Contribution to the Design of Data Types in E-LOTOS

Version 1.0

Hubert GARAVEL*
INRIA Rhône-Alpes
VERIMAG — Miniparc-ZIRST
rue Lavoisier
38330 MONTBONNOT ST MARTIN
FRANCE
Tel : +(33) 76 90 96 34
Fax : +(33) 76 41 36 20
E-mail : hubert.garavel@imag.fr

July, 17, 1995

Abstract

In the framework of the revision of the LOTOS standard undertaken within ISO, we present some considerations for improving the data type part of LOTOS.

1 Introduction

E-LOTOS users, as in most computer languages, one will 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.

Although some languages, like ASN.1, do not allow algorithms to be expressed, this is clearly not acceptable for a formal description technique like E-LOTOS.

In standard LOTOS, due to the adoption of ACTONE, algorithms and data structures are not clearly separated, which proves to confuse most users of the language. E-LOTOS should adopt a more pragmatic style, closer to what exists in most computer languages. It is worth noticing that almost all algebraic languages posterior to ACTONE (such as PLUSS, LPG, etc.) carefully distinguish *constructors* and *defined functions*.

2 Specification of data structures

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

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

Clearly, PASCAL-like types (as well as Ada-like or C-like types) are not appropriate for E-LOTOS, at least for two very strong reasons:

- As noticed during the COST 247 meeting in Warsaw (June 1995), nobody supports the introduction of pointer types in E-LOTOS, due to semantical issues such as: references to dead objects, concurrent accesses to the same object by several parallel processes...

  On the other hand, if pointers are forbidden, PASCAL-like without pointers are not sufficient to express dynamic data structures, such as lists.

- Discriminated unions in PASCAL raise another painful 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.

  In such case, the semantics of the field access is totally undefined. 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.

  Ada improves on PASCAL by detecting these situations at the expense of run-time checking. Unfortunately, this approach slows down the generated code.

The only solution for avoiding these semantic issues and preserving run-time efficiency is to use ML-like *recursive types* or — which is equivalent — to use *sorts generated by free constructors* as in most algebraic specification languages.

It is well-known that ML-like types with constructors are far superior to PASCAL-like records (see for instance [Wat90, pp. 15 and 277]). Moreover, using such types seems to be the only way to achieve a great degree of compatibility with existing ACTONE. This approach has already been promoted for E-LOTOS, for instance in [Pec94].

To express such sorts, several syntaxes (borrowed either from functional languages like ML or from algebraic languages) are plausible. However, several common-sense requirements should be considered:

1. The constructors of a given sort should be declared together with this sort, in a close syntactic proximity, instead of being disseminated in the whole LOTOS description, as it is currently permitted in LOTOS.

   This should improve the readability of E-LOTOS descriptions. It would also forbid to modify existing (or predefined) type libraries by adding new constructors to sorts that are already defined.

2. It should be possible to name the arguments (i.e., formal parameters) of constructors, which simply means that it should be possible to name the fields of records and discriminated unions.

   This is not the case in standard LOTOS, where the arguments of an operation are only defined by their sorts. As LOTOS is supposed to be a specification language, this is an clear drawback. For instance, in the following example taken from the formal description in LOTOS of the OSI95 enhanced transport service developed at the University of Liège, we see:

```
(*
 The failure probability parameters have the following 4-tuple
 structure: Failures = Prob x Prob x Prob x 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 been explained but with a comment. The following declaration would be certainly preferable:

```
type TCFailureProbabilities is Probability
   sorts
      Failures
   opns
      Failures (TCEstablishment : Prob,
                Transfer : Prob,
                TCResilience : Prob,
                TCRelease : Prob) -> Failures
   ...
 endtype
```

Giving names to the formal parameters may allow to define implicitly projection functions, which allow to consult or modify the fields of the record, as well as short-hand notations to handle values of record types.

Moreover, this would also be closer to function declarations in most languages (either algorithmic or functional) and would also be aligned with process declarations in the behaviour part of LOTOS itself.

Keeping in mind these requirements, we have basically two possible syntaxes, as shown below in the case of a boolean list. First, we may choose a ML-like syntax, similar to the one proposed in [Pec94] but modified so as to name the formal parameters[1]:

```
sort LIST is
   NIL |
   CONS (ITEM:BOOL, NEXT:LIST)
```

Note: in this case, the equivalent ML syntax would be:

```
datatype LIST =
    NIL of UNIT |
    CONS of INT * LIST
```

Another possible syntax could be the following one, which is compatible with the syntax of process definitions:

```
sort LIST is
   constructor NIL : LIST
   constructor CONS (ITEM:BOOL, NEXT:LIST) : LIST
endsort
```

If necessary for backward compatibility reasons, the old ACTONE syntax (with unnamed parameters) could be also admitted in E-LOTOS:

[1] The "|" symbol could be replaced with a "[]" symbol, in order to express a choice between several possibilities and in order to avoid the introduction of a new keyword

```
sort LIST is
   constructor NIL : -> LIST
   constructor CONS : BOOL, LIST -> LIST
endsort
```

3 Specification of computations

During the E-LOTOS interim meeting in Paris, it was decided to use a functional style for the data part of E-LOTOS.

3.1 Definition of functions

As a first consequence of this design choice, the definition of an E-LOTOS function (i.e., non-constructor) should be concentrated in a single place, instead of being defined in several equations disseminated in the whole description, as it can happen with LOTOS. Therefore, a plausible syntax (aligned with the one of process definitions) could be:

```
function F (X:BOOL, Y, Z:INT) : INT is
   (* body of function F *)
endfunc
```

The definition of `F` may take several forms:

- It may be "external", meaning that an implementation of `F` must be available in some description or programming language.
- It can be defined "functionally", as a value expression:

  ```
  function F (X:BOOL, Y, Z:INT) : INT is
     if X = false then Y
     elsif Y = Z  then Y
     else 0
     endif
  endfunc
  ```

- It may be defined as a set of algebraic equations, in the style of ACTONE if one wants to retain a great level of compatibility with standard LOTOS:

  ```
  function F (X:BOOL, Y, Z:INT) : INT is
     F (false, Y, Z) = Y;
     Y = Z  => F (true, Y, Z) = Y;
     Y <> Z => F (true, Y, Z) = 0;
  endfunc
  ```

  In such case, whether the semantics should be the initial algebra semantics or a rewrite rule semantics is an open debate. The initial algebra semantics might have problems to coexist with functional definitions.

In the sequel, we concentrate on value expressions, which are likely to be the main way to define functions.

3.2 Syntax of expressions

The following terminal (i.e., lexical) symbols will be used:

symbol	meaning
$F, F_1, F_2, ...$	function identifier
$C, C_1, C_2, ...$	constructor identifier
$S, S_1, S_2, ...$	sort identifiers
$X, X_1, X_2, ...$	variable identifier

The following non-terminal (i.e., syntactic) symbols will be defined:

symbol	meaning
$V, V_1, V_2, ...$	value expression
$P, P_1, P_2, ...$	pattern expression

In standard LOTOS, the syntactic definition of value expression is rather limited:

$$\begin{array}{lcl} V & ::= & X \\ & | & F(V_1, ..., V_n) \\ & | & V_1 = V_2 \\ & | & V_0 \ \textbf{of}\ S \\ & | & (V_0) \end{array}$$

We propose to generalize the notion of value expression, by introducing new constructs ("**let**", "**if**", "**case**", ...) that will increase the expressiveness and will also create a symmetry between value expressions and behaviour expressions.

Pattern expressions are a limited form of value expressions, which are used in pattern-matching context. A pattern expression has the following syntax:

$$\begin{array}{lcl} P & ::= & X \\ & | & X : S \\ & | & \textbf{any}\ S \\ & | & C(P_1, ..., P_n) \\ & | & P_0 \ \textbf{of}\ S \\ & | & (P_0) \end{array}$$

Value expressions generalize the value expressions that exist in LOTOS. They have the following syntax:

$$\begin{array}{lcl} V & ::= & X \\ & | & C(V_1 ... V_n) \\ & | & F(V_1 ... V_n) \end{array}$$

| $V_1 = V_2$
| $V_1 <> V_2$
| V_0 **of** S
| V_1 **andthen** V_2
| V_1 **orelse** V_2
| **if** V_0 **then** V_0'
elsif V_1 **then** V_1'
...
elsif V_n **then** V_n'
else V_{n+1}'
endif
| **case** V_0 **in**
$P_1[V_1] -> V_1'$
...
$P_n[V_n] -> V_n'$
endcase
| **assert** $V_1, ..., V_n$ **in** V_0
| **let** $P_1 = V_1, ..., P_n = V_n$ **in** V_0
| (V_0)

3.3 Static semantics of expressions (1): binding

- All occurrences of sort identifiers in pattern or value expressions are *use*-occurrences[2] that shall be bound to the definition of the corresponding sort.
- All occurrences of constructor identifiers in pattern or value expressions are *use*-occurrences that shall be bound to the definition of the corresponding constructor (possibly with some overloading resolution).
- All occurrences of function identifiers in pattern or value expressions are *use*-occurrences that shall be bound to the definition of the corresponding function (possibly with some overloading resolution).
- All occurrences of variable identifiers in value expressions are *use*-occurrences that shall be bound to the definition of the corresponding variable.
- Occurrences of variable identifies in pattern expressions can be either *use*-occurrences or *def*-occurrences[3], depending on the context.

More precisely, the set of variables declared in a pattern expression P, noted $vars\{P\}$, is defined as follows:

[2] also called *place-marking occurrences* in [ISO88]
[3] also called *binding occurrences* in [ISO88]

$$
\begin{array}{rcl}
vars\{X\} & = & \emptyset \\
vars\{X : S\} & = & \{X\} \\
vars\{\textbf{any } S\} & = & \emptyset \\
vars\{C(P_1, ..., P_n)\} & = & vars\{P_1\} \uplus ... \uplus vars\{P_n\} \\
vars\{P_0 \textbf{ of } S\} & = & vars\{P_0\} \\
vars\{(P_0)\} & = & vars\{P_0\}
\end{array}
$$

where the operator "$\uplus$" denotes the union of sets the intersection of which is empty. If the intersection is not empty (which could occur in a pattern expressions such as "$C(X : S, X : S)$" for instance), this is a static semantics error.

For a given pattern expression P_i, the variables of $vars\{P_i\}$ have the following scope:

- These variables are not visible in P_i. For instance, in the following pattern expression "$C(X : S, Y : S, X+Y)$", variables X and Y occurring in "$X+Y$" shall not be bound to the definitions "$X : S$" and "$Y : S$" contained in P_i.
- If P_i occurs in some clause "$P_i[V_i] -> V_i'$" contained in "`case`" expression, these variables are only visible in sub-expressions V_i and V_i'. They mask all variables with the same names, possibly defined in enclosing scopes.
- If P_i occurs in a "**let**" expression of the form "**let** $P_1 = V_1, ..., P_n = V_n$ **in** V_0", these variables are only visible in V_0. They mask all variables with the same names, possibly defined in enclosing scopes.

3.4 Static semantics of expressions (2): typing

The result sort of a pattern expression P, noted $sort\{P\}$, is defined as follows:

$$
\begin{array}{rcl}
sort\{X\} & = & \textit{sort of variable } X \\
sort\{X : S\} & = & S \\
sort\{\textbf{any } S\} & = & S \\
sort\{C(P_1, ..., P_n)\} & = & \textit{sort of the result of constructor } C \\
sort\{P_0 \textbf{ of } S\} & = & S \\
sort\{(P_0)\} & = & sort\{P_0\}
\end{array}
$$

The result sort of a value expression V, noted $sort\{V\}$, is defined as follows:

$$
\begin{array}{rcl}
sort\{X\} & = & \textit{sort of variable } X \\
sort\{C(V_1 ... V_n)\} & = & \textit{sort of the result of constructor } C \\
sort\{F(V_1 ... V_n)\} & = & \textit{sort of the result of function } F
\end{array}
$$

$$
\begin{aligned}
sort\{V_1 = V_2\} &= bool \\
sort\{V_1 <> V_2\} &= bool \\
sort\{V_0 \textbf{ of } S\} &= S \\
sort\{V_1 \textbf{ andthen } V_2\} &= bool \\
sort\{V_1 \textbf{ orelse } V_2\} &= bool \\
sort\{\textbf{if } V_0 \textbf{ then } V_0' ...\} &= sort\{V_0'\} \\
sort\{\textbf{case } V_0 \textbf{ in} ... -> V_1' ...\} &= sort\{V_1'\} \\
sort\{\textbf{assert } V_1, ..., V_n \textbf{ in } V_0\} &= sort\{V_0\} \\
sort\{\textbf{let } P_1 = V_1, ..., P_n = V_n \textbf{ in } V_0\} &= sort\{V_0\} \\
sort\{(V_0)\} &= sort\{V_0\}
\end{aligned}
$$

There are additional type-checking constraints:

- In a pattern expression of the form "$C(P_1, ..., P_n)$", each $sort\{P_i\}$ must be equal to the sort of the i-th argument of constructor C.
- In a pattern expression of the form "P_0 **of** S", one must have $sort\{P_0\} = S$.
- In a value expression of the form "$C(V_1, ..., V_n)$", each $sort\{V_i\}$ must be equal to the sort of the i-th argument of constructor C.
- In a value expression of the form "$F(V_1, ..., V_n)$", each $sort\{V_i\}$ must be equal to the sort of the i-th argument of constructor F.
- In a value expression of the form "$V_1 = V_2$" or "$V_1 <> V_2$", one must have $sort\{V_1\} = sort\{V_2\}$.
- In a value expression of the form "V_0 **of** S", one must have $sort\{V_0\} = S$.
- In a value expression of the form "V_1 **andthen** V_2" or "V_1 **orelse** V_2", one must have $sort\{V_1\} = sort\{V_2\} = bool$.
- In an "**if**" value expression, one must have $sort\{V_0\} = sort\{V_1\} = ... = sort\{V_n\} = bool$ and $sort\{V_0'\} = sort\{V_1'\} = ... = sort\{V_n'\} = sort\{V_{n+1}\}$.
- In a "**case**" value expression, one must have $sort\{V_0\} = sort\{P_1\} = ... = sort\{P_n\}$, and $sort\{V_1\} = ... = sort\{V_n\} = bool$, and $sort\{V_1'\} = ... = sort\{V_n'\}$.
- In an "**assert**" value expression, one must have $sort\{V_1\} = ... = sort\{V_n\} = bool$.
- In a "**let**" value expression, for all i, on must have $sort\{P_i\} = sort\{V_i\}$.

3.5 Dynamic semantics of expressions

Basically, the dynamic semantics is governed by the following principles:

- Pattern expressions are not meant to be evaluated: they only act as filters for pattern-matching and unification.
- As in standard LOTOS, value expressions are meant to be evaluated, provided that free variables are bound to known values.

- As in standard LOTOS, the evaluation of a given value expression leads to a unique, deterministic result[4].
- Unlike LOTOS, which is based on initial algebra semantics, the proposed evaluation strategy is essentially functional. It combines different evaluation strategies:
  - Any *use*-occurrence of a variable in a pattern or value expression is systematically replaced by the value bound to this variable.
  - In most cases, *call-by-value* (also, *eager evaluation*) is used. For instance, when evaluating value expressions such as "$C(V_1...V_n)$" or "$F(V_1...V_n)$", sub-expressions $V_1, ..., V_n$ are evaluated first (in an undefined order).
  - However, in some other cases, *call-by-need* (also, *lazy evaluation*) is necessary. This obviously happens with value expressions such as "**andthen**", "**orelse**", "**if**", and "**case**". Due to these cases, the semantics cannot be fully strict.
  - Also, some order of evaluation must be respected. For instance, sub-expressions $V_1, ..., V_n$ occurring in a "**let**" or "**assert**" expression must be evaluated before sub-expression V_0 is evaluated.
  - Finally, some value expressions (namely "**let**" and "**case**") rely on pattern-matching and even on conditional rewriting with priority.
- There is a special value, noted "$\perp$", which denotes a run-time error. This value is returned when the evaluation of a value expression fails[5]. There are several possible reasons for the evaluation to fail:
  - Entering into a non-terminating (diverging) computation, such as evaluating $F(0)$ for a function F defined as $F(X) := F(X+1)$;
  - Failing to find an appropriate pattern when evaluating a "**case**" or a "**let**" value expression;
  - Obtaining a false boolean guard when evaluating an "**assert**" expression.

Formally, the evaluation of a given expression V in a memory environment M (i.e., a partial application that maps a set of variables to their corresponding values) could be defined as a function $eval\{V, M\}$. The result returned by this function is either a value expression containing only constructors, or the undefined value $\perp$.

In the sequel, we do not define formally $eval\{V, M\}$, but we explain instead the meaning of some value expressions.

3.6 Informal description of the pattern-matching mechanism

The pattern-matching mechanism can be described as follows:

- Pattern "X" matches a single value, which is the value of variable X
- Pattern "$X : S$" matches any value of sort S and binds this value to the (new) variable X
- Pattern "**any** S" matches any value of sort S

[4] For the sake of simplicity, non-deterministic expressions, such as "**any bool**", are not accepted; otherwise, evaluating an expression would give birth to a transition system of all possible results. If necessary, the behaviour part can be used to describe such specific situations.

[5] Even in this case, the evaluation leads to a deterministic result.

- Pattern "$C(P_1, ..., P_n)$" matches any value expression of the form $C(v_1, ..., v_n)$ such that each sub-pattern P_i also matches the sub-value v_i.
- Pattern "P_0 **of** S" has the same effect as pattern "P_0"
- Pattern "(P_0)" has the same effect as pattern "P_0"

It is worth noticing that this mechanism is closely related to the notion of *value matching* used in the behaviour part of LOTOS, when it is necessary to match an "!"-offer against an "?"-offer.

3.7 Informal description of the "case" operator

The semantics of a given "**case**" expression V of the form:

case V_0 **in**
$P_1[V_1] -> V_1'$
...
$P_n[V_n] -> V_n'$
endcase

can be explained as follows:

- First, V_0 is evaluated, leading to a result v_0. If v_0 is equal to $\perp$, then V evaluates also to $\perp$ (strict semantics).
- Otherwise, the algorithm searches for the smallest $i \in \{1, ..., n\}$ such that pattern P_i matches v_0 and such that the boolean guard V_i (if present) evaluates to *true*.
- If no such i can be found, or if some guard V_i evaluates to $\perp$, then V evaluates also to $\perp$ (strict semantics).
- If some i is found, then V_i' is evaluated. V evaluates to the result of V_i'.
- When values V_i and V_i' are evaluated, the variables of $vars\{P_i\}$ (defined in P_i and used in V_i and V_i') are replaced by their values.
- Note: there is no need for a "**otherwise**" or "**default**" clause before "**endcase**" to order to all catch missing cases. This can be done by having the last pattern P_n be "**any** S", where S is equal to $sort\{V\}$.

For instance, the boolean function `AND` could be defined in many different ways using the proposed "**case**" constructs:

```
function AND (X, Y:BOOL) : BOOL is
        case X in
        true -> Y
        false -> false
        endcase
endfunc

function AND (X, Y:BOOL) : BOOL is
        case X in
        true -> Y
        any bool -> false
```

```
        endcase
endfunc

function AND (X, Y:BOOL) : BOOL is
        case X in
        true -> Y
        Z:bool -> false
        endcase
endfunc

function AND (X, Y:BOOL) : BOOL is
        case X in
        Z:bool [Z = true] -> Y;
        Z:bool [Z = false] -> Z
        endcase
endfunc

function AND (X, Y:BOOL) : BOOL is
        case X in
        true -> case Y in
                true -> true
                false -> false
                endcase
        false -> false
        endcase
endfunc
```

The proposed "**case**" construct is extremely powerful, since it combines different mechanisms into a single, unified framework:

- Keeping in mind that all existing LOTOS tools implement ACTONE equations as rewrite rules, it is easy to translate sets of equations (aka rewrite rules) into "**case**" expressions.

  Due the possibility to declare variables in patterns (i.e., the "$X : S$" patterns), "**forall**" clauses are no longer necessary to declare quantified variables.

  The proposed "`case`" is more powerful than current (rewrite-oriented dialects of) ACTONE, because it allows "`case`" expressions to be arbitrarily nested, which is not possible with sets of rewrite rules. To express nested "**case**s" in ACTONE, it is necessary to define auxiliary functions, which proves to be tedious in practice.

- This "**case**" construct is also more powerful than the two similar constructs found in ML:
  - ML has a "`case`" construct (with the "`_`" symbol to denote the "**otherwise**" clause) [Wat90, p. 33], but it only applies to constant values (i.e., a very limited form of pattern-matching).
  - ML allows functions to be defined by pattern-matching on the structure of their arguments [Wat90, p. 233]. However, pattern-matching constructs
    - cannot be nested,
    - are limited (a single pattern matching per function definition),
    - only allow "one level" simple-patterns,
    - do not have an "**otherwise**" clause,

* do not allow boolean guards[6].

Our proposed "**case**" construct generalizes both ML constructs, none of which has an equivalent expressiveness. For instance, our construct allows to define simply a function which tests whether the size of a list is greater or equal to 2 (which is not so easy in ML):

```
function SIZE_2 (L:LIST) : BOOL is
        case L in
                CONS (any ITEM, CONS (any ITEM, any LIST)) -> true
                any LIST -> false
        endcase
endfunc
```

3.8 Informal description of the "assert" operator

- All boolean guards $V_1, ..., V_n$ are evaluated. If at least one of them evaluates to $\perp$ or to *false*, then V evaluates to $\perp$.
- If each V_i evaluates to *true*, then V_0 is evaluated. The result obtained for V_0 is also the result of V.

Note: the expression "**assert false in** V", where V is any value expression, provides an explicit representation for the undefined value $\perp$.

The "**assert**" operator has several roles:

- It 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
```

- It can also express post-conditions relating 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
```

3.9 Informal description of the "let" operator

The semantics of a given "**let**" expression V of the form:

$$\textbf{let } P_1 = V_1, ..., P_n = V_n \textbf{ in } V_0$$

can be explained as follows:

[6] Therefore, translating ACTONE sets of conditional equations in ML might be difficult.

- First, all expressions $V_1, ..., V_n$ are evaluated. If at least one of them evaluates to $\perp$, then V also evaluates to $\perp$ (strict semantics).
- Otherwise, the obtained results are filtered against the respective patterns $P_1, ..., P_n$. If at least, one matching fails, then V evaluates to $\perp$.
- Otherwise, V_0 is evaluated by replacing all free variables in $vars\{P_1\} \uplus ... \uplus vars\{P_n\}$ by their corresponding values. The result obtained for V_0 is also the result of V.

Clearly:

- Having a "**let**" construct in value expressions is very useful since it allows to avoid redundant computations, by storing intermediate results in auxiliary variables. A "**let**" construct also exists in ML. In ACTONE, there is no "**let**" construct, it is often necessary to define auxiliary functions, which obscures the data type descriptions.
- The "**let**" construct which already exist in the behaviour part of LOTOS is a particular case of the proposed "**let**" construct. This particular case is obtained when each pattern P_i has the form $X_i : S_i$.

  Note: Rigorously speaking, there is a form of "**let**" with multiple variables (such as in "**let** $X_1, X_2 : S$ **in** ...") which can no longer be expressed with the proposed construct. But this form of "**let**" is completely useless.
- The proposed "**let**" construct is also much more general, since it allows *value destructuration*, thus avoiding to use explicit projection functions. In the following example:

```
let Failures (TCEstablishment : Prob,
              Transfer : Prob,
              TCResilience : Prob,
              TCRelease : Prob)
    = V in
    TCEstablishment * Transfer * TCResilience * TCRelease
```

  the value expression V will be evaluated, leading to a result of the form:

$$\texttt{Failures}\ (p_1, p_2, p_3, p_4)$$

  . The values p_1, p_2, p_3, p_4 will be extracted and respectively assigned to the free variables `TCEstablishment`, `Transfer`, `TCResilience` and `TCRelease`.

Note: reasonably, there should be at least one free variable in each $vars\{P_i\}$.

References

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

[Pec94] Charles Pecheur. Extended data types. Technical Report, University of Liège, July 1994. Annex H of ISO/IEC JTC1/SC21/WG1 N1349 Working Draft on Enhancements to LOTOS.

[Wat90] David A. Watt. *Programming Language Concepts and Paradigms.* International Series in Computer Science. Prentice-Hall, New-York, 1990.