A Proposal for Data Types for E-LOTOS

Charles Pecheur, Université de Liège (B), Systèmes et Automatique[1].
<pecheur@montefiore.ulg.ac.be>

Abstract

This is a proposal for two enhancements to the data part of LOTOS. The first one is a revision of the core part of the data model and language, leading to a more functional view of data evaluation and better computational properties. The second one is collection of convenient syntax extensions for denoting data terms, viewed as general purely syntactical extensions.

This needs to be combined with other proposals in order to offer a full blown language for LOTOS data specification. To help this, we present a general framework, in which this proposal is put in perspective. We also discuss the integration of the work presented here with other ongoing work on LOTOS enhancements.

This document will be contributed to the ISO working group on LOTOS enhancements, to be appended as annex I of [RD94b].

1. INTRODUCTION

1.1. Motivation

Current LOTOS is equipped with a data type definition language based on algebraic semantics, which is unanimously denounced as the main weakness of the language. Accordingly, the ISO JTC1/SC21/WG1/Q48.6 Work Item on "LOTOS Enhancements" has recognized data type enhancements as one of its tasks. The Revised Draft issued from the January meeting in Madrid [RD94] considers the following directions for enhancements:

a. Built-in types

b. Composed types

c. Partial functions or subtyping

d. Constructive types

e. Interface with data types in other languages

Another enhancement direction that is of vital importance for data type is modularity: in current LOTOS, any type definition can be jeopardized by a faulty definition in any other type.

The present document makes proposals for several aspects of new data types in LOTOS that meet these requirements. It does not intend to be a completely self-contained new language definition, but only to propose specific enhancements and to comment their relation with other existing proposals and related work. The proposed enhancements cover two aspects :

[1]This work is supported by the CEC under EC-CA Project 001:76099 "EUCALYPTUS".

- More constructive type definitions, based on a syntactical and semantic distinction between constructors and functions. This allows for computing data values more easily, while still offering a high abstraction level, strong expressive power and rigorous semantics. It also enhances modularity by allowing requirements for type preservation to be not only expressed but enforced and checked statically.
- A richer term syntax, allowing more concise and readable notations for sequences (`[a,b,c,d]`), selectors (`people.name`, `matrix.(1,1)`), modifiers (`people<-name-"Charles"`), etc. These syntactical forms are available for operations in any user-defined type, like infix operations in current LOTOS; this is in contrast with other proposals where those notations are restricted to some built-in types only.

The first proposal addresses point (d) and has consequences on modularity; point (c) is also concerned. This is discussed in more details in section 4.4. The second proposal is related to point (a) and point (b) above, in that it provides facilities that are usually tied to specific types.

1.2. Conventions

In what follows, "ACT ONE" will refer to the data type definition part of current LOTOS (i.e. according to [IS89]), whereas our proposal will be designated as "E-LOTOS".

Syntax definitions will be given using common BNF conventions, with "|" for alternatives, "*" for repetitions ("+" for non-empty repetitions) and "[…]" for optional parts. Constant tokens will be surrounded by double quotes.

Non-scalar values (sequences, tuples) will often be captured under a single symbol. We will write x_I for $(x_1, \ldots, x_n)$, with the assumption that $I = (1, \ldots, n)$. More generally, given a context $C[]$, we will write $(C[x])_I$ for $(C[x_1], \ldots, C[x_n])$. As expected, $\sum x_I = x_1 + \ldots + x_n$, $\prod x_I = x_1 \times \ldots \times x_n$, etc. The empty sequence is denoted () and may be omitted if no confusion arises.

2. A FRAMEWORK FOR E-LOTOS

Several well distinguished kinds of extensions are currently considered for E-LOTOS. To help their integration we present a general framework for the definition of E-LOTOS. We only consider the data part here, but suggest that a similar approach be used for the behaviour part.

The definition of the data part of E-LOTOS consists of the following steps:

- a data model,
- a core language syntax and semantics,
- an extended syntax for function definitions,
- an extended syntax for data terms,
- a module model and declaration syntax,
- built-in types,
- syntax extensions for module schemes.

This layering is compatible with the structure proposed in [G] and with other pending proposals for E-LOTOS. The rest of this proposal focuses on the two first steps. The layered approach allows for some uncoupling in the definition of different layers.

Each layer adds a complementary level of abstraction to the language. The semantics for this layer may or may not be expressed in terms of lower layers, but in any case, any implementation compatible with this semantics is acceptable. For example, the semantics for extended function definition forms like **case-of** or **if-then-else** are defined via the generation of hidden functions. Admittedly this may become heavy to handle if new functions are effectively generated, but more efficient compatible implementations may exist.

Each layer is detailed in the following sections.

2.1. Data model

Before defining any syntax, the first step describes the semantic structures that will capture the meaning of data definitions. Those structures are used in the behaviour part for deciding guards and synchronizations. One of the requirements for E-LOTOS is the possibility of interfacing with other data definition languages such as ASN.1 or SDL. To support this, the meaning of those languages must be expressible within the model, which therefore should be kept as general as possible. This proposal uses algebras as data model, as described in section 3 below.

2.2. Core language

The second step provides a syntax and associated semantics for defining data type components: declaration of types[2] and operations, definition of functions, denotation of data expressions. The stress is put on minimality and orthogonality of concepts, which eases subsequent formal treatment of core specifications. Further syntactical and semantic sugar is introduced at later steps, and expressed in terms of the core language. Section 4 defines a core data language for E-LOTOS.

2.3. Extended function definitions

The core language is barely usable in itself, but provides the building bricks for defining more convenient language constructions as syntactical extensions. Section 5 extends the core language of section 4 with new constructions for defining functions, such as **if-then-else**, **case-of** and **let-in**.

2.4. Rich term syntax

Section 6 defines a richer term syntax, allowing more concise and readable notations for common values such as sequences, numbers, strings, etc. These are introduced as a purely syntactical facility, rather than in association with particular types.

2.5. Modules

Modularization is another important requirement for E-LOTOS. This involves the ability of encapsulating core elements (data types, functions, processes, …) into modules, but also of defining module interfaces as well as constructions for combining

[2]In conformance with functional language practice, we will call *type* the characterization of a data domain in the language (rather than *sort*, as in algebraic practice). Accordingly, a consistent collection of sorts and operations will be a particular case of module (rather than a *type*).

them together in various ways: union, renaming, hiding, parameterization, etc. No module formalism is defined in this proposal, see [G] and [H] for two propositions for introducing modules in LOTOS.

2.6. Built-in types

Common data types of general utility should be provided as language "built-ins" and be available within any specification (i.e. be *pervasive*). Both base types (numbers, booleans, characters,…) and compound types (records, arrays, …) are needed, together with associated functions. The latter require some parameterization, that can be provided either in the core or module part of the language. To enhance modularity of specifications, built-in types and associated operations should be encapsulated into built-in modules.

This proposal does not define any complete set of built-in types, but built-in booleans are introduced at the core language level. [G] and [J] both propose a collection of built-in data types.

The semantics of built-in types and functions must be expressed in the core model, so that specifications that use it can be given a meaning. It may be expressed either in the core language or directly as a semantic object of the data model, using any convenient mathematical device.

2.7. Type schemes

Type schemes are syntactical extensions that provide ad-hoc notations for defining instances of parameterized built-in data modules. Their primary targets are modules defining compound types like records or arrays.

The concept of type schemes is defined in [G] and underlies several older proposals for translatable extensions to LOTOS. This proposal does not define any such scheme.

3. DATA MODEL

3.1. Introduction

The core model for data definitions in E-LOTOS is based on standard universal algebras, with some refinements:

- distinction of constructor operations,
- lifted data domains for handling partial or non-terminating functions.

We first define the algebraic structures, then give them a (classical) interpretation in terms of evaluation functions. The properties imposed to the constructors provide a basis for defining pattern matching on data values.

3.2. Algebraic structures

We begin with the standard definitions for *signatures* and *algebras.*

3.2.1. Definition (signature)

A *signature* is a pair $\Sigma = (S, \Omega)$, where S is a set of *sorts*[3] and Ω is a $(S^* \times S)$-indexed family $\Omega(s_I, s')$ of *operations*. We write $op : s_I \to s'$ for $op \in \Omega(s_I, s')$.

Note that S and Ω need not be finite. For example, there may be an infinity of list sorts (for various element sorts) and an infinity of numeric constants.

3.2.2. Definition (algebra)

Given a signature $\Sigma = (S, \Omega)$, a *Σ-algebra* is a structure $A(\Sigma)$, where

- $A(s) \mid s \in S$ is a S-indexed family of *carrier sets*,
- $A(op) : A(s_I) \to A(s') \mid op : s_I \to s' \in \Omega$ is a Ω-indexed family of *functions,*

with $A(s_I) = \prod (A(s))_I$.

The first refinement to standard algebras is the lifting of carrier sets with a bottom element, for handling partial or non-terminating functions.

3.2.3. Definition (lifted algebra)

Given a signature $\Sigma = (S, \Omega)$, a *lifted Σ-algebra* is a Σ-algebra A, where

- there is a distinguished *bottom element* $\perp_s \in A\ (s)$ for all $s \in S$,
- $A(op)(v_I) = \perp_s$ if $v_i = \perp_{si}$ for any i in I, for all $op : s_I \to s' \in \Omega$.

More generally, the *lifting* of a set S is the set $S_\perp = S \cup \{\perp\}$ and the *strict extension* of a function $f : S \to S'$ is $f_\perp : S_\perp \to S'_\perp$ such that $f_\perp(x) =$ (if $x = \perp$ then $\perp$ else $f(x)$).

We define $\mathbf{dom}\, f = \{v_I \mid f(v_I) \neq \perp\}$ and $\mathbf{codom}\, f = \{f(v_I) \mid v_I \in \mathbf{dom}\, f\}$

We will drop the sort subscript of bottom elements when they are obvious from context. We will also speak of algebras for lifted algebras, carrier sets for lifted carrier sets and functions for strict functions. Bottom elements induce a partial definedness ordering on carrier sets and all derived structures in the usual sense; a least fixed point over this ordering is used to define recursive functions.

Lifted algebras are isomorphic to partial algebras as defined in [G], as suggested by the definition of **dom**. We prefer lifted algebras because they lead to a simpler handling of non-termination in the definition of recursive functions.

As a second refinement, a subset of the operations is chosen to be the constructors for the algebra. These constructors are used to build patterns that can be matched against values. The corresponding functions are used backwards, i.e. to find arguments from which a given value is built. For this to be well-defined we will have to set requirements on constructor functions. Constructors provide a basis for introducing pattern-matching in E-LOTOS.

3.2.4. Definition (constructors)

A *signature with constructors* is a triple $\Sigma = (S, \Omega, \Omega_c)$, where (S, Ω) is a signature and $\Omega_c \subseteq \Omega$ is a set of *constructors.* We write Σ_c for (S, Ω_c). When used in signature contexts, (S, Ω, Ω_c) stands for (S, Ω).

Note that algebras need not be constructor-generated, i.e. some values may not be constructible from any constructor at all. In particular, some sorts may have no

[3]Since we deal here with algebras, we step back from the previous note and speak about *sorts*. Consequently, *types* of the E-LOTOS language will become *sorts* in algebra signatures (and *carrier sets* in algebras themselves).

constructors. These can be thought of a abstract sorts since no information is given about the nature of the values of those sorts.

In practical cases, the usage of operations as either constructors or functions will be distinguished by the syntactical context in which they occur: functions in expressions and constructors in patterns.

In general, several functions may return the same value. If these functions are constructors, it means that a given value may be constructed in different ways. We now require that values be at most uniquely constructed.

3.2.5. Definition (free constructors)

Given a signature with constructors $\Sigma = (S, \Omega, \Omega_c)$, and a lifted Σ-algebra A, the constructors are *free* in A, or A is *freely constructed*, iff the following conditions hold:

a. if $A(c)(v_I) = \perp$ then $v_i = \perp$ for some i in I,

b. if $A(c)(v_I) = A(c')(v'_J) \neq \perp$ then $c = c'$ and $v_I = v'_J$,

with $c : s_I \rightarrow s''$, $c' : s'_J \rightarrow s''$ in Ω_c, v_I in $A(s_I)$, v'_J in $A(s'_J)$.

3.2.6. Conjecture (freeness)

This corresponds to the algebraic notion of free constructors.

The intuition behind free constructors is that if some value v is constructed from some constructor c applied to arguments v'_I, i.e. $v = c(v'_I)$, then no other constructor or arguments may return the same v.

We also require that constructors be well founded: no value can be infinitely decomposed from its constructors. This is a natural restriction that provides a foundation for reasoning inductively on constructors and for applying term rewriting techniques to specifications.

3.2.7. Definition (well-founded constructors)

Given a signature with constructors $\Sigma = (S, \Omega, \Omega_c)$, and a lifted Σ-algebra A, the *containment relation* $< \subseteq A \times A$ is the non-reflexive transitive closure of the relation $\leftarrow$ with $v \leftarrow v'$ iff $v' = A(c)(v''_I) \neq \perp$ and $v''_I = v$ for some $i \in I$.

Constructors are *well-founded* in A , or A is *well constructed*, iff $<$ is a well-founded partial order over A.

Finally we add equality predicates to all sorts. As a prerequisite we need booleans in algebras.

3.2.8. Definition (boolean algebra)

A signature with constructors $\Sigma = (S, \Omega, \Omega_c)$ *contains booleans* iff $\Sigma_{\mathbf{bool}} \subseteq \Sigma$, with

$S_{\mathbf{bool}} = \{\mathbf{bool}\}$,

$\Omega_{\mathbf{bool}} = \Omega_{c,\mathbf{bool}} = \{\mathbf{true} : \rightarrow \mathbf{bool}, \mathbf{false} : \rightarrow \mathbf{bool}\}$.

A lifted Σ-algebra A contains booleans iff Σ contains booleans and

$A(\mathbf{bool}) = \{A(\mathbf{true}), A(\mathbf{false})\}_\perp$.

3.2.9. Definition (algebra with equality)

Given a signature with constructors and booleans $\Sigma = (S, \Omega, \Omega_c)$, Σ *augmented with equality* is a signature with constructors $\Sigma^{\mathbf{eq}}$ with $S^{\mathbf{eq}} = S$, $\Omega_c{}^{\mathbf{eq}} = \Omega_c$ and

$$\Omega^{\mathbf{eq}} = \Omega \cup \{\mathbf{eq} : s, s \to \mathbf{bool} \mid s \in S\}.$$

Given a lifted Σ-algebra A with booleans, A *augmented with equality* is a $\Sigma^{\mathbf{eq}}$-algebra $A^{\mathbf{eq}}$ with

$$A^{\mathbf{eq}}(\mathbf{eq})(v, v') = (\text{if } v = v' \text{ then } A(\mathbf{true}) \text{ else } A(\mathbf{false}))_\perp.$$

Imposing equality *as a predicate* to be generally available and faithful to equality *in the algebraic sense* saves the specifier from having to consider two levels of logic (= and eq), which was one major source for confusion in LOTOS.

An algebra is well-formed for our purpose if it meets all the above requirements.

3.2.10. Definition (well-formed algebra)

A *well-formed signature* is a signature with constructors and booleans.

A *well-formed algebra* is a freely and well constructed lifted algebra with booleans.

Equality is not a requirement, but well-formed algebras will be augmented with equality when needed.

3.3. Data evaluation

Every signature defines a grammar of terms. Every term induces an evaluation function, mapping variable assignments to values.

3.3.1. Definition (term, term algebra)

Given a signature $\Sigma = (S, \Omega)$ and a S-indexed set of *variables* $X(s) \mid s \in S$, the set $\mathbf{T}(\Sigma, X)(s)$ of *terms* of sort s is the smallest set such that

- $X(s) \subseteq \mathbf{T}(\Sigma, X)(s)$
- if $t_I \in \mathbf{T}(\Sigma, X)(s_I)$ and $op : s_I \to s' \in \Omega$ then $t' = op(t_I) \in \mathbf{T}(\Sigma, X)(s')$

with $\mathbf{T}(\Sigma, X)(s_I) = \prod (\mathbf{T}(\Sigma, X)(s))_I$.

$\mathbf{T}(\Sigma, X)$ is extended to a *term Σ-algebra* with

$$\mathbf{T}(\Sigma, X)(op)(t_I) = op(t_I), \text{ with op} : s_I \to s' \in \Omega \text{ and } t_I \in \mathbf{T}(\Sigma, X)(s_I)$$

The set $\mathbf{T}(\Sigma)(s)$ of *ground terms* of sort s is $\mathbf{T}(\Sigma, \varnothing)(s)$.

The set $\mathbf{vars}(t)$ of *variables* of t is defined recursively as $\mathbf{vars}(x) = \{x\}$ and $\mathbf{vars}(op(t_I)) = \mathbf{vars}(t_I)$, with $\mathbf{vars}(t_I) = \cup\, (\mathbf{vars}(t))_I$.

3.3.2. Definition (term evaluation)

Given $t \in \mathbf{T}(\Sigma, X)(s)$, the *evaluation* of t in A is the function $A(t) : A^X \to \mathrm{A}(s)$, defined by:

- if $t = x \in X$ then $A(t)(\sigma) = \sigma(x)$,
- if $t = op(t'_I)$ then $A(t)(\sigma) = A(op)(A(t'_I)(\sigma))$,

with $A(t'_I)(\sigma) = (A(t')(\sigma))_I$.

A^X is the set of *assignments* of X in A. We say that t is *defined* under $\sigma \in A^X$ iff $\sigma \in \mathbf{dom}\, A(t)$.

While general terms denote *computations*, constructor terms, which we call *patterns*, denote *constructions* of values. If an algebra is freely constructed then constructors are invertible and term evaluation on patterns may be inverted as well into a term matching function, mapping values to variable assignments. That is, given a constructor pattern p and a value v, there is at most one assignment σ over variables of p such that $A(p)(\sigma) = v$. This function is easily defined constructively if restricted to linear patterns, i.e. such that no variable occurs more than once.

3.3.3. Definition (linear terms)

Linear terms $t \in \mathbf{T}(\Sigma, X)(s)$ are defined inductively as:

- if $t = x \in X(s)$ then t is linear,
- if $t = op(t'_I)$, t'_I are linear and $\cap\,(\mathbf{vars}(t'))_I = \emptyset$ then t is linear.

3.3.4. Definition (pattern matching)

Given a signature Σ with constructors and a freely constructed Σ-algebra A, the *projection function* $A^{-1}(c) : A(s) \to A(s_I)$ for constructor $c : s_I \to s' \in \Omega_c$, is defined as:

$A^{-1}(c)(v) =$ if $v \neq \bot$ and $v = A(c)(v'_I)$ then v'_I else $\bot$.

The *pattern matching function* in A for a linear pattern $p \in \mathbf{T}(\Sigma_c, X)(s)$ is $A^{-1}(p)$: $A(s) \to A^Y$, with $Y = \mathbf{vars}(p)$, defined by:

- if $p = x$ then $A^{-1}(p)(v) = \{x \to v\}$,
- if $p = c(p'_I)$ then $A^{-1}(p)(v) = \cup\,(A^{-1}(p'_I))(A^{-1}(c)(v))$,

with $A^{-1}(p_I)(v_I) = \cup\,(A^{-1}(p)(v))_I$.

The unicity of $A^{-1}(p)(v)$ derives from the constructors being injective. The union of assignments in the second case for pattern matching is well defined thanks to the linearity requirement..

3.3.5. Conjecture (correctness of pattern matching)

If $p \in \mathbf{T}(\Sigma_c, X)(s)$ with $Y = \mathbf{vars}(p)$ then $A^{-1}(p)(v) = \sigma$ iff $A(p)(\sigma) = v$ and $\sigma \in A^Y$.

The proof of this conjecture is based on induction on pattern size.

The separation of codomains for constructors also ensures determinacy between different patterns, in the following sense: if a single value matches two different patterns, then these patterns are unifiable. Conversely, if two patterns are not unifiable then any value will match at most one. Such patterns are said to be mutually exclusive; they are used in the core language to ensure deterministic function definitions.

3.3.6. Definition (substitution, term unification)

Given a term algebra $\mathbf{T}(\Sigma, X)$, a *substitution* is an assignment over terms $\sigma \in \mathbf{T}(\Sigma, X)^X$ and we write $\sigma(t)$ for $\mathbf{T}(\Sigma, X)(t)(\sigma)$. A substitution σ *unifies* two terms t, t' in $\mathbf{T}(\Sigma, X)$ iff $\sigma(t) = \sigma(t')$. t and t' are *unifiable* if such a substitution exists, otherwise they are *mutually exclusive*.

It is straightforward to find unifiers or check mutual exclusion of linear terms algorithmically, by recursion on term structure.

The following example defines a core algebra for natural numbers. The names `f0`, `f1`, … have been chosen stress the difference with the corresponding semantic values; they would be (`0`, `1`, …) in the text of concrete specifications.

3.3.7. Example (algebra for naturals)

The algebra of natural numbers is a $\langle S, \Omega\rangle$-algebra *Nat*, where

$S = \{\texttt{nat}\}$
$\Omega_c = \{\texttt{f0}, \texttt{f1}, \ldots : \rightarrow \texttt{nat}\}$
$\Omega = \Omega_c \cup \{\texttt{add}, \texttt{div} : \texttt{nat}, \texttt{nat} \rightarrow \texttt{nat}\}$

$Nat(\texttt{nat}) = \mathbf{N}_\perp$
$Nat(\texttt{f0})() = 0$, $Nat(\texttt{f1})() = 1, \ldots$
$Nat(\texttt{add})(m, n) = m + n$
$Nat(\texttt{div})(m, n) = \text{if } n \neq 0 \text{ then max } \{q \in \mathbf{N} \mid q \leq m / n\} \text{ else } \perp$

Evaluation of the expression `add(f3(),f5())` gives

$Nat(\texttt{add(f3(),f5())}) = Nat(\texttt{f3()}) + Nat(\texttt{f5()}) = 3 + 5 = 8$

Pattern matching is trivial since there are only constants.

3.4. Discussion

This provides a very general model for describing types and functions. An important difference with respect to the LOTOS approach is that no hypothesis is made on how these algebras are defined. In contrast, LOTOS considers algebras only as derived from equational specifications. Data domains need not be defined as (classes of) terms here; some values may even not correspond to any term. This allows the consideration of non-countable types like real numbers on a sound semantic basis.

Of course, the practical usability of such algebras mostly depends on how they are defined. Three ways are considered:

- via the core language,
- via other data languages,
- as built-in types.

The core language defined in section 4 has nice computational properties. For other data languages this will depend on how their semantics is defined. The same remark holds for built-ins. In particular, there are some advantages in defining them using the core language wherever possible:

- built-in types will benefit from the properties of the core language,
- the specification will provide a default implementation for tool builders that do not want to provide a specific support for those types,
- it will reduce the amount of mathematical definitions by re-using the core language.

Once modules are introduced, it is desirable that separate modules define separate algebras and that they combine in a compositional way. In contrast, LOTOS types can be considered as modules, but their combination is defined at the equation level and the resulting algebra is only computed at the top level, with sad consequences on modularity.

4. CORE DATA E-LOTOS

4.1. Introduction

ACT ONE uses equational specifications and initial algebra semantics. While this could seem very appealing for its high level of abstraction and expressiveness, it is now well known that it has nasty consequences on the practical processing of LOTOS specifications:

- Equational specifications are semi-decidable. This means that tools cannot implement a procedure that checks equality between two values in a general way. Instead, derived evaluation techniques are used (term rewriting, narrowing,…) that are correct only on a restricted class of specifications.
- There is no type protection mechanism: since the resulting algebra is computed globally for each specification, a local faulty equation may collapse one, several or all type definitions throughout the whole specification.

On the other hand, equational specifications are now supported by a strong background of theoretical results and lend themselves nicely to formal treatments such as theorem proving. This is an important feature for a large spectrum specification language such as LOTOS; for example, the latest generation of LOTOS simulators use proof-oriented techniques (narrowing) for handling data types symbolically [Wolz90].

The core E-LOTOS language gives up equational specifications for a more easily computable formalism, based on a clear distinction between constructors and functions. The latter are defined through a restricted form of oriented rewrite rules. This provides computable specifications and safe combination of definitions, while remaining within the scope of the well developed theory of term rewrite systems.

4.2. Core language syntax

The core language provides syntax and semantics only for the essential data-related parts in a specification:

- evaluation of expressions and patterns,
- definition of new data types,
- definition of data functions.

The syntax for expressions and patterns is merely a more concrete form for abstract terms as presented above. At the core level, only functional prefix notation is used; no attempt is made to provide convenient notations for particular terms. The distinction between patterns and expressions depends on the context; both share the same syntax at the core level.

Types are always denoted by type identifiers in this proposal, because all types must be named. If the type system is enriched in a later version then other kinds of type expressions could be introduced.

4.2.1. Syntax (terms)

data-term	::=	*var-id*
	|	"(" *data-term* ")"
	|	*opn-id* ["(" *data-term* ("," *data-term*)* ")"]
pattern	::=	*data-term*
formula	::=	*data-term*

expr ::= *data-term*

type ::= *type-id*

opn-desc ::= *opn-id* ["(" *type* ("," *type*)* ")"]

The core language supports only one kind of data types, which are a generalization of variant records as they appear in many conventional languages. A new type is defined as a family of named record structures. Direct and indirect recursion between data types is supported.

4.2.2. Syntax (data type definition)

con-descs ::= *opn-desc*
| *con-descs* "|" *con-descs*

type-descs ::= *type* "**is**" *con-descs*
| *type-descs* "&" *type-descs*

type-def ::= "**type**" *type-descs* "**endtype**"

4.2.3. Examples (data type definition)

```
type     msg   is request(service,args) | reply(result)
endtype

type     tree  is leaf(nat) | node(trees)
&        trees is nil | cons(tree,trees)
endtype
```

The name of each variant record is a constructor for values of that type. Using this gives cleaner semantics than using a record field as discriminant; it amounts to defining an implicit enumerated discriminant type with the variant names as values. Recursive types provide a clean abstraction for defining structures built with pointers in more conventional languages, such as lists or trees.

A function is defined as an operation signature associated to a set of evaluation rules. As for constructors, several function declarations may be grouped together to allow for mutually recursive definitions.

4.2.4. Syntax (function definition)

rules ::= *fun-id* ["(" *pattern*("," *pattern*)* ")"] ":=" *formula*
| *rules* "|" *rules*

fun-descs ::= *opn-desc* ":" *type* "**is**" *rules*
| *fun-descs* "&" *fun-descs*

fun-def ::= "**fun**" *fun-descs* "**endfun**"

The patterns are required to be left-linear and mutually exclusive. This is easy to check statically and provides very nice properties for handling function definitions symbolically. In the previous version of this proposal, rules were considered with sequential priority instead. Though the old solution gave more expressive power and often lead to more concise definitions, it has been abandoned because the properties of priority rewrite systems are much weaker and still largely unexplored [Baeten$$$]. implementors could decide to apply sequential evaluation nevertheless but should at least issue warnings when overlapping patterns are found.

4.2.5. Examples (function definition)

```
fun countleaves(tree) : nat is
          countleaves(leaf(n)) := 1
    |     countleaves(node(nil)) := 0
    |     countleaves(node(cons(t,ts))) :=
```

```
                 add(countleaves(t),countleaves(node(ts)))
endfun

fun search(nat,tree) : bool is
          search(n,leaf(m)) := eq(m,n)
    |     search(n,node(ts)) := searchall(n,ts)
&   searchall(nat,trees) : bool is
          searchall(n,nil) := false
    |     searchall(n,cons(t,ts)) :=
                 or(search(n,t),searchall(n,ts))
endfun
```

A core data specification is any sequence of such definitions. Any mechanism for parameterization, hiding or renaming concerns the module language.

4.2.6. Syntax (data specification)

data-defs	::=	*type-def*
	|	*fun-def*
	|	*data-defs data-defs*
data-spec	::=	"**data**" *data-defs* "**enddata**"

4.3. Core language semantics

We now define semantic mappings **f** [a] which, for any instance a of one of the syntactical classes above, build a corresponding semantic structure. These mappings are partial, in that **f** [a] is defined only for well-formed a. The well-formedness criteria are the *static semantics* of the core language; they are introduced with "Req:" in the sequel.

We do not consider here the resolution of name conflicts. At this level we suppose that all identifiers are different. Identifiers are used to populate the needed base semantic classes.

4.3.1. Definition (base semantic classes)

The following semantic classes are defined:

$\mathbf{S} = \{tid \mid tid$ is a *type-id*$\}$ is the class of all sorts,
$\mathbf{OP} = \{opid \mid opid$ is a *opn-id*$\}$ is the class of all operations,
$\mathbf{V} = \{vid \mid vid$ is a *var-id*$\}$ is the class of all variables,
$\mathbf{W} = \mathbf{OP} \times \mathbf{S}^* \times \mathbf{S}$ is the class of operation signatures $op : s_I \to s'$,
$\mathbf{X} = \mathbf{V} \times \mathbf{S}$ is the class of variable signatures,
$\mathbf{SIG} \subseteq \mathbf{S}^* \times \mathbf{W}^* \times \mathbf{W}^*$ is the class of all signatures with constructors,
$\mathbf{T} = \mathbf{T}((\mathbf{S}, \mathbf{W}), \mathbf{X})$ is the class of all terms,
$\mathbf{R} = \mathbf{T}^* \times \mathbf{T}$ is the class of all rules $p_I \to e$,
ALG is the **SIG**-indexed class of lifted Σ-algebras $A : \Sigma$.

The definition for **S** is sufficient in the present setting because all sorts have to be declared before their use. With a richer typing, **S** could become a structure over *type-id*.

The semantics for syntactic terms are abstract terms, as defined in 3.3.1. The latter are either evaluated as expressions or matched as patterns, according to a given algebra, depending on the context in which they occur. This is illustrated by the semantics for functions below.

4.3.2. Definition (semantics for terms)

If	t is a *data-term*
Then	**term** $[t] \in (\mathbf{SIG} \times \mathbf{S}) \to \mathbf{T}$
With	**term** $[vid]\ (\Sigma, s) = vid$
	term $[(\ t\ )]\ (\Sigma, s) =$ **term** $[t]\ (\Sigma, s)$

term [$opid$] (Σ, s) = $opid$()
Req: $opid : \rightarrow s \in \Sigma$
term [$opid(t_I)$] (Σ, s) = $opid$((**term** [t] (Σ, s'))$_I$)
Req: $opid : s'_I \rightarrow s \in \Sigma$

Data type and function definitions define algebras. The interpretation is incremental, i.e. definitions are taken sequentially and added to the initial algebra. This imposes that sorts and functions be defined before they are used.

The initial types for this incremental construction are contained in a *base algebra* $A°$ of signature $\Sigma°$. At this level $A°$ contains only booleans. Once modules and built-in types are introduced, $A°$ will become more richly populated and will depend on the imports of the defined module.

4.3.3. Definition (semantics for definitions)

If *spec* is a *data-spec*
Then **alg** [*spec*] $\in$ **ALG**
With **alg** [**data** *defs* **enddata**] = **alg** [*defs*] ($A°$: $\Sigma°$)
$\Sigma° = \Sigma_{\mathbf{bool}}$
$A° = T(\Sigma_{\mathbf{bool}})_\perp$

If *defs* is a *data-defs*
Then **alg** [*defs*] $\in$ **ALG** $\rightarrow$ **ALG**
With **alg** [*tdef*] (A : Σ) = **talg** [*tdef*] (A : Σ)
alg [*fdef*] (A : Σ) = **falg** [*fdef*] (A : Σ)
alg [*defs defs'*] (A : Σ) = **alg** [*defs'*] (**alg** [*defs*] (A : Σ))

Type definitions define new sorts, constructors and functions. The new carrier sets contain terms generated over inherited carrier sets. To define this, a term algebra **T** (Σ, X) is used, where the carrier sets of A are taken as sets of variables X.

4.3.4. Definition (semantics for types)

If *tdef* is a *type-def*
Then **talg** [*tdef*] $\in$ **ALG** $\rightarrow$ **ALG**
With **talg** [**type** *tdescs* **endtype**] = (A' : Σ')
$\Sigma' = (S', \Omega', \Omega_c') = (S \cup S'', \Omega \cup \Omega'', \Omega_c \cup \Omega'')$
S'' = **tsorts** [*tdescs*]
Ω'' = **topns** [*tdescs*] (S')
$A'(s)$ = if $s \in S$ then $A(s)$ else $(\mathbf{T}(\Sigma_g, X_g)(s))_\perp$
$A'(op)$ = if $op \in \Omega$ then $A(op)$ else $(\mathbf{T}(\Sigma_g, X_g)(op))_\perp$
$\Sigma_g = (S', \Omega'')$
$X_g(s)$ = if $s \in S$ then $A(s)$-$\{\perp\}$ else $\varnothing$

If *tdescs* is a *type-descs*
Then **tsorts** [*tdescs*] $\in$ $\mathbf{S}^*$
topns [*tdescs*] $\in$ $\mathbf{S}^* \rightarrow \mathbf{W}^*$
With **tsorts** [*tid* **is** *cdescs*] = {*tid*}
tsorts [*tdescs* & *tdescs'*] = **tsorts** [*defs*] $\cup$ **tsorts** [*tdescs'*]
topns [*tid* **is** *cdescs*] (S) = **copns** [*cdescs*] (S, *tid*)
topns [*tdescs* & *tdescs'*] (S) = **topns** [*tdescs*] (S) $\cup$ **topns** [*tdescs'*] (S)

If *cdescs* is a *con-descs*
Then **copns** [*cdescs*] $\in$ $(\mathbf{S}^* \times \mathbf{S}) \rightarrow \mathbf{W}^*$
With **copns** [*opid*] (S, s) = $\{opid : \rightarrow s\}$
copns [$opid(tid_I)$] (S, s) = $\{opid : tid_I \rightarrow s\}$

Req: $\{tid_I\} \subseteq S$
copns $[cdescs \mid cdescs']$ (S, s) =
copns $[cdescs]$ $(S, s) \cup$ **copns** $[cdescs']$ (S, s)

Function definitions define new functions over existing sorts. Intuitively, evaluation of functions may not terminate since general recursion is allowed. In algebras such cases will return $\perp$. To cover those cases we have to use a fixed point in the semantics.

4.3.5. Definition (semantics for functions)

If *fdef* is a *fun-def*
Then **falg** $[fdef] \in$ **ALG** $\rightarrow$ **ALG**
With **alg** [**fun** *fdescs* **endfun**] $(A : \Sigma) = (A' : \Sigma')$
$\Sigma' = (S', \Omega', \Omega_c') = (S, \Omega \cup \Omega'', \Omega_c)$
Ω'' = **fopns** $[fdescs]$ (S')
D = **fdefs** $[fdescs]$ (Σ')
A' = the least fixed point of the recursive definition of A''
$A''(s) = A(s)$
$A''(op)(v_I)$ = if $op \in \Omega$ then $A(op)(v_I)$ else
if $A^{-1}(p_I)(v_I) = \sigma$ for some $(p_I, e) \in D(op)$
then $A''(e)(\sigma)$
else $\perp$

If *fdescs* is a *fun-descs*
Then **fopns** $[fdescs] \in$ **S*** $\rightarrow$ **W***
With **fopns** $[opid : tid'$ **is** $rules]$ $(S) = \{opid : \rightarrow tid'\}$
Req: $tid' \in S$
fopns $[opid(tid_I) : tid'$ **is** $rules]$ $(S) = \{opid : tid_I \rightarrow tid'\}$
Req: $\{tid_I\} \subseteq S, tid' \in S$
fopns $[fdescs \,; fdescs']$ (S) =
fopns $[fdescs]$ $(S) \cup$ **fopns** $[fdescs']$ (S)

If *fdescs* is a *fun-descs*
Then **fdefs** $[fdescs] \in$ **SIG** $\rightarrow$ (**OP** $\rightarrow$ **R***)
With **fdefs** $[opid : tid'$ **is** $rules]$ $(\Sigma) = \{opid \rightarrow R\}$
R = **rules** $[rules]$ $(\Sigma, opid : \rightarrow tid')$
fdefs $[opid(tid_I) : tid'$ **is** $rules]$ $(\Sigma) = \{opid \rightarrow R\}$
R = **rules** $[rules]$ $(\Sigma, opid : tid_I \rightarrow tid')$
fdefs $[fdescs \,\&\, fdescs']$ (Σ) =
fdefs $[fdescs]$ (Σ) + **fdefs** $[fdescs']$ (Σ)

If *rules* is a *rules*
Then **rules** $[rules] \in$ (**SIG** $\times$ **W**) $\rightarrow$ **R***
With **rules** $[op := expr]$ $(\Sigma, op' : s_I \rightarrow s') = \{((), e)\}$
e = **term** $[expr]$ $(\Sigma^{\mathbf{eq}}, s')$
Req: $op = op', s_I = ()$, **vars**$(e) = \emptyset$
rules $[op(pat_I) := expr]$ $(\Sigma, op' : s_J \rightarrow s') = \{(p_I, e)\}$
p_I = (**term** $[pat]$ $(\Sigma_C, s))_I$
e = **term** $[expr]$ (Σ, s')
Req: $op = op', op(p_I)$ is linear, **vars**$(e) \subseteq$ **vars**(p_I)
rules $[rules \mid rules']$ $(\Sigma, op' : s_I \rightarrow s') = (p_I, e)_J \cup (p'_I, e')_{J'}$
$(p_I, e)_J$ = **rules** $[rules]$ $(\Sigma, op' : s_I \rightarrow s')$
$(p'_I, e')_{J'}$ = **rules** $[rules']$ $(\Sigma, op' : s_I \rightarrow s'')$
Req: $p_{I,j}$ and $p'_{I,j'}$ are mutually exclusive, for all $j \in J, j' \in J'$.

Note the use of Σ^{eq}(rather than Σ) for right hand sides of rules. Σ^{eq} should be used similarly in the semantics of behaviours, wherever terms are evaluated.

4.3.6. Conjecture (persistency of core semantics)

If $(A' : \Sigma') =$ **alg** [*defs*] $(A : \Sigma)$ then A' is a persistent extension of A.

4.4. Discussion

Though the semantics for functions is expressed directly in algebraic terms, their interpretation as as a non-conditional term rewriting system is straightforward. As a consequence of the restrictions on the nature of those rules, the resulting rewrite system is strongly confluent (and even orthogonal [Dershowitz90]), so that effective evaluation is possible. Termination is not enforced though, since this would impose a strong restriction on the expressivity of the language.

The restrictions also ensure that a function definition cannot affect previously defined types or functions and thus protection of imported types is also ensured, as suggested by the incremental nature of the semantics and conjectured in 4.3.6.

Backwards compatibility with ACT ONE is not achieved in the sense that an ACT ONE specification cannot in general be translated into core E-LOTOS. This was to be expected since the latter has computability properties that the former lacks, and this is precisely the reason for the proposed change. On the other hand, an core E-LOTOS specification is easily translatable into ACT ONE, so that the adaptation of existing tools should be relatively straightforward. Indeed, the proposed evaluation method is already used in some tools as a surrogate for ACT ONE semantics.

The type definitions offered by the core language allow the definition of any freely, finitely generated type. This is not as general as one could wish, in particular, some types cannot be expressed in core E-LOTOS with the same level of abstraction as in LOTOS. However, hiding can be used at the module level to provide satisfactory abstractions.

Consider, for example, the definition of integers based on operations 0, succ and pred. This could be defined as an E-LOTOS module as follows (the syntax for modules is inspired from [G]).

4.4.1. Example (integers in E-LOTOS)

```
descr integerdescr :=
    type int endtype
    fun 0 : int & s(int) : int & p(int) : int endfun
enddescr

mod integer :=
    export integerdescr;
    type int is 0 | cs(int) | cp(int) endtype
    fun s(int) : int is
           s(0) := 0
    |      s(cs(n)) := cs(cs(n))
    |      s(cp(n)) := n
    endfun
    fun p(int) : int is
           p(0) := 0
    |      p(cs(n)) := n
    |      p(cp(n)) := cp(cp(n))
    endfun
endmod
```

The net effect of this is that the `integer` module contains different values for all terms built from `0`, `cp` and `cs`, but since it exports functions that return well-formed terms, only those well-formed terms will be accessible from the outside. Moreover, no constructor is exported so pattern matching is not available on those values (this makes the given definition rather useless; some predicates should be defined and exported in a real case).

Yet this approach will not work in some cases because it still requires a unique construction for any given value. The canonical counter-example is the set, for which there is no general way to chose among the various permutations of its elements. The best that can be done in this case is to define set equality explicitly as a function; but the built-in `eq` will be finer than that. Another possible approach is to allow the definition of `eq` to be overridden, but this is a dangerous thing to do because the newly defined `eq` is not guaranteed to be a congruence and that could make the definition of LOTOS synchronization unsound.

Yet another approach is to define sets directly as a built-in (schema of) algebra. There are still some delicate points to cope with, however: if the properties expressed in section 3.2 are to be preserved, then functions over sets have to be deterministic w.r.t. set equality, which means that picking an element or listing the elements of a set must return equal values for equal sets.

5. SYNTAX EXTENSIONS

5.1. Extended definition syntax

The core syntax for function definitions is rather crude: every function is essentially defined as a collection of unconditional, mutually exclusive rewrite rules. This section builds more conventional and convenient constructions on top of that basis. Three new syntactic forms are provided: **case-of**, **if-then-else** and **let-in**. The two latter are translated into an instance of the former. The translation for **case-of** is more elaborate and requires the generation of auxiliary "hidden" function definitions.

The syntax class of formulas, used for the right hand sides of function definition rules, is extended with the new forms.

5.1.1. Syntax (formulas)

formula ::= *expr*
| "**case**" *expr* ":" *type* "**of**" *case-rules* "**endcase**"
| "**if**" *expr* "**then**" *formula* ["**else**" *formula*] "**endif**"
| "**let**" *pattern* ":" *type* "=" *expr*
"**in**" *formula* "**endlet**"

case-rules ::= *pattern* "=>" *formula*
| *case-rules* "|" *case-rules*

5.1.2. Example (extended formulas)

```
fun takeone(list,element) : element  is
   takeone(l,y) :=
   case takeall(l) : list of
         nil => y
     |   cons(x1,l1) => x1
   endcase
endfun

fun takeall(list) : list is
   takeall(nil) := nil
|  takeall(cons(x,l)) :=
```

```
    if take(x)
          then cons(x,takeall(l))
          else takeall(l)
    endif
endfun

fun taketwice(list,element) : list is
    taketwice(l,y) :=
    let x : element = takeone(l,y) in
          cons(x,cons(x,nil))
    endlet
endfun
```

The translation of an extended *fun-def* is defined via a function **reduce** that returns a reduced *fun-def* in the core syntax. The central part is the translation of **case-of** forms: in essence, each phrase **case** e : t **of** p_1 => f_1 | … | p_n => f_n **endcase** is replaced by an occurrence of a new hidden function $h(t)$, and $h(t)$ itself is defined as $h(p_1) := f_1$ | … | $h(p_n) := f_n$. The technical definition is a little more complex, because other variables may occur in f_i and their value must therefore be passed as parameters to h.

Translated phrases are enclosed in brackets […]. *txt* + *txt'* denotes the concatenation of phrases *txt* and *txt'*. *txt* +@+ *txt'*, where @ is replaced by any syntactical token, stands for *txt* + [@] + *txt'*, if either *txt* or *txt'* is not empty, otherwise *txt* + *txt'*. Iteration of +@+ over txt_I is denoted $\sum_{@} txt_I$.

5.1.3. Definition (translation of function definitions)

If [*fdef*] is a *fun-def*
Then **reduce** [*fdef*] is a *fun-def*
With **reduce** [**fun** *fdescs* **endfun**] = [**fun** *fdescs'* **endfun**]
  [*fdescs'*] = **reduce** [*fdescs*]

If [*fdescs*] is a *fun-descs*
Then **reduce** [*fdescs*] is a *fun-descs*
With **reduce** [*opid* : *tid'* **is** *rules*] = [*opid* : *tid'* **is** *rules'*] +&+ [*fdescs'*]
  ([*rules'*], [*fdescs'*]) = **reduce** [*rules*] ([], [*tid'*])
 reduce [*opid*(*tids*) : *tid'* **is** *rules*] =
  [*opid*(*tids*) : *tid'* **is** *rules'*] +&+ [*fdescs'*]
  ([*rules'*], [*fdescs'*]) = **reduce** [*rules*] ([*tids*], [*tid'*])
 reduce [*fdescs* & *fdescs'*] = [*fdescs''*] +&+ [*fdescs'''*]
  [*fdescs''*] = **reduce** [*fdescs*]
  [*fdescs'''*] = **reduce** [*fdescs'*]

If [*rules*] is a *rules*
Then **reduce** [*rules*] is a (*type**, *type*) → (*rules*, *fun-descs*)
With **reduce** [*op* := *f*] ([*tids*], [*tid'*]) = ([*op* := *e*], [*fdescs*])
  ([*e*], [*fdescs*]) = **reduce** [*f*] ([], [*tids*], [*tid'*])
 reduce [*op*(*pats*) := *f*] ([*tids*], [*tid'*]) = ([*op*(*pats*) := *e*], [*fdescs*])
  ([*e*], [*fdescs*]) = **reduce** [*f*] ([*pats*], [*tids*], [*tid'*])
 reduce [*rules* | *rules'*] ([*tids*], [*tid'*)] =
  ([*rules''*] +|+ [*rules'''*], [*fdescs*] +&+ [*fdsecs'*])
  ([*rules''*], [*fdescs*]) = **reduce** [*rules*] ([*tids*], [*tid'*])
  ([*rules'''*], [*fdescs'*]) = **reduce** [*rules'*] ([*tids*], [*tid'*])

If [*f*] is a *formula*
Then **reduce** [*f*] is a (*pattern**, *type**, *type*) → (*formula*, *fun-descs*)
With **reduce** [*t*] ([*pats*], [*tids*], [*tid'*]) = ([*t*], [])
 reduce [**case** *t* : *tid''* **of** *cases* **endcase**] ([*pats*], [*tids*], [*tid'*]) =
  ([*h*(*hargs*)], [*fdescs*] +&+ [*hdesc*])

[*hdesc*] = [*h*(*htids*) : *tid'* **is** *hrules*]
([*hrules*], [*fdescs*]) =
reduce [*cases*] ([*h*], [*tid''*], [*pats*], [*tids*], [*tid'*])
[*h*] is a fresh *opn-id*
[*hargs*] = [*t*] +,+ [*pats*]
[*htids*] = [*tid''*] +,+ [*tids*]
reduce [**if** *t* **then** *f* **else** *f'* **endif**] ([*pats*], [*tids*], [*tid'*]) =
reduce [**case** *t* : **bool of true** => *f* | **false** => *f'* **endcase**]
([*pats*], [*tids*], [*tid'*])
reduce [**if** *t* **then** *f* **endif**] ([*pats*], [*tids*], [*tid'*]) =
reduce [**case** *t* : **bool of true** => *f* **endcase**]
([*pats*], [*tids*], [*tid'*])
reduce [**let** *p* : *tid* = *e* **in** *f* **endlet**] ([*pats*], [*tids*], [*tid'*]) =
reduce [**case** *e* : *tid* **of** *p* => *f* **endcase**]
([*pats*], [*tids*], [*tid'*])

If [*cases*] is a *case-rules*
Then **reduce** [*cases*] is a (*opn-id*, *type*, *pattern**, *type**, *type*)
→ (*rules*, *fun-descs*)
With **reduce** [*pat'* => *f*] ([*h*], [*tid''*], [*pats*], [*tids*], [*tid'*]) = ([*rule*], [*fdescs*])
[*rule*] = [*h*(*hpats*) := *e*]
([*e*], [*fdescs*]) = **reduce** [*f*] ([*hpats*], [*htids*], [*tid'*])
[*hpats*] = [*pat'*] +,+ [*pats*]
[*htids*] = [*tid''*] +,+ [*tids*]
reduce [*cases* | *cases'*] ([*h*], [*tid''*], [*pats*], [*tids*], [*tid'*]) =
([*rules*] +|+ [*rules'*], [*fdescs*] +&+ [*fdsecs'*])
([*rules*], [*fdescs*]) =
reduce [*cases*] ([*h*], [*tid''*], [*pats*], [*tids*], [*tid'*])
([*rules'*], [*fdescs'*]) =
reduce [*cases'*] ([*h*], [*tid''*], [*pats*], [*tids*], [*tid'*])

5.1.4. Example (translation of function definitions)

The function `takeone` from example 5.1.2 is translated into:

```
fun takeone(list,element) : element  is
    takeone(l,y) := h(takeall(l),l,y)
&   h(list,list,element) : list is
           h(nil,l,y) := y
      |    h(cons(x1,l1),l,y) := x1
endfun
```

Note that the **if-then-else** construct defined this way has a different and more natural behaviour than an if-then-else function: `if true then a else b endif` evaluates to `a` even if `b` is undefined, whereas the strictness requirement on functions would impose `if(p,a,b)` = ⊥ in that case.

6. RICH TERM SYNTAX

6.1. Introduction

ACT ONE sees every operation in a uniform algebraic framework. Accordingly, the concrete syntax for operations, and consequently for terms built upon them, admits no exception to the uniform, functional or infix notation. In contrast, most conventional languages provide ad-hoc syntax for common values and functions such as sequences, numbers, array indexing and so on.

Introduction of such facilities is a largely awaited enhancement to LOTOS. The usual proposed solution is to associate those notations with specific types or type families.

Our proposal is to treat such notations in a more generic way, similarly to the infix notation option already available in ACT ONE. This means that those facilities can be declared and subsequently used for any operation in any type, provided that such operations have a compatible signature (e.g. infix notation is usable only for binary operations). In particular, they will be available for built-in types that will be defined in a later stage.

While making such syntax enrichments, it is important to pay attention to the growing grammar: ambiguous notations must be avoided and the resulting language must remain manageable using conventional parsing techniques. This has been taken into account as much as possible here, though no formal analysis or prototyping has been made.

Two notations are introduced for every new operation syntax:

- the term syntax for the operation itself, as used for building terms,
- a declaration syntax for defining constructors or functions that use that syntax.

They extend the syntax of the non-terminals *data-term* and *opn-desc* defined in 4.2.1, respectively. In most cases the latter is the former where terms are replaced by sort identifiers. A complementary case, which is not handled here, is the syntax used in renaming and actualization mappings.

Since some of these notations have no variable part in their syntax, they depend heavily upon operation overloading and a corresponding resolution procedure, as in ACT ONE. Optional explicit sort indication would also be welcome, though it is not explicitly redefined here.

6.2. New syntax forms

The two existing notations in ACT ONE are the usual functional prefix notation and the infix notation, mentioned here for the sake of completeness.

6.2.1. Syntax (functional, infix)

opn-desc	::=	*opn-id* ["(" *type* ("," *type*)* ")"]
	|	*type opn-id type*
data-term	::=	*var-id*
	|	"(" *data-term* ")"
	|	*opn-id* ["(" *data-term* ("," *data-term*)* ")"]
	|	*data-term opn-id data-term*

Selectors are intended as operations that extract a sub-component of some composite value. Two notations are provided, for selecting a fixed or indexed component, respectively. This covers record field selectors, array indexing, extraction of sub-lists or sub-strings, etc.

6.2.2. Syntax (selectors)

opn-desc	::=	…
	|	*type* "." *opn-id*
	|	*type* "." "(" *type* ("," *type*)* ")"

data-term ::= …
| *data-term*"." *opn-id*
| *data-term*"." "(" *data-term*("," *data-term*)* ")"

Their usage is illustrated in the following example. Note how the same kind of notation is used to select elements in pairs (complex numbers), lists and lists of lists. Let us insist on the fact that the provided clauses are part of the example of *use* of the notations, their existence is by no means related to the notations themselves. They must be provided separately, either by the specifier or from some built-in type.

6.2.3. Example (selectors)

```
type      complex is comp(real,real) endtype

fun       complex.re : real is comp(x,y).re := x endfun
fun       complex.im : real is comp(x,y).im := y endfun

type      nats is nil | cons(nat,nats) endtype

fun       nats.hd : nat is cons(n,ns).hd := n endfun
fun       nats.tl : nats is cons(n,ns).tl := ns endfun
fun       nats.(nat) : nat is
          cons(n,ns).(0) := n
    |     cons(n,ns).(succ(k)) := ns.(k)
endfun

type      natm is nil | cons(nats,natm) endtype

fun       natm.(nat,nat) : nat is
          cons(ns,nm).(0,j) := ns.(j)
    |     cons(ns,nm).(succ(i),j) := nm.(i,j)
endfun
```

Corresponding modifiers are also offered, that is, notations intended for changing the value of some component of a composite value.

6.2.4. Syntax (modifiers)

opn-desc ::= …
| *type* "<-" *opn-id* "-" *type*
| *type* "<-" "(" *type* ("," *type*)* ")" "-" *type*

data-term ::= …
| *data-term*"<-" *opn-id* "-" *data-term*
| *data-term*"<-" "(" *data-term*("," *data-term*)* ")" "-" *data-term*

6.2.5. Example (modifiers)

```
fun       complex <-re- real : complex is
          comp(x,y) <-re- x1 := comp(x1,y)
endfun

fun       nats <-(nat)- nat : nats is
          cons(n,ns) <-(0)- n1 := cons(n1,ns)
    |     cons(n,ns) <-(succ(k))- n1 := cons(n,ns <-(k)- n1)
endfun

fun       natm <-(nat,nat)- nat : natm is
          cons(ns,nm) <-(0,j)- n1 := cons(ns <-(j)- n1, nm)
    |     cons(ns,nm) <-(succ(i),j)- n1 := cons(ns,nm <-(i,j)- n1)
endfun
```

Tuples and record notations are intended as constructors for composed values with heterogeneous components, according to a fixed signature. Tuples keep the usual positional designation whereas records use labels for associative designation.

6.2.6. Syntax (tuples, records)

opn-desc ::= ...
| "(" *type* ("," *type*)+ ")"
| "{" *opn-id* ":=" *type* ("," *opn-id* ":=" *type*)* "}"

data-term ::= ...
| "(" *data-term* ("," *data-term*)+ ")"
| "{" *opn-id* ":=" *data-term* ("," *opn-id* ":=" *data-term*)* "}"

The tuple provides little difficulty, it is basically an atypical case of functional notation, where the operation name is empty. Tuple notations must have at least two arguments, otherwise an ambiguity with bracketed expressions would result.

The record notation is more delicate, and must be considered with the following rule:

> Two record denotations { l_1:=v_1, ..., l_n:=v_n } and { l'_1:=v'_1, ..., l'_n:=v'_n }, where l'_1, ..., l'_n is a permutation of l_1, ..., l_n so that the correspondence of value sorts to labels is preserved, are instances of the same operation.

Intuitively, one can think of record denotations as being automatically permuted so that labels (and associated values) occur in alphabetical order.

6.2.7. Examples (tuples, records)

```
type      real2 is (real,real) endtype

type      ENheader is
   {      source := ENaddress,
          dest := ENaddress,
          type := ENtype }
endtype

fun       minmax(real2) : real2 is
          minmax((x,y)) := (min(x,y),max(x,y))
endfun

fun       ENheader.source : ENaddress is
          {source:=s, dest:=d, type:=t} .source := s
endfun

fun       myENhdr : ENheader is
          myENhdr := {type:=mytype, source:=me, dest:=ENbroadcast}
endfun
```

The sequence notation allows for terms built as accumulation of values of the same sort. It is intended to be used mainly as a constructor for lists, sets, etc.

6.2.8. Syntax (sequences)

opn-desc ::= ...
| "[" *type*"," "..." "]"

data-term ::= ...
| "[" *sequence*"]"

sequence ::= [(*data-term*",")* (*data-term* "..." | *data-term*)]

The sequence notation is in fact one versatile syntax covering two operations. The notation can express every possible combination of those two operations; they are not and need not be made accessible to the specifier. These operations are classical list builders `nil` and `cons`, like the ones used in previous examples.

`[a,b,c]` is the external denotation of a term whose structure is `cons(a,cons(b,cons(c,nil)))`. A whole suffix of the sequence can also be denoted using the "..." (ellipsis) convention, as in `[a,b,s...]` which denotes `cons(a,cons(b,s))`. The following examples of correspondence between denotation and internal structure should ease the understanding. There is some redundancy, in that the same term may have several equivalent denotations. Most of those variants are unclear, if not counter-intuitive; their used should be considered as bad practice and discouraged, as in the four last examples below. As a guideline, nested brackets should be avoided, unless they really denote nested sequences.

Denotation	*Structure*
`[]`	`nil`
`[a]`	`cons(a,nil)`
`[a,b]`	`cons(a,cons(b,nil))`
`[a,s...]`	`cons(a,s)`
`[s...]`	`s`
`[[s...]...]`	`s`
`[[]...]`	`nil`
`[a,[b]...]`	`cons(a,cons(b,nil))`
`[a,[b,[]...]...]`	`cons(a,cons(b,nil))`

Conversely, there will be several solutions for generating a concrete denotation for a term containing sequential operations. Any choice is valid; the following algorithm is suggested (where "^" is string concatenation):

print (*t*) := **case** *t* **of**
… (other forms)
| `nil` => "`[]`"
| `cons` (*t*,*ts*) => "`[`" ^ *print* (*t*) ^ *print-tail* (*ts*) ^ "`]`"

print-tail (t) := **case** t **of**
`nil` => ""
| `cons` (*t*,*ts*) => "`,`" ^ *print* (*t*) ^ *print-tail* (*ts*)
| *t* => "`,`" ^ *print* (*t*) ^ "`...`"

As an example, this is a definition of sets using infix and sequential notation. In various places, `[s...]` is used rather than simply `s`, to give a more uniform notation.

6.2.9. Example (sequences)

```
type     set is [element,...] endtype

fun      element isin set : bool is
         x isin [] := false
    |    x isin [y,s...] := (x eq y) or (x isin [s...])
endfun

fun      set incl set : bool is
         [] incl [s2...] := true
    |    [x,s1...] incl [s2...] :=
               (x isin [s2...]) and ([s1...] incl [s2...])
endfun

fun set seteq set : bool is
         s1 seteq s2 := (s1 incl s2) and (s2 incl s1)
endfun
```

Characters are considered here in an abstract sense, i.e. as atomic graphical units that are chained for composing text. Conventional programming languages usually provide strongly implementation-oriented definitions of characters, with explicit reference to an underlying encoding (ASCII, typically). While this surely makes sense in a *programming* language, a *specification* language like LOTOS should keep a much more abstract view.

The simplest solution would be to accept any conceivable character, suitably quoted, as a valid term. This is not acceptable however, because it would mean that the alphabet of meaningful characters acceptable in LOTOS specifications would be left open; in particular, their practical encoding for computer processing would become problematic. Instead of this, we limit the directly quotable characters to a universally available subset, and we allow for so-called escape sequences for denoting any other character. We thereby introduce a generic syntax for constants (i.e. nullary operations) that are intended to be used as constructors for defining character sorts.

6.2.10. Syntax (characters)

opn-desc ::= …
| (') *char-desc* (')

data-term ::= …
| (') *char-desc* (')

char-desc ::= *normal-char* | "{" *normal-char** "}"

normal-char ::= " " | "!" | … | "0" | "1" | … | "A" | …
(all printable ASCII characters except ({), (}) and ("))

(Remark 1: this must be parsed in a special mode, where white space is significant and upper and lower case are distinguished.)

(Remark 2: as a derogation to the general BNF convention in the rest of this document, parenthesis are used to quote the characters "single quote" and "double quote" (ASCII 23 and 18).)

(Remark 3: conventionally some escape sequences should be reserved for the three characters excluded from *normal-char*, e.g. {"} for ("), {(} for ({) and {)} for (})).

Using different delimiters ({) and (}) to indicate escape sequences makes such sequences easily processable via conventional regular expression techniques.

Characters and strings may in fact serve two purposes:

- They provide a direct denotation for specifications that deal with readable text. This is likely to occur in application-level communication protocols, where human-oriented data come into play. Note that character denotations are only useful if the behaviour of the specified system depends on specific text values (commands, keywords, etc.); otherwise a simple opaque type specification will do.
- They are a handy quick prototyping tool because they allow versatile human-meaningful values to be entered and handled. For example, when prototyping some protocol specification, one would replace abstract user data types by strings.

This proposal is clearly targeted towards the former use but supports both. The latter should be provided as a built-in data type, but using the general denotation. Accordingly, the general notation requires a more costly implementation, since abstract character terms cannot in general be stored in fixed size cells, but a specific built-in type may use an optimized encoding.

Some examples are given below, that cover digit sets, non-printable characters and accented letters. They illustrate how this versatile *syntax* mechanism may support various *semantic* kinds of definitions for characters. For conciseness, only a small sample of the real sets of characters is given.

6.2.11. Example (characters)

```
type      hexdigit is
          '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
   |      'a'|'b'|'c'|'d'|'e'|'f'
endtype

fun       'A' : hexdigit is 'A' := 'a'
&         'B' : hexdigit is 'B' := 'b'
(* &      ... *)
endfun

type      asciichar is
          ' '|'!'|'{"}'|'''|'{(}'|'{)}'  (* |... *)
   |      '{NUL}'|'{CR}'|'{LF}'|'{ESC}' (* |... *)
endtype

fun       '{000}' : asciichar is '{000}' := '{NUL}'
(* &      ... *)
&         '{065}' : asciichar is '{065}' := 'A'
(* &      ... *)
endfun

type      latin1char is
          ' '|'!'  (* ... same as ASCII *)
   |      '{'a}'|'{`a}'|'{^a}'|'{~a}'  (* ... | *)
```

If a particular specification makes heavy use of some particular exotic characters (for example of accented letters), then one may adopt and document the convention of using those characters directly in the specification, as convenient syntactic abbreviations of corresponding escape sequences (e.g. `'à'` stands for `'{'a}'`, `'©'` stands for `'{Oc}'`, etc.).

Strings are intended as denotations for sequences of characters, and are associated with the sequence notation described in the preceding sub-section, that is, a string notation is syntactically equivalent to a sequence notation containing the characters that make up the string. For example, `"abc{CR}"` is equivalent to `['a','b','c','{CR}']`. Therefore there is no particular operation declaration syntax for strings; string term syntax may be used for any sort that has declared a sequence operation over a character sort. This also means that (non-constant) string *patterns* can only be described in sequence notation, since there is no way of putting variables in a string notation.

6.2.12. Syntax (strings)

data-term ::= …
| (") *char-desc** (")

(Remarks 1 and 2 above still hold)

6.2.13. Example (strings)

```
type      asciistring is [asciichar,...] endtype

fun       myName : asciistring is
          myName := "Charles"
endfun

fun       nospace(asciistring) : asciistring is
          nospace([]) := []
   |      nospace([' ',chs...]) := [nospace(chs)...]
   |      nospace([ch,chs...]) := [ch,nospace(chs)...]
endfun
```

A generic notation for numeric values can be defined in the same spirit as strings, but this implies a quite complex mapping, in order to map the various conventional formats (123, -26, 12.1, 6.02E+23) in a useful manner into the general term framework. The usefulness of this effort is questionable, since such notations make little sense apart from a few built-in data types.

For this reason a more specific approach is taken: numeric notations are seen as distinguished terms whose internal structure is meaningful only to privileged built-in data types.

Only decimal notation is supported. Binary, octal or hexadecimal notations are used mostly for denoting computer data rather than truly numeric values; as such they are not considered to be required as a number notation facility. Instead, they should be defined as functions over corresponding string denotations.

A tilde "~" is used as negative sign, instead of the more common "-", to facilitate lexical analysis: no conflict arise between number signs and "-" as an operation identifier, and expressions such as "x-2" are scanned correctly without requiring a space between "-" and "2".

In complement to the general exponent notation, one-letter suffixes are proposed for the standard powers of 1000 (pico=10^{-12}, nano=10^{-9}, micro=10^{-6}, milli=10^{-3}, kilo=10^{3}, mega=10^{6}, giga=10^{9} and tera=10^{12}). This may seem of little use but is also of very little cost anyway.

6.2.14. Syntax (numbers)

data-term	::=	…
	|	*mantissa* [*exponent*]
mantissa	::=	["~"] *digit*+ ["." *digit*+]
exponent	::=	("e" | "E") ["~"] *digit*+
	|	"p" | "n" | "u" | "m" | "k" | "M" | "G" | "T"
digit	::=	"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

(Remark: this is a lexical syntax, i.e. white space is significant)

The structure associated with such a denotation can be defined as follows. Remember that this structure belongs to the meta-language; the general syntax denotations are used for convenience only. Note that the whole structure of the number denotation is kept instead of directly computing a corresponding fractional number. This allows for expressing syntactic requirements on the denotation at the static semantics level, when a built-in numerical sort is associated with it. For example, integer numbers should have an empty fractional part (i.e. 1.234e+3 is not accepted, though it has in fact a non-fractional value).

```
type     number is (sign,digits,digits,sign,digits)
&        sign := + | -
&        digits := [digit,...]
&        digit := '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
endtype
```

The mapping of denotations to the corresponding structures is straightforward. Signs are positive if omitted and fractional and exponent parts correspond to empty sequences if omitted. The following examples should make this clear.

denotation	structure
`421`	`(+,"421","",+,"")`
`~12`	`(-,"12","",+,"")`
`3.1416`	`(+,"3","1416",+,"")`
`1e6`	`(+,"1","",+,"6")`
`6.02e23`	`(+,"6","02",+,"23")`
`4.7e~9`	`(+,"4","7",-,"9")`

```
14.4k                    (+,"14","4",+,"3")
12.5u                    (+,"12","5",-,"6")
```

6.3. Discussion

We know of no other language that provides such a level of independence between operation syntax and types. The data typing language of SDL [CCITT89], which is another dialect of ACT ONE, offers means to define so-called name classes, which are families of constants defined by a regular expression.

Many characters have been assigned a special meaning. The syntactic classes for operation identifiers should be restricted accordingly:

Syntax

opn-id ::= *letter* [(*letter* | *digit* | "_")* (*letter* | *digit*)]
| *special*+

letter ::= "a" | "b" | … (all letters)

digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

special ::= "#" | "%" | "&" | "+" | "-" | "*" | "/" |" <" | "=" | ">"
| "@" | "\" | "^"

(Remark: this is a lexical syntax, i.e. white space is significant)

Moreover, "<-" is considered as a reserved keyword and cannot be used as an operation identifier. "-" can be used normally, however; there is sufficient context information in the modifier notation to avoid any ambiguity.

7. REFERENCES

[Baeten87] Baeten J.C.M., Bergstra J.A., Klop J.W.
"Term Rewriting Systems with Priorities"
in: Lescanne P., ed., Rewriting Techniques and Applications, Lecture Notes in Computer Science 256, Springer Verlag, 1987, pp. 83-94.

[CCITT89] CCITT
"Functional Specification and Description Language (SDL)"
CCITT Blue Book vol. X, fasc. X.1, Geneva, 1989.

[Dershowitz90] DershowitzN., Jouannaud J.P.
"Rewrite Systems"
in: Van Leeuwen J., ed., Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, Elsevier, 1990, pp. 243-320.

[G] "Data Types and Modules"
Annex G of [RD94b].

[H] Mañas J.A.
"Modular LOTOS"
Annex H of [RD94b].

[IS89] ISO/IEC JTC 1/SC 21/WG 1/FDT/C
"Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour"
ISO IS 8807, 1989.

[J] Mañas J.A.
"Predefined and External Data Types for LOTOS"
Annex J of [RD94b].

[RD94] "Revised Draft on Enhancements to LOTOS"
ISO JTC1/SC1/WG1 N1314, February 1994.

[RD94b] "Revised Draft on Enhancements to LOTOS"
New version of [RD94], future ISO JTC1/SC1/WG1 Nxxxx, after October 1994.

[Wolz90] Wolz D.
"Design of a Compiler for Lazy Pattern Driven Narrowing"
in: Ehrig H., Jantke K.P., Orejas P., Reichel H., eds., Recent Trends in Data Type Specification, Lecture Notes in Computer Science 534, Springer Verlag, pp.362-379.