data.

As a example of the power of generic modules, we can define generic modules to build a language of dataflow processes. A dataflow module is one with the signature:

```
signature Dataflow :=
  eqtype In
  eqtype Out
  proc Flow : { in : In Channel, out : Out Channel } → noexit
endsig
```

For example, a dataflow module for addition is:

```
module Add :=
  open Integer
  datatype In == Int*Int endtype
  datatype Out == Int endtype
  process Flow { in, out } :=
    in?(x,y) ; out!(x+y) ; Flow { in, out }
  endproc
endsig
```

Such dataflow modules can then be combined using generic modules, for example two modules can be composed in sequence as:

```
generic Sequence (A : Dataflow, B : Dataflow, sharing A.Out = B.In) :=
  datatype In == A.In endtype
  datatype Out == B.Out endtype
  process Flow { in, out } :=
    hide mid in
      A.Flow { in, mid } |[mid]| B.Flow { mid, out }
  endproc
endgen
```

and in parallel as:

```
generic Sync (A : Dataflow, B : Dataflow, sharing A.In = B.In, A.Out = B.Out) :=
  datatype In == A.In endtype
  datatype Out == B.Out endtype
  process Flow { in, out } :=
      A.Flow { in, out } |[in,out]| B.Flow { in, out }
  endproc
endgen
```

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

7.3.8 Relationship with existing ACT ONE specifications

The functional part of E-LOTOS will include algebraic specifications in signatures, as discussed in Section 7.3.6

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 :=
  eqtype M
  const 0 : M
  fun + : M * M → M
  eqns forall (x,y,z) →
    (x + 0) = x ;
    (0 + x) = x ;
    ((x + y) + z) = (x + (y + z))
  endeqns
endsig
module Monoid : Monoid external endmod
open Monoid
```

There is a strong resemblance between such specifications and ACT ONE datatype 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 **include** and **open** declarations and ACT ONE extended type specifications is not obvious,
3. the relationship between generic modules and ACT ONE parameterized types and type renaming is not obvious.

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

We propose to provide a formal correspondence between ACT ONE specifications and modules.

8 Base environment

8.1 Overview

The modules language for E-LOTOS has not been decided yet. However, for clarity, it will be necessary to give concrete examples of modules. For this purpose, we will often the syntax, vocabulary, concepts of SML modules in this section.

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, 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.
- If we allow polymorphism, we should specify whether the module will be defined using polymorphism, genericity or both.
- We should specify whether the types and functions contained in the module will be defined using the data 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.

8.2 Booleans

As an example standard module, we will give a possible module for booleans. The other standard modules will be similar, but probably more complex. Note that some of the Boolean module (e.g. defining ‘eq’ and ‘ne’) is purely for compatibility with the existing standard library.

```
signature Boolean :=
  datatype Bool := true | false endtype
  fun not : Bool → Bool
  fun and : Bool * Bool → Bool
  fun or : Bool * Bool → Bool
  fun xor : Bool * Bool → Bool
  fun implies : Bool * Bool → Bool
  fun iff : Bool * Bool → Bool
  fun eq : Bool * Bool → Bool
  fun ne : Bool * Bool → Bool
  eqns forall (x,y) →
    not false ;
    not true ⇒ false ;
```

```
      (not true) = false ;
      (not false) = true ;
      (x and true) = x ;
      (x and false) = false ;
      (x or true) = true ;
      (x or false) = x ;
      (x xor y) = ((x and not y) or (y and not x)) ;
      (x implies y) = (y or not x) ;
      (x iff y) = ((x implies y) and (y implies x)) ;
      (x eq y) = (x iff y) ;
      (x ne y) = (x nor y)
    endeqns
  endsig
```

The module body can be given by:

```
  module Boolean : Boolean :=
    function not x := x ⇒ false endfun
    function x and y := x andthen y endfun
    function x or y := x orelse y  endfun
    function x xor y := not (x = y) endfun
    function x implies y := x ⇒ y endfun
    function x iff y := x = y endfun
    function x eq y := x = y endfun
    function x ne y := not(x = y) endfun
  endmod
```

Booleans must be pervasive because they are used in the definition of shorthand notations (e.g., **andthen**, **orelse**, **if-then-else**, etc.).

8.3 Characters

For efficiency reasons, there is a demand for pragmatic ASCII-like characters and character strings, that can be implemented as bytes and byte strings. One solution would be to add a predefined ‘Char’ type. This should represent characters in an encoding-independent fashion. Such a solution is limited to one particular character set but allows for optimized implementations.

User-defined character sets can be declared as enumerated types. For instance, the LOTOSPHERE proposal suggested the following interface for characters:

```
  signature Character is
    datatype Char := ‘a’ | ‘b’ | ... | ‘z’ | ... endtype
    fun tolower : Char → Char
    fun toupper  : Char → Char
    fun isalpha : Char → Bool
    fun isdigit : Char → Bool
    fun islower : Char → Bool
    fun isupper : Char → Bool
    fun _>_ : Char, Char → Bool
    ...
  endsig
```

However, since LOTOS is a specification language, it should remain as open-ended as possible, so it is questionable to tie it to a given character set.

A more general solution is to define an open character syntax and support user-definable character sets and strings. This would allow for choosing a representation for any new characters and using it in compact string denotations. It should be also possible to map these new characters onto externally-defined character implementations.

We propose to include a standard module for characters. The precise syntax for expressions of this type, and the support for user-defined character sets is left as a subject for further investigation. We will investigate existing models of strings, such as Z.

8.4 Character strings

For pragmatic reasons, we wish to include a module containing a datatype for character strings. Semantically, this should be isomorphic to ‘Char List’, but will probably be treated differently by tools.

We propose to include a standard module for character strings, with explicit type conversion between strings and char lists. The syntax and semantics for expressions of these types are left as a subject for further investigation.

8.5 Binary data

Many protocols require the transmission and computation of binary data, and so E-LOTOS should provide a means for binary data to be easily manipulated.

We should keep the binary types that already exist in LOTOS standard library: Bit, Octet, HexadecimalNumber, DecimalNumber, BitStrings, OctetStrings, etc. We might also introduce 16-bit words, 32-bit words, *n*-bits words, etc., together with the usual arithmetical and logical operations.

We propose to include a standard module for binary data, with explicit type conversion between binary data and boolean lists, numbers, etc. The syntax and semantics for expressions of these types are left as a subject for further investigation.

8.6 Integers

E-LOTOS requires a datatype for integers. For compatibility with the ‘Nat’ existing type, we may require a datatype for natural numbers.

We propose to include a standard module for integers, with the semantics **Z**. We propose to investigate whether a standard module for naturals is required, with the semantics **N**. We will investigate existing models of integers, such as ACT ONE, SML and Scheme.

8.7 Rationals, floats and reals

This is an obvious need for types representing non-integer arithmetic, especially for real-time applications, but it is system-dependent and difficult to reason about. It is also an equality type which cannot be easily translated into an ACT-ONE specification.

The semantics of the ‘real’ datatype can be specified in three main flavours:

1. as finite floats (i.e. floating point approximations), as in most programming languages,
2. as mathematically pure real numbers, or
3. as mathematically pure rational numbers.

Solution 1 has the drawback that its semantics is tortuous and system-dependent. Even if floats are used in implementations, any resulting inaccuracy or overflow should be considered as irrelevant to the specification itself. Note that this vision requires that a distinction be made between division by zero (an error in the model) and division by a too small number (an error in the implementation). It also has problems with equational proofs about specifications (e.g. addition might not form a commutative monoid).

The difference between solutions 2 and 3 is that real numbers are not countable, cannot be fully represented by any term algebra and cannot be specified with equations. This is no major obstacle for incorporating them as a new built-in type (in replacement of ML’s reals). However, by doing this we introduce non-countable infinity in the semantic model, and this might have subtle consequences on its mathematical soundness (e.g. non-countably infinite branching processes).

On the other hand, rational numbers are countable, term-generated and specifiable by equations. They may also be implemented with exact representations, though most implementors might prefer to approximate them with floats. However it can be frustrating not to have the uniformity of reals, and the formal definition of irrational functions (exp, log, sqrt) becomes problematic with rationals.

We propose to include two standard modules: one for floats and one for rationals, and not to include a module for reals. The module for rationals will be given a formal semantics, but the float module will only be specified informally (since it is so system-dependent).

8.8 Lists

Lists are a heavily used built-in data type, and a standard module can be provided for them. If we are allowing polymorphism, then the datatype is:

```
datatype ’a List :=
  nil | ’a :: ’a list
endtype
```

If we are not allowing polymorphism, then the lists module will have to be generic. We should support syntactic sugar such as the list notation [a,b,c].

We propose to include a standard module for lists.

8.9 Sets and Bags

Sets are a very common data abstraction in specifications (the standard library of LOTOS contains a generic set type). Bags (or multisets) are also useful though less common. Sets can be provided in a simple way as a structure whose signature might include (if we are allowing polymorphism):

```
signature Set :=
  type ”a Set
  const empty : ”a Set
  fun insert : ”a * ”a Set → ”a Set
  fun delete : ”a * ”a Set → ”a Set
  fun member : ”a * ”a Set → Bool
  ...
endsig;
```

and similarly for bags. If we are not allowed polymorphism, the Set and Bag modules should be generic.

We propose to provide standard modules for sets and bags.

8.10 Arrays

Arrays are a very common feature of imperative programming, but they are a classic problem with functional programming, due to the need to avoid copying the whole array every time an array update is performed. However, for a specification language such as E-LOTOS, such implementation issues are less important.

Several approaches may be considered:

1. The solution adopted in SML/NJ (and inherited in PML) is to support updatable arrays using side-effects (arrays are similar to reference types). This implementation therefore has the same semantic problems as ref types (see Section 5.2.2).
2. Another option is to provide a datatype for arrays with functional update. This can be given a clean semantics treating arrays as any other data. It is also possible to simulate functional arrays using non-functional arrays and linked lists of updates, thus making code generation possible. This is similar to SML/NJ’s vector type constructor, but allows vectors to be updated functionally. SML/NJ has a special syntax for vectors, for example:

   ```
   const v := #[1,2,3] : int vector endconst
   ```

3. A third option is to use Wadler’s adaptation of Moggi’s *computational monads*, and to provide a monad for arrays. This may require higher-order functions, and monadic specification requires some sophistication on the part of the specifier.
4. A fourth option is to investigate whether a linear type system could be used, which would only allow expressions where arrays were not copied except by an explicit copying function.

The form of arrays given in SML/NJ assume that arrays are indexed by integers starting at 0. Trying to generalize this to any ‘index type’ is difficult, since it is not obvious what the definition of ‘index type’ should be, and this opens up the issue of subtyping.

We propose to provide a standard module for functional arrays, since these allow for simple specifications and semantics, although at a cost for code generation.

8.11 Associative arrays

One solution to the problem of only allowing integer array indices is to introduce a type constructor for *associative arrays*, as used, for example, in Perl. These are implemented as hash tables, but can be viewed as arrays where the index type is allowed to be any equality type. A suitable signature might include (if we are allowed polymorphism):

```
signature AssArray :=
  type (”a,’b) assarray
  const empty : (”a,’b) assarray
  fun update : (”a,’b) assarray * ”a * ’b → (”a,’b) assarray
  fun lookup : (”a,’b) assarray * ”a → ’b
  exception NotFound
  ...
endsig
```

Associative arrays have the same problems with copying as arrays do, and if associative arrays are to be allowed, they should have a similar semantics to arrays. Associative arrays are normally implemented by tools as hash tables.

We propose to provide a standard module for functional associative arrays.

8.12 Other modules

The base environment may wish to contain other modules for commonly used data and program structures, for compatibility with the existing LOTOS predefined types, and for standard mathematical structures such as partial orders, monoids, groups and so on.

9 Relationship with the behaviour part of LOTOS

9.1 Overview

This section describes the syntactic and semantic relationship between the datatypes language and LOTOS.

9.2 Symmetry between data and behaviour

We propose to establish a nice symmetry between the data and behaviour parts of E-LOTOS by having similar operators and shorthand notations in both of them.

For instance, **case** statements, **let** statements, and **if-then-else** statements should be available for data and behaviours as well, with the similar syntaxes and semantics.

If **assert** statements are introduced in the data part of E-LOTOS, then a similar construct **assert** should exist in the behaviour part.

If the pattern-matching function declarations presented in Section 6.11 are adopted in E-LOTOS, then pattern-matching process declarations should be available as well.

9.3 Value expressions in behaviour expressions

Data expressions can be used in many places in LOTOS specification:

1. in communications,
2. in guarding and selection predicates,
3. in instantiating parametric processes,
4. in **let** expressions, and
5. in **exit**.

We propose to replace all existing uses of ACT-ONE value expressions in LOTOS with value expressions from the data language.

9.4 Variable declarations in behaviour expressions

In the behaviour part of LOTOS, variables are bound to data expressions in:

1. in communications,
2. in declaring parametric processes and specifications,
3. in **let** expressions,
4. in **accept**, and
5. in **choice**.

As shown in Section 6.5, the usual **let** statement can be extended by allowing patterns in place of variable declarations, i.e., with syntax "**let** P **in** E" instead of "**let** X : S **in** E".

It is naturally tempting to replace other occurrences of variable declarations with patterns. The examples given below make use of features such as pattern-matching process declarations, polymorphic processes, type elision in variable declarations, and **accept** statements with pattern-matching:

```
process stack [in,out] (nil) :=
          in?x ; stack [in,out] ([x])
      | stack [in,out] (x::xs) :=
          in?y ; stack [in,out] (y::x::xs) [ ]
          out!x ; stack [in,out] (xs)
endproc
process query [db] :=
          search [db] ≫ accept
            SOME result ⇒ inspect (result) |
            NONE ⇒ complain
endproc
```

We propose to investigate whether some (or all) occurrences of variable declarations in the behaviour part of LOTOS could be replaced with patterns from the data language.

In the next section, we investigate the consequences of replacing variable declarations in experiment offers with patterns.

9.5 Communication pattern-matching

In LOTOS, two different forms of pattern-matching exist:

- Pattern-matching is used to define the semantics of functions. This form of pattern matching, which is more or less implicit in ACT ONE equations, will become fully explicit in E-LOTOS as **case** statements and patterns are introduced.
- Pattern-matching is also used in rendez-vous communications, especially when several !-offers are to be synchronized (this is called *value matching* in LOTOS).

However, both forms of pattern-matching are incompatible, resulting in awkward description style and inefficient implementations. For instance, let's consider a packet type defined as follows (using the first approach to abstract syntax):

```
type pdu is
   conreq (a:address)
   condis (a:address, r:reason)
endtype
```

In LOTOS, a process that receives a packet and performs different actions depending on the packet contents can be described as follows:

```
g ?p:pdu;
  (
  [discriminant (p) = conreq] → ...
  []
  [discriminant (p) = condis] → ...
  )
```

This approach does not involve pattern-matching at all. Packet discriminant and fields have to be extracted using projection functions. It is cumbersome and leads to inefficient implementations, since the same packet is accessed many times.

Therefore, another approach taking advantage of communication pattern-matching is often preferred:

```
g !conreq ?a:address; ...
[]
g !condis ?a:address ?r:reason; ...
```

However, this approach is not compatible with the previous one, because it requires that the packet has to be broken into its fields. Both specification styles can be used together only at the expense of auxiliary conversion processes in charge of breaking packets into fields and vice-versa. This is awkward and increases the whole complexity of the labelled transition system by adding interleaved actions, which are not pertinent for the actual system to be described.

Practically, the decision of choosing one particular specification style must be taken very early in the formal description process, and cannot be reversed easily.

It would be desirable to solve these issues in E-LOTOS by ensuring compatibility and symmetry between the pattern-matching facilities available in the data part and those available in communications.

In current LOTOS, experiment offers have the following syntax:

$$\begin{array}{lll} O & ::= & ?X : S \\ & \mid & !E \end{array}$$

We foresee at least two possible approaches:

- A first possibility is to replace query-experiment offers with patterns. The new syntax would become:

  $$\begin{array}{lll} O & ::= & ?P \\ & \mid & !E \end{array}$$

  Using this new syntax, traditional query-offers are available as particular cases in which the pattern P is equal to "$X : S$". Also, "anonymous" query-offers are obtained as "? **any** T".

  Although this extension would increase the possibilities for pattern-matching in communication, it should not change much the static and dynamic semantics of LOTOS rendez-vous. The only difference is that the notion of pattern-matching should be generalized to structured patterns instead of single variables.

- As LOTOS allows several experiment offer to be presented simultaneously on the same gate, one may wish to merge LOTOS ?- and !-experiment offers into a single "data structure". This would introduce a class of *extended patterns* in which !-offers and ?-offers can be combined.

  Therefore, a second possibility is to provide a new syntactic class of extended patterns, which extend data language patterns with input and output:

  $$\begin{array}{lll} O & ::= & \textbf{any} \\ & \mid & CO \\ & \mid & \{L_1 = O_1, \ldots, L_n = O_n [, \textbf{...}]\} \\ & \mid & O \textbf{ of } T \\ & \mid & ?P \\ & \mid & !E \end{array}$$

  In this approach, we can replace any LOTOS experiment offer list with an extended pattern. For instance, assuming that the language has ML-like tuples, the following replacements could be performed:

LOTOS action prefix	*Using an extended pattern*
g ?x:int ?y:int; B	g (?x:int, ?y:int) ; B
g ?x:int !(y+z); B	g (?x:int, !(y+z)) ; B

> Such an approach might lead to a semantics similar to that of the FP2 language [35]. In FP2, concurrent processes can synchronize and communicate using a rendez-vous mechanism, which generalizes LOTOS rendez-vous. In FP2, experiment offers are simply algebraic terms, possibly with bound variables. Two processes offering terms T_1 and T_2 respectively will synchronize if and only T_1 and T_2 can be unified; in such event, both processes will agree on a common value, which is the most general unifier of T_1 and T_2
>
> For instance, consider the synchronization of the processes:
>
> ```
> process P1 [G] is
>   G ! cons (1, cons (X1:int, cons (2, L1:int_list)))
> endproc
> process P2 [G] is
>   G ! cons (X2:int, cons (3, L2:int_list))
> endproc
> ```
>
> The synchronization is possible and, after synchronizing, the local variables of P1 and P2 will have the following values (where L is any value of sort ‘int_list’):
>
> ```
> X1 = 3   L1 = L   X2 = 1   L2 = cons (2, L)
> ```
>
> It is clear that LOTOS communication mechanisms (namely value matching, value transfer, and value generation) are special cases of the FP2 mechanism.
>
> On the opposite, the FP2 approach is more complex, because it requires unification where LOTOS only requires value equality. However, in FP2, there are syntactical restrictions on the form of terms, to ensure that synchronizations can be determined at compile-time and to avoid performing unification at run-time (which would have a high penalty in terms of performances). For instance, (non-constructor) functions are not allowed in communication patterns, except in outputs and free variables are only allowed in inputs.

In both existing proposals for introducing gate typing in E-LOTOS, experiment offers are labelled with “tags”. These tags could be either replaced with the names of constructor arguments or record field labels (depending on which abstract syntax is selected for expressions).

For instance, assuming that the data language has SML-like records, this would allow named fields, for example:

```
g { name = !”fred”, age = ?x } ; B1 |[g]| g { age = !27, name = ?y } ; B2
```

This would provide a clean semantics for tagged gates, whilst keeping compatibility with the datatypes language. If the datatypes language is extended with subtyping on records, this would provides a mechanism for partially constraining the values of a communicated record, for example:

```
g { name = !”fred”, title = !”Dr”, ... } ; B1 |[g]| g { age = !27, name = ?y ... } ; B2
```

Note that communication requires synchronization, and so any experiment offers have to be of equality type.

Also, pattern-matching requires the constructors are visible.

We propose to seek for compatibility between forms of pattern-matching in the data and behaviour parts. We propose to investigate whether LOTOS experiment offer lists can be extended to include more powerful forms of pattern-matching, taking in account the motivations for introducing gate typing in E-LOTOS.

9.6 Gates

So far, we have not mentioned how gates should be treated as data. There are (at least) four possibilities:

1. treat gates as first class citizens, by adding a gate module into the base environment, and making gates an equality type constructor,
2. as above, but making gates a non-equality type constructor,
3. making gates a separate kind (in the same way that exceptions are), or
4. having gates totally separated from the data language (as is the case currently).

If we were to allow gates as first-class citizens, then we could allow gate *expressions* not just gate *identifiers*. For example, a simple router might be (N.B., this is *not* currently legal LOTOS):

```
process Router (
  input : (string * int) Gate,
  table : (string, int gate) assarray
) :=
  input?(s,x) ;
  (lookup (table,s))!x ;
  Router (input,table)
endproc
```

This is a very powerful facility, and allows a simple coding of processes such as some ODP systems. However, allowing gate expressions rather than just gate identifiers into LOTOS is a major change, and should not be taken lightly.

If a gate type constructor is introduced, we should consider whether to allow it as an equality type or not. Since terms of equality type are allowed to be communicated between processes, we are effectively allowing the power of the π-calculus [18, 19, 16, 17] into LOTOS by making gates equality types. The π-calculus is very powerful, and can express some systems (such as some ODP systems) more easily than current LOTOS can. However, its semantics is more complex than that of LOTOS, and Pitts and Stark have shown that (in the presence of higher-order functions) finding denotational models for unique name generation is non-trivial [28].

By making gates a different ‘kind’ (in the same way that exceptions are a different ‘kind’ than expressions), we lose the power to treat gates as data, and we return to the more usual treatment of gates in LOTOS.

By making gates part of the datatypes language, whether by a gate type constructor, or a **gate** declaration, gate declarations can be combined with other declarations, for example in **where** clauses, or in modules and signatures.

We propose to investigate whether there are systems (such as some ODP systems, perhaps) which are easier to specify using gates as first-class citizens. If there are, then we will include a type constructor for gates. If there are not, then **gate** declarations will be separate declarations.

9.7 Process declarations

The same arguments hold for process declarations as for gate declarations, except that behaviours should definitely *not* be equality types.

We could introduce an ‘Behaviour’ type constructor, and view process declarations as declarations of values of Behaviour type. For example:

```
function buffer (in : ”a Gate, out : ”a Gate, contents : ”a List) : ”b Behaviour :=
  case contents in
    [ ] → in?y ; buffer (in, out, [y])
  | (x::xs) → in?y ; buffer (in, out, y::x::xs)
       [ ] out!x ; buffer (in, out, xs)
endfun
```

The ‘Behaviour’ type constructor is similar to the ‘event’ type constructor of Reppy’s CML [31, 32], or the computation type constructor of Moggi’s monadic metalanguage [22] and Jeffrey’s CMML [13]. There has recently been much work on the semantics of higher-order processes, such as [11, 13, 37]. These processes provide new means to describe object-oriented or ODP applications.

However, such languages are still the subject of research, and the semantics of higher-order processes is much more complex than that of first-order. For example, in the presence of higher-order processes and unique name generation, the obvious definition of bisimulation is no longer satisfactory, and one has to use the more complex context bisimulation.

We propose not to allow higher-order processes, and to keep the distinction between constant, function and process declarations.

9.8 Process functionality and exceptions

The static semantics of LOTOS defines the notion of *process functionality*. If exceptions are introduced in the data language of E-LOTOS, it might be nice to unify the notions of functionality for processes and exceptions for functions into a single concept.

By doing so, it should be possible to present the static semantics of the behaviour language in a similar style to the data language. This would give users a uniform type system for data expressions and processes, and (if implicit typing is allowed) would allow users to leave the types of some variables to be deduced by the type system.

For example, using a style similar to the SML static semantics, the type of **choice** is given by:

$$\frac{C \vdash p \Rightarrow (VE,\sigma) \quad C+VE \vdash B \Rightarrow \textbf{exit}\ \tau}{C \vdash \textbf{choice}\ p\ []\ B \vdash \textbf{exit}\ \tau}$$

This type system should be compatible with the ‘func’ function for determining functionality.

However, we should also consider that process functionality is the only form of gate typing that exists in LOTOS (process functionality types the offers emitted on δ gates). Therefore, process functionality could be related not only to exceptions, but also to rendez-vous and gate typing.

9.9 Static semantics

LOTOS allows for constraint-based programming, which requires that any data communicated on any channel be comparable for equality. Fortunately, we already has the concept of an ‘equality type’, so we just have to restrict communications to being of equality type.

There may, however, be issues about ensuring the type system is sound, for the same reason as for references: polymorphic processes may create problems.

We propose to find a static semantics for the behaviour part of LOTOS which is compatible with existing specifications, and with the data language.

9.10 Dynamic semantics

We propose to define an operational semantics for the behaviour part of E-LOTOS.

There should be a formal definition of the relationship between the dynamic semantics of the data and behaviour parts of E-LOTOS.

Also, unless polymorphism is selected, the dynamic semantics should be defined only for fully-actualized behaviour expressions, i.e., expressions that not contain “generic” features. This is already the case in LOTOS.

9.10.1 Layers in the dynamic semantics

The dynamic semantics of LOTOS has two layers. An axiomatic semantics is provided for the data part of the language, on top of which an operational semantics is provided for the behaviour part.

If the dynamic semantics of the E-LOTOS data language is denotational, two layers will also be necessary, since the dynamic semantics of the behaviour part will be defined operationally.

If the dynamic semantics of the E-LOTOS data language is operational, we may have one or two layers, depending whether both sets of derivation rules are merged or not. In any case, we should not allow the derivations for the data part to be freely interleaved with those of the behaviour part, since this would result in semantic problems such as exponential state space explosion (this would be “small step semantics”). Derivations for the data part should be performed in priority, so that value expressions are completely evaluated before considering the possible derivations for the behaviour part (this is “big step semantics”).

9.10.2 A possible dynamic semantics for exceptions

If exceptions are introduced in E-LOTOS, the semantics for expressions should be of the form $E \Rightarrow V$ (successful termination) or $E \Rightarrow$ **raise** N (raising an exception). Then, the semantics of **exit** can be:

$$\frac{E \Rightarrow V}{\textbf{exit}\,V \xrightarrow{\delta\langle V\rangle} \textbf{stop}} \qquad \frac{E \Rightarrow \textbf{raise}\,N}{\textbf{exit}\,E \xrightarrow{\chi\langle N\rangle} \textbf{stop}}$$

This uses an operational semantics for raising exceptions based on the proposed semantics of generalized termination. For example:

$$\textbf{exit}(1+2) \xrightarrow{\delta\langle 3\rangle} \textbf{stop} \qquad \textbf{exit}\ (1 \text{ div } 0) \xrightarrow{\chi\langle \text{Zdiv}\rangle} \textbf{stop}$$

where ‘Zdiv’ is the exception raised by division by zero.

Similarly, the semantics of guards is:

$$\frac{E \Rightarrow \text{true} \quad B \xrightarrow{a} B'}{[E] \rightarrow B \xrightarrow{a} B'} \qquad \frac{E \Rightarrow \textbf{raise}\,N}{[E] \rightarrow B \xrightarrow{\chi\langle N\rangle} \textbf{stop}}$$

For example:

$$[1{+}1 = 2] \rightarrow \textbf{exit}(1{+}2) \xrightarrow{\delta\langle 3\rangle} \textbf{stop} \qquad [1 \text{ div } 0 = 2] \rightarrow \textbf{exit}(1{+}2) \xrightarrow{\chi\langle \text{Zdiv}\rangle} \textbf{stop}$$

Note that this semantics means that any expression which raises an exception will have that exception propagated to the process level. The process construct this affects most is **choice**, since selection predicates can now raise exceptions. For example:

$$\textbf{choice}\ \text{x}\ [\ ]\ [1 \text{ div x} = 2] \rightarrow \textbf{exit}(1{+}2) \xrightarrow{\chi\langle \text{Zdiv}\rangle} \textbf{stop}$$

This requires a syntax for exception handling in LOTOS processes, compatible with the exceptions used in the data language.

This semantics does mean that data evaluation can cause behaviour transitions, and in particular resolve some choices. This may have undesirable effects on the semantics of LOTOS. We have to decide whether the programming power of exception handling is worth the nastier semantics.

9.10.3 Possible dynamic semantics for non-termination

One problem with this treatment of the interaction between the dynamic semantics of the functional and behavioural parts is divergence (i.e., non-terminating functions). There are several possible ways of taking divergence into account in the dynamic semantics:

- In a first approach, we might consider that a divergence in the data part does not lead to any observable transition in the behaviour part. In this approach, divergence is treated similarly in the behavioural part and the functional part: by a term with no reductions. For example if we define:

  function loop () = loop () **endfun**
  process Loop := Loop **endproc**

  then the following behaviours are strongly bisimilar:

[loop ()] → **exit**
Loop
stop

Although this semantics gives the same treatment to divergence in the functional and behavioural parts, it presents problems for simulators, which may produce different results from simulating the following:

([loop ()] → **exit**) [ ] a;**stop**
Loop [] a;**stop**
a;**stop**

since the first two might diverge while the last communicates.

- For this reason, we might wish to distinguish between divergence (considered as a run-time error) and deadlock. This could be done by using some technology from divergence-sensitive bisimulation, and extending the operational semantics of LOTOS with a 'can terminate' predicate which can distinguish between deadlock and divergence. This predicate, noted "$B\Downarrow$" is true iff it can be proven in a finite number of steps that B terminates. It is defined by induction on the abstract syntax of behaviour expressions. For example:

  $$\frac{E \Rightarrow \text{true} \quad B\Downarrow}{([E] \rightarrow B)\Downarrow} \qquad \frac{E \Rightarrow \text{false}}{([E] \rightarrow B)\Downarrow} \qquad \frac{B_1\Downarrow \quad B_2\Downarrow}{(B_1 \; [] \; B_2)\Downarrow} \qquad \frac{B\Downarrow}{P\Downarrow}[P := B]$$

  Using this semantics, we can distinguish between Loop and **stop**, since **stop** terminates, whereas Loop does not.

  One problem with this semantics is that there is no obvious treatment of **choice**, since (in order to keep the correspondence between **choice** and [ ]) we would have to adopt the rule:

  $$\frac{\forall \vdash V \Rightarrow T \,.\, B[V/X]\Downarrow}{(\textbf{choice}\; X : T \; [] \; B)\Downarrow}$$

  This requires a universal quantification over all values, and may present in semantic problems.

  Divergence-sensitive bisimulation is slightly more complex than bisimulation, so it may be appropriate to present divergence-insensitive bisimulation as the standard semantics, and provide divergence-sensitive bisimulation as an annex (similar to the treatment of testing equivalence in the existing standard).

We propose to provide a formal link between the dynamic semantics of the functional and behavioural parts of E-LOTOS. The precise treatment of divergence is a topic for further work.

10 Relationship with other E-LOTOS proposals

10.1 Time

The ET-LOTOS proposal uses a datatype of time, using time expressions with free variables. To combine time and data, the time datatype should be part of the data language.

It should be possible to treat the time extensions to LOTOS as a module containing a ‘time’ datatype, and some operations on time and processes.

10.2 Termination

The outline of the dynamic semantics in Section 9.10 makes use of generalized termination to give a semantics for exceptions.

If exceptions are used in the data language of E-LOTOS, the syntax used for raising and handling exceptions should be compatible with the syntax in the behavioural language.

10.3 Typed gates

There is a strong link between the data language and the two proposals for typed gates. The concept of communication pattern-matching suggested in Section 9.5 — although it has to be worked out — might extend (possibly replace) gate typing proposals, provided that all the practical motivations behind gate typing are satisfied.

10.4 Modules

One of the tasks of the E-LOTOS work item is to investigate modularization of LOTOS specifications.

We have proposed adding a module system for data, which could be extended to include processes.

10.5 Mobility

There is a link between the notion of gate as first-class citizen, and the proposal for mobile LOTOS.

Acknowledgements

Thanks to Simon Thompson, for comments on the Paris output document, some of which have been incorporated into this text.

Thanks to Philippe Schnoebelen for his insightful comments about FP2.

This work was discussed at the June 1995 Warsaw meeting of COST 247, and we would like to thank the organizers of COST 247 for giving us a meeting place, and to the WG1 participants for providing constructive remarks and participating in the discussion.

This work has been carried out in the context of ISC-CAN-65 EUCALYPTUS 2, COST 247, and EC BRA 7166 CONCUR 2.

When used as the name of a programming language, Miranda is a trademark of Research Software Limited.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[2] T. P. Baker. A one-pass algorithm for overload resolution in Ada. *ACM Transactions on Programming Languages and Systems*, (4):601–614, April 1982.

[3] E. Brinksma and G. Leih. *Enhancements of LOTOS*. In T. Bolognesi, J. van de Lagemaat, and C. Vissers, editors, *LOTOSphere: Software Development with LOTOS*, pages 453–466. Kluwer Academic Publishers, 1995.

[4] T. Bolognesi, J. van de Lagemaat, and C. Vissers, editors. *LOTOSphere: Software Development with LOTOS*. Kluwer Academic Publishers, 1995.

[5] Ed Brinksma. *On the Design of Extended LOTOS, a Specification Language for Open Distributed Systems*. PhD thesis, University of Twente, November 1988.

[6] Nachum Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1):69–115, February 1987.

[7] Nachum Dershowitz. 33 Examples of Termination. In *Term rewriting*, Hubert Comon and Jean-Pierre Jouannaud, editors. Volume 909 of *Lecture Notes in Computer Science*, Berlin, 1995. Springer Verlag.

[8] A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. Algebraic programming in OPAL. *Bulletin EATCS*, (50):171–181, June 1993.

[9] ISO. ESTELLE — A Formal Description Technique Based on an Extended State Transition Model. International Standard 9074, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[10] ISO 8807. *LOTOS—A formal description technique based on the temporal ordering of observational behaviour*, 1989.

[11] M. Hennessy. A fully abstract denotational model for higher-order processes. *Information and Computation*, 112(1):55–95, 1994.

[12] P. Hudak, S. L. Peyton Jones, P. Wadler, et al. A report on the functional language Haskell. *SIGPLAN Notices*, 1992.

[13] Alan Jeffrey. A fully abstract semantics for a concurrent functional language with monadic types. In *Proc. LICS 95*, pages 255–264, 1995.

[14] José A. Manas. Modular LOTOS. Annex G of ISO/IEC JTC1/SC21/WG1 N1349 Working Draft on Enhancements to LOTOS, October 1994.

[15] David McQueen. Modules in standard ML. LFCS Report ECS-LFCS-86-2, Department of Computer Science, Edinburgh University, 1986.

[16] Robin Milner. The polyadic π-calculus: a tutorial. In *Proc. International Summer School on Logic and Algebra of Specification*, Marktoberdorf, 1991.

[17] Robin Milner. Functions as processes. *Math. Struct. in Comput. Science*, 2:119–141, 1992.

[18] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes I. *Information and Computation*, 100(1):1–40, September 1992.

[19] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes II. *Information and Computation*, 100(1):41–77, September 1992.

[20] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[21] Robin Milner, Mads Tofte, and Robert Harper. *Commentary on Standard ML*. MIT Press, 1991.

[22] Eugenio Moggi. Notions of computation and monad. *Inform. and Computing*, 93:55–92, 1991.

[23] Harold B. Munster. Comments on the LOTOS Standard. NPL Technical Memorandum DITC 52/91, National Physical Laboratory, Teddington, Middlesex, UK, September 1991.

[24] 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.

[25] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[26] 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.

[27] P. Pepper. The Programming Language Opal — Implementation Language. Technical Report, Fachbereich Informatik, Technische Universität Berlin, February 1994.

[28] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Proc. MFCS 93*, pages 122–141. Springer-Verlag, 1993. LNCS 711.

[29] Juan Quemada, editor. Initial Draft on Enhancements to LOTOS. ISO/IEC JTC1/SC21/WG1 N8023 Project 1.21 Q1/48.6, November 1993.

[30] R. Roth, J. de Meer, and S. Storp. *Data specifications in modular LOTOS*. In T. Bolognesi, J. van de Lagemaat, and C. Vissers, editors, *LOTOSphere: Software Development with LOTOS*, pages 467–479. Kluwer Academic Publishers, 1995.

[31] J. H. Reppy. A higher-order concurrent language. In *Proc. SIGPLAN 91*, pages 294–305, 1991.

[32] J. H. Reppy. *Higher-Order Concurrency*. Ph.D thesis, Cornell University, 1992.

[33] Don Sanella. Formal program development in extended ML for the working programmer. LFCS Report ECS-LFCS-89-102, Department of Computer Science, University of Edinburgh, 1989.

[34] W. Schulte and W. Grieskamp. Generating efficient portable code for a strict applicative language. In J. Darlington and R. Dietrich, editors, *Declarative Programming*. Springer Verlag, 1992.

[35] Ph. Schnoebelen and Ph. Jorrand. Principles of FP2. Term Algebras for Specification of Parallel Machines. In J. W. de Bakker, editor, *Languages for Parallel Architectures: Design, Semantics, Implementation Models*, chapter 5, pages 223–273. Wiley and sons, 1989.

[36] Silke Storp. Integration of Standard Data Types into LOTOS. Master's thesis, Offene Kommunikations Systeme (OKS), Berlin, June 1991.

[37] Bent Thomsen. *Calculi for Higher-Order Communicating Systems*. Ph.D thesis, Imperial College, 1990.

[38] The OPAL Tutorial. Jürgen Exner. Technical Report 94-9, Technische Universität Berlin, May 1994.

[39] David A. Watt. *Programming Language Concepts and Paradigms*. International Series in Computer Science. Prentice-Hall, New-York, 1990.

[40] Åke Wikström. *Functional Programming using Standard ML*. Prentice Hall, 1987.

[41] R. Wilhelm and D. Maurer. *Les compilateurs, théorie, construction, génération*. Manuels informatiques Masson. Masson, Paris, 1994.