French-Romanian comments regarding some proposed features for E-LOTOS data types[*]

Hubert Garavel Mihaela Sighireanu
INRIA Rhône-Alpes
VERIMAG — Miniparc-ZIRST
rue Lavoisier
38330 MONTBONNOT ST MARTIN
FRANCE

December 1995

Abstract

This report investigates some features that have been proposed as design choices for the E-LOTOS datatype language. Examples of these features are: polymorphism, type inference, anonymous records, and functions/constructors having a single argument. We analyze the consequences of these features and exhibit a number of defects affecting expressiveness, simplicity, compatibility with existing LOTOS, interoperability with other languages and efficiency of implementations. We therefore advise the E-LOTOS Committee not to include such undesirable features into the future E-LOTOS addendum.

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

Contents

1 Introduction

This report is based upon two existing documents:

- Annex A of the output document of the E-LOTOS meeting in Ottawa [JGL+95],
- and an input contribution to the E-LOTOS meeting in Liège [Jef95].

The first document contains two main approaches for the E-LOTOS datatype language. These two approaches are defined in Sections 4, 5, and 6 of [JGL+95]. The essential differences between these two approaches are the following:

- The first approach is compatible with current LOTOS. It does not support polymorphism (generic types are used instead) nor type inference. Tuples and records have to be named. Constructors and functions can have zero, one or more arguments. Overloading is allowed.
- The second approach is based on SML. It includes polymorphism, type inference, tuples and records as first-class citizen (*anonymous records*). All functions and constructors have a single argument. It forbids overloaded functions.

The second document [Jef95] develops and formalizes the second approach, with some modifications. We focus here on the data type proposal given in [Jef95] and we leave other considerations (behavioural language, data abstraction, and gates as first-class citizen) for further discussion.

In this report, we analyze the main differences between those two approaches and their practical consequences.

2 About polymorphism

A detailed criticism of polymorphism can be found in Section 5.2.10 (pages 23–24) of [JGL+95]. We remind here the main arguments against introducing polymorphism in E-LOTOS:

1. It would imply the suppression of overloading facilities that currently exist in LOTOS (see below in Section 3). It would go against the goal of compatibility with LOTOS defined in the scope of the New Work Item.
2. The existing type-checking algorithm for polymorphism is significantly more complex than the one for overloading (12 pages versus 4 pages in [ASU86], 8 pages versus 3 pages in [WM94]). This would increase the complexity of both E-LOTOS static semantics and E-LOTOS type-checkers.
3. Moreover, type-checking for overloading can be performed in linear-time, whereas type-checking for polymorphism is exponential. There a risk that large, polymorphism-based protocol descriptions could not be type-checked in a reasonable amount of time.
4. Apart from the languages of the ML family, polymorphism is not used in any major programming or specification language. Although interesting ML features (e.g., union types, pattern-matching) have been reused in recent languages, this is not the case with polymorphism.
5. Lack of polymorphism has not been a usual complaint heard from people actually working with LOTOS. In this respect, it is worth noticing that none of the proposals for E-LOTOS enhancements listed in Section 3 of [JGL+95] suggest to enbody polymorphism as a desirable feature. The reason for this is explained in the next items.
6. The genericity features of ACTONE already cover most of the functionalities provided by polymorphism, by allowing the definition of sorts and functions parameterized by one or several

formal sorts. The same approach can be found in many algebraic languages, as well as in functional languages like Opal. Although the ActOne syntax for actualization is not very user-friendly, it could be improved. Such changes would rather affect the module system than the type system of the core language itself.

7. If polymorphism is included in E-Lotos, it is feared that it will be largely redundant with genericity features. For instance, a specifier wanting to define a stack (or a fifo queue) of parameterized items will be offered two possibilities: either declaring a polymorphic stack type, or declaring a generic stack type (using ActOne-like generic modules or SML-like functors).

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

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

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

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

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

We also want to point out two new considerations:

1. We observe that the latest SML-based proposal [Jef95] does not include polymorphism.

2. We notice that polymorphism cannot be simply extended to the behaviour part of Lotos. If polymorphic functions are allowed, it would be suitable (for symmetry reasons) to have polymorphic processes also. But this would create problems, as several values of different types can be sent on the same gate. For instance, one could imagine a polymorphic one-slot buffer defined as:

```
process BUFFER [INPUT, OUTPUT] : noexit :=
   INPUT ?X:'T;
      OUTPUT !X;
         BUFFER [INPUT, OUTPUT]
endproc
```

   where `X` is a variable having the polymorphic type `'T`. However, trouble ensues if we connect this buffer in the following environment:

```
   BUFFER [SEND, OUTPUT]
   |[SEND]|
   SENDER [SEND]
where
   process SENDER [SEND] : noexit :=
      SEND !0;
         SEND !false;
            stop
```

```
    endproc
```

Depending on the instant, the value sent on gate `SEND` can be either a natural number or a boolean. Therefore the `X` variable of the buffer has to be sometimes a boolean, and sometimes a natural number. This contradicts the ML assumption that each expression has always a single (mono)type and implies that type-checking has to be deferred until run-time.

Having polymorphic processes (such as the polymorphic buffer) would not be compatible with the decision of the E-LOTOS Committee to have static type-checking.

3 About overloading

The proposed SML-based approach would have the undesirable effect of removing overloading (which already exists in the LOTOS standard) from E-LOTOS. A discussion about the implied drawbacks can be found in Section 5.2.9 page 19 of [JGL+95]. We reproduce here the main arguments:

1. Overloading is primarily intended for notational convenience. It is not obvious that removing overloading from E-LOTOS is the right way to obtain "a more user-friendly notation for datatype descriptions" (which is the goal defined in the scope of the new Work Item).

2. Overloading exists in most computer languages. Some languages (like Fortran, Algol, Pascal, C, and SML) only allow overloading for built-in operators (e.g., + on integers, and + on reals).

   Some other recent languages (such as Ada, C++, Eiffel, Opal, etc.) have made overloading uniformly available for both built-in and defined functions. This results in a greater convenience for the programmer to exploit and fewer special cases to memorize.

3. User-defined, overloaded functions do not prevent type-checking from being done at compile time (this was done for the first time in 1962 for APL). Overloading does not add much complexity to languages and compilers, since overloading treatment is usually isolated in a well-defined part of static semantics, without impact on dynamic semantics.

4. Well-known, efficient algorithms exist to perform type-checking and solve overloading simultaneously [ASU86]. These algorithms use either two passes (as in the case of Ada) or even a single bottom-up pass [Bak82].

5. We advocate that overloading supports compositionality, because it does not require that all functions have different names, thus reducing the risk of name clashes when importing new modules. For instance, if two modules, a *fifo_queue* module and a *stack* module, both of them exporting an *is_empty* function with different profiles, are imported simultaneously in a third module, no name clash will occur, and both functions will be coexist and be accessible.

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

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

   Without overloading, it is also necessary to actualize all these functions by giving them new names (e.g., is_empty_packet_queue, is_empty_message_queue, etc.), which is rather cumbersome.

7. Overloading is existing practice in LOTOS. Forbidding overloading in E-LOTOS would raise difficult compatibility issues with the current standard.. In such case, an algorithm should be provided to translate existing LOTOS descriptions into ones without overloading.

8. The "rich-term syntax" proposed by Charles Pecheur [Pec94] relies on the existence of overloading.

We also want to add the following new arguments:

1. We do not agree with Alan Jeffrey's statement that overloading is more or less incompatible with type theory.

   This is not a universal truth: although some theorem-provers like HOL and COQ do not accept overloading, there exist other theorem-provers based on type theory, such as PVS and ISABELLE, which support overloading very well.

2. From our own experience, we claim that most LOTOS descriptions standardized by ISO make plain use of overloading. It is worth noticing that both constructors and/or defined functions (including selector functions) happen to be overloaded. We illustrate this fact by giving examples taken from the CCR and OSI-TP protocols:

   - Example 1 (excerpt from CCR protocol):

```
sorts   general
opns
        null (*! constructor *) : -> general
        succ (*! constructor *) : general -> general
sorts   key
opns
        0 (*! constructor *) : -> key
        succ (*! constructor *) : key -> key
sorts   user_data
opns
        null (*! constructor *) : -> user_data
        succ (*! constructor *) : user_data -> user_data
```

   - Example 2 (excerpt from CCR protocol):

```
get_bs : branch_identifier -> branch_suffix
get_bs : CPDU -> branch_suffix

get_aet : name -> ae_title
get_aet : name, ccr_ver, ae_title, ae_title -> ae_title

name (*! constructor *) : ae_title -> name
name (*! constructor *) : side -> name

get_aais : CSP -> atomic_action_suffix
get_aais : CPDU -> atomic_action_suffix

get_ud : CSP -> PDUqueue (* User Data *)
get_ud : ASP -> PDUqueue (* User Data *)
get_ud : PSP -> PDUqueue (* User Data *)
get_ud : CPDU -> user_data
```

```
get_rs : CSP -> requestor_recovery_state
get_rs : ASP -> result_source
get_rs : CPDU -> requestor_recovery_state
get_rr : CSP -> responder_recovery_state
get_rr : CPDU -> responder_recovery_state

sp (*! constructor *) : CSP -> SP
sp (*! constructor *) : ASP -> SP
sp (*! constructor *) : PSP -> SP
```

- Example 3 (excerpt from Osi-Tp protocol):

```
suffix : atomic_action_identifier -> suffix
suffix : branch_identifier -> suffix
suffix (*! constructor *) : OctetString -> suffix
suffix (*! constructor *) : Integer ->  suffix

no_reason_given (*! constructor *): -> service_user_diagnostic
no_reason_given (*! constructor *): -> service_provider_diagnostic

is_user, is_provider : PDU -> bool
is_user, is_provider : ServicePrim -> bool
is_user, is_provider : a_associate_result_source -> bool
is_user, is_provider : a_abort_source -> bool

tp_abort_ri (*! constructor *) : tp_abort_type, PDUqueue -> PD
tp_abort_ri (*! constructor *) : tp_abort_type, tp_P_abort_diagnostic -> PDU

purging (*! constructor *) : nat, nat -> purge_sort
purging (*! constructor *) : bool -> purge_sort

_._ (*! constructor *) : PDUlist, PDUkey -> PDUlist
_._ : PDUkey, PDUkey -> PDUlist
_._ (*! constructor *) : PDU, PDUqueue -> PDUqueue

_+_ : PDUqueue, PDU -> PDUqueue
_+_ : PDU, PDU -> PDUqueue

_.=_ (*! constructor *) : nat, bool -> assignment
_.=_ (*! constructor *) : range, bool -> assignment
_.=_ (*! constructor *) : range, service_info -> assignment

_&=_ (*! constructor *) : nat, bool -> assignment
_&=_ (*! constructor *) : range, bool -> assignment
```

- Example 4 (excerpt from Osi-Tp protocol):

```
tp_begin_dialogue_ri (*! constructor *) :
  TPSU_title_Opt, TPSU_title_Opt, functional_units, bool_Opt,
  tp_begin_dialogue_confirmation, nat, correlator_Opt, PDUqueue -> PDU

tp_begin_dialogue_ri (*! constructor *) :
  functional_units, nat, channel_utilization, correlator_Opt -> PDU

tp_begin_dialogue_rc (*! constructor *) :
  functional_units_Opt, tp_begin_dialogue_result,
  tp_begin_dialogue_diagnostic_Opt, nat, PDUqueue -> PDU

tp_begin_dialogue_rc (*! constructor *) :
  tp_begin_dialogue_result, tp_begin_channel_diagnostic_Opt, nat -> PDU
```

4 About type inference

1. As regards type inference, there is a persistent ambiguity in [JGL+95] and [Jef95] Although it is stated in the abstract of [Jef95] that the "core functional language [...] is explicitly typed and monomorphically typed", the examples given page 14, 16, 18, 19, 23 of [Jef95] rely on type inference, which needs to be clarified.

2. Type inference is closely related to polymorphism. If polymorphism is not introduced, there is no need for type inference.

3. As regards the consequences of type inference, we fully agree with the point of view expressed in [Wat90, page 139]:

   > "ML adopts a *laissez-faire*, attitude to typing. The programmer can voluntarily state the type of a declared entity, or leave the compiler to infer the type.
   >
   > In longer pieces of program than [*the two-line program given as example page 139*], however, it is unwise to rely too much on type inference. Consider a very large function definition, written without any types being stated explicitly. A (human) reader might have to scan pages simply to discover the type of the function. Even the author of the function definition can get into difficulties: a slight programming error in the function body might confuse the compiler, causing it to produce obscure error messages, or even to infer a different type from the one intended by the programmer. So explicitly stating types, even if redundant, is good programming practice."

5 About anonymous records and single-argument functions

There are other features in [Jef95] which desire to be closely examined:

- Records are first-class citizen, which means that it is possible to define record types without naming them explicitly.
- Functions have a single argument and a single result. Both the argument and the result can be a record.
- Similarly, constructors have a single argument and a single result.

For a number of reasons exposed in the following sections, we believe that introducing such features in E-LOTOS would be a very poor design choice.

5.1 No added expressiveness

It should be clear that, from a purely theoretical point of view, introducing anonymous records in E-Lotos does not add any extra expressiveness to the language.

As the number of anonymous records in a given description is necessarily finite, it is always possible to translate any description with anonymous records into an equivalent description in which all distinct records types are given unique type identifiers.

5.2 A limited and questionable convenience

From a practical point of view, one might see two advantages to anonymous records:

- Convenience for the users, which can use structured types without declaring them: this argument is a very questionable one, as it will be discussed in Section 5.9.
- Possibility to declare easily functions that return several results: this argument is more convincing, as functions with multiple results do not directly exist in standard Lotos. If we consider the example of a function that search an item in a table, it is easier to write:

  ```
  function SEARCH_ITEM (I:ITEM, T:TABLE) : (FOUND:boolean, KEY:natural) is
     if { exists K:natural such that T [K] = I }
     then (true, K)
     else (false, 0)
     endif
  endfunc
  ```

  rather than:

  ```
  type RESULT_OF_SEARCH is
     PAIR (FOUND:boolean, KEY:natural)
  end

  function SEARCH_ITEM (I:ITEM, T:TABLE) : RESULT_OF_SEARCH is
     if { exists K:natural such that T [K] = I }
     then PAIR (true, K)
     else PAIR (false, 0)
     endif
  endfunc
  ```

  We believe that this convenience is perhaps the only justification for introducing anonymous records in Lotos. However, we will show in Section 5.11 that functions returning records are not appropriate if we want E-Lotos to interoperate smoothly with other major computer languages, including those standardized by Iso.
- Furthermore, we will explain in Sections 5.3 to 5.6 why the presence of anonymous records in E-Lotos prevents from having convenient features (such as selectors and updaters for union types) which have proven to be necessary in standardized Lotos descriptions of Iso protocols and services.

5.3 Lack of convenience: heavy record notations

The notation for record values proposed in [Jef95] is cumbersome, because all field labels have to be explicitly mentioned. For instance, let's consider the following type definition:

```
datatype GEOMETRICAL_SHAPE :=
   CIRCLE of {RADIUS:real}
 | RECTANGLE of {LENGTH:real, WIDTH:real}
endtype
```

Every time the constructor `RECTANGLE` is invoked, it is necessary to recall the labels of its fields; one must write "`RECTANGLE {LENGTH=10, WIDTH=20}`" or "`RECTANGLE {WIDTH=20, LENGTH=10}`".

It is not possible to write "`RECTANGLE {10, 20}`" simply, because this would introduce an ambiguity on the type of the record.

It is to be feared that specifiers will shorten as much as possible the field labels in order to keep short notations or even to remove these labels by using the SML "tuple" shorthand.

In both cases, the resulting specifications will be poorly documented (this problem has already been discussed in Section 5.2.5 of [JGL+95]).

5.4 Lack of convenience: the "..." notation

The proposal made in [Jef95] does not contain the "..." notation in patterns [JGL+95]. This abbreviated notation allows to elide constructor arguments (i.e., record fields) which are of no interest in the pattern. For instance, it can be useful for selecting the 18th and 22th bits of a 64-bit word.

The "..." notation is trivial to implement using the first approach proposed in [JGL+95]. However, in an SML-based approach, it raises the difficult problem of "polymorphic records" mentioned in Section 5.2.11 of [JGL+95]. This problem is not addressed in the simplistic semantics given in [Jef95].

Also, the "..." notation might be also necessary to write default values in constructor or function calls. For instance, this could be useful to write a 64-bits record whose 18th and 22th bits are equal to 1, the remaining bits being set to 0:

```
Octet (B18 := 1, B22 := 1, ... := 0)
```

5.5 Lack of convenience: selectors for union types

Moreover, having records as first-class citizen disallows "extended selectors". The selectors (or projection functions) proposed in Section 6.8 of [JGL+95] allow to extract fields of record types. However, experience indicates that it is also necessary to extract fields from union types, a problem that is not addressed in [Jef95]. For instance, given the following type definition:

```
type PACKET is
   REQUEST (SOURCE:ADDRESS, TARGET:ADDRESS, CONTENTS:STRING)
 | INDICATION (SOURCE:ADDRESS, TARGET:ADDRESS)
 | DATA (SOURCE:ADDRESS, CONTENTS:STRING)
endtype
```

if `P` is a value of type `PACKET`, one may wish to write `P.SOURCE` or `P.CONTENTS` to extract the source and contents fields; the latter function is undefined if `P` is an indication. If selectors are only available for record values, the extended selectors can be defined as follows:

```
P.SOURCE == case P in
                 REQUEST X -> X.SOURCE
               | INDICATION X -> X.SOURCE
               | DATA X -> X.SOURCE
```

```
             endcase

   P.CONTENTS == case P in
                    REQUEST X -> X.CONTENTS
                  | DATA X -> X.CONTENTS
                 endcase
```

However, for real specifications it is necessary to have extended selectors built in E-LOTOS. For instance, the LOTOS description of the CCR protocol has 56 extended selectors, some of which require up to 16 equations. Similarly, the LOTOS description of the OSI-TP protocol has 187 selectors, some of which require up to 26 equations.

The example below is taken from the OSI-TP description. It shows the definition of the **get_ud** function that extracts fields **ud** from a sort **ServicePrim** having 54 constructors; this function is defined for 32 constructors only and undefined for the remaining ones:

```
  get_ud : ServicePrim -> PDUqueue
    forall
      acn : application_context_name,
      aais :   suffix,
      brid : branch_identifier,
      bdc : tp_begin_dialogue_confirmation,
      bdr : tp_begin_dialogue_result,
      bddO : tp_begin_dialogue_diagnostic_Opt,
      btO : bool_Opt ,
      hr : heuristic_report,
      iapiO :  API_id_Opt,
      iaeiO :  AEI_id_Opt,
      iaeqO :  AE_qual_Opt,
      iaptO :  AP_title_Opt,
      ittO :   TPSU_title_Opt,
      map : mapping,
      qosO : quality_of_service_Opt,
      rapiO :  API_id_Opt,
      rapt :   AP_title,
      raeqO :  AE_qual_Opt,
      raeiO :  AEI_id_Opt,
      rbk : bool ,
      rsri : recovery_state,
      rsrc : recovery_response ,
      rttO : TPSU_title_Opt,
      sfu : functional_units,
      ud : PDUqueue

  ofsort PDUqueue
    get_ud( TPBeginDialogueReq(
        ittO, rapt, rapiO, raeqO, raeiO, rttO, sfu, qosO, acn, btO, bdc, ud)) =
      ud;
    get_ud( TPBeginDialogueInd(
        iaptO, iapiO, iaeqO, iaeiO, ittO, sfu, btO, bdc, ud)) =
      ud;
    get_ud(TPBeginDialogueRsp(bdr, ud)) = ud;
    get_ud(TPBeginDialogueCnf(fuO, bdr, bddO, ud, rbk)) = ud;
    get_ud(TPUAbortReq(ud)) = ud;
    get_ud(TPUAbortInd(ud, rbk)) = ud;
    get_ud(
      AFBeginDialogueReq(itt0, rtt0, sfu, bt0, bdc, dcc0, lpi0, ud)) =
      ud;
    get_ud(AFBeginDialogueInd(itt0, rtt0, sfu, bt0, bdc, dcc0, lpi0, ud)) = ud;
    get_ud(AFBeginDialogueRsp(fu0, bdr, bdd0, dcc0, map, ud)) = ud;
    get_ud(AFBeginDialogueCnf(fu0, bdr, bdd0, dcc0, map, ud)) = ud;
    get_ud(AFAbortReq(user, map, ud)) = ud;
    get_ud(AFAbortInd(user, map, ud)) = ud;
    get_ud(AFAbortAndHeuristicReportReq(map, hr, ud)) = ud;
    get_ud(AFAbortAndHeuristicReportInd(map, hr, ud)) = ud;
    get_ud(CPrepareReq(ud)) = ud;
    get_ud(CPrepareInd(ud)) = ud;
    get_ud(CCommitReq(ud)) = ud;
    get_ud(CCommitInd(ud)) = ud;
    get_ud(CCommitRsp(ud)) = ud;
    get_ud(CCommitCnf(ud)) = ud;
    get_ud(CRollbackReq(ud)) = ud;
    get_ud(CRollbackInd(ud)) = ud;
    get_ud(CRollbackRsp(ud)) = ud;
    get_ud(CRollbackCnf(ud)) = ud;
    get_ud(CRecoverReq(rsri, aaid, brid, ud)) = ud;
    get_ud(CRecoverInd(rsri, aaid, brid, ud)) = ud;
    get_ud(CRecoverRsp(rsrc, aaid, brid, ud)) = ud;
    get_ud(CRecoverCnf(rsrc, aaid, brid, ud)) = ud;
    get_ud( PTokenGiveReq(ud)) = ud;
    get_ud(PTokenGiveInd(ud)) = ud;
    get_ud(PDataReq(ud)) = ud;
    get_ud(PDataInd(ud)) = ud;
```

5.6 Lack of convenience: updaters for union types

Similarly, the record updaters proposed in [Jef95] are not sufficient for real-life protocols:

- They are limited to record types, whereas they should also work for union types. In the case of union types with multiple constructors, it is necessary to use an explicit "`case`" statement for specifying the effects of updating on each constructor.
- They do not allow to reference the value of the field to be modified. For instance, if one wants to increment the field `X` of a record (or a union) `P`, if is not possible to write "`P.{X := X+1}`" simply, one has to write "`P.{X := P.X +1}`", which is only acceptable if `P` is a named value.

"Extended updaters" are badly needed in real-life specification. Let's consider for instance, in the LOTOS description of the LAPB protocol, there is a `makecorruptadr` function that updates the field `address` of a value `f` of type `eframe`. This function is defined as follows:

```
m (*! constructor *) : frame,inforr,corrlength,lesslength,abortf -> eframe
makecorruptadr    : eframe -> eframe
forall f : eframe
     ofsort eframe
       is_i(get_f(f)) => makecorruptadr(f) = m(make_iframe(flg1(f),
           corrupt_adr(adrs(f)), ctrl(f), pack(f), fcss(f),
           flg2(f)), get_infor(f), get_c(f), get_l(f), get_a(f));
       is_rr(get_f(f)) => makecorruptadr(f) = m(make_rr(flg1(f),
           corrupt_adr(adrs(f)), ctrl(f), fcss(f),
           flg2(f)), get_infor(f), get_c(f), get_l(f), get_a(f));
       is_rnr(get_f(f)) => makecorruptadr(f) = m(make_rnr(flg1(f),
           corrupt_adr(adrs(f)), ctrl(f), fcss(f),
           flg2(f)), get_infor(f), get_c(f), get_l(f), get_a(f));
       is_rej(get_f(f)) => makecorruptadr(f) = m(make_rej(flg1(f),
           corrupt_adr(adrs(f)), ctrl(f), fcss(f),
           flg2(f)), get_infor(f), get_c(f), get_l(f), get_a(f));
       is_sabm(get_f(f)) => makecorruptadr(f) = m(make_sabm(flg1(f),
           corrupt_adr(adrs(f)), ctrl(f), fcss(f),
           flg2(f)), get_infor(f), get_c(f), get_l(f), get_a(f));
       is_sabme(get_f(f)) =>
           makecorruptadr(f) = m(make_sabme(flg1(f),
           corrupt_adr(adrs(f)), ctrl(f), fcss(f),
           flg2(f)), get_infor(f), get_c(f), get_l(f), get_a(f));
       is_disc(get_f(f)) => makecorruptadr(f) = m(make_disc(flg1(f),
           corrupt_adr(adrs(f)), ctrl(f), fcss(f),
           flg2(f)), get_infor(f), get_c(f), get_l(f), get_a(f));
       is_ua(get_f(f)) => makecorruptadr(f) = m(make_ua(flg1(f),
           corrupt_adr(adrs(f)), ctrl(f), fcss(f),
           flg2(f)), get_infor(f), get_c(f), get_l(f), get_a(f));
       is_dm(get_f(f)) => makecorruptadr(f) = m(make_dm(flg1(f),
           corrupt_adr(adrs(f)), ctrl(f), fcss(f),
           flg2(f)), get_infor(f), get_c(f), get_l(f), get_a(f));
       is_frmr(get_f(f)) => makecorruptadr(f) = m(make_frmr(flg1(f),
           corrupt_adr(adrs(f)), ctrl(f), reasfrmr(f), fcss(f),
           flg2(f)), get_infor(f), get_c(f), get_l(f), get_a(f));
```

Using extended updaters, this function could be defined in a more readable, shorter, safer way:

```
    function makecorruptadr (F: eframe) : eframe is
        F.{FR := FR.{A := corrupt_adr(A)}}
    endfunc
```

There are other examples in the Lapb protocol:

```
    function makeundefctl (F: eframe) : eframe is
        F.{FR := FR.{C := undef (C)}}
    endfunc

    function makecorruptfc (F: eframe) : eframe is
        F.{FR := FR.{FCC := corrupt_fcs (FCC)}}
    endfunc

    function makenotcorrlength (F: eframe) : eframe is
        F.{C := incorrect}
    endfunc

    function makelesslength (F: eframe) : eframe is
       F.{L := less}
    endfunc
```

and in the Osi-Tp protocol:

```
   function inc_dcc (CO: correlator_Opt) : correlator_Opt is
      CO.{N := N + 1}
   endfunc
```

```
function set_dccO (dcc1: correlator_Opt, SP: AF_ServicePrim) : AF_ServicePrim is
     SP.{DCCO := dcc1}
endfunc

function set_lpiO (lpi1: correlator_Opt, SP: AF_ServicePrim) : AF_ServicePrim is
     SP.{LPIO := lpi1}
endfunc

 function set_map (map1: mapping, SP: AF_ServicePrim) : AF_ServicePrim is
     SP.{MAP := map1}
endfunc

 function set_brid (BI: branch_identifier, E: assoc_entry) : assoc_entry is
     E.{BRIDO := Opt (BI)}
endfunc

 function remove_SentRcvd (K: LookUpKey, E: assoc_entry) : assoc_entry is
     E.{SENT := SENT - K}
endfunc

function remove_Pending (K: LookUpKey, E: assoc_entry) : assoc_entry is
     E.{PEND := PEND - K}
endfunc

function change_pending_to_sent (K: LookUpKey, E: assoc_entry) : assoc_entry is
     E.{SENT := SENT + get (K, PEND), PEND := PEND - x}
endfunc

function re_attach (SP: ServicePrim, E: assoc_entry) : assoc_entry is
     E.{SENT := nilSP + SP, PEND := nilSP}
endfunc

function add_SentRcvd (SP: ServicePrim, E: assoc_entry) : assoc_entry is
     E.{SENT := SENT + SP}
endfunc

function add_Pending (SP: ServicePrim, E: assoc_entry) : assoc_entry is
     E.{PEND := PEND + SP}
endfunc

function set_flag (B: bool, E: assoc_entry) : assoc_entry is
     E.{FLAG :=B}
endfunc

function detach (E: assoc_entry) : assoc_entry is
     E.{ATTACH := false}
endfunc

function remove_commitSPs (E: assoc_entry) : assoc_entry is
     E.{SENT := remove (commitSP, SENT), PEND := remove (commitSP, PEND)}
endfunc

function end_purge (E: assoc_entry) : assoc_entry is
     E.{PURGE := end (PURGE)}
 endfunc

 function trans_init (E: assoc_entry) : assoc_entry is
      E.{SENT := SENT + TPBeginTransactionReq, PURGE := trans_init (PURGE)}
 endfunc

 function inc_purge (E: assoc_entry) : assoc_entry is
      E.{PURGE := inc (PURGE)}
 endfunc

 function dec_purge (E: assoc_entry) : assoc_entry is
      E.{PURGE := dec (PURGE)}
 endfunc

 function TPDone_owing (TPPM: tppm_info) : tppm_info is
      TPPM.{TDO := true}
 endfunc

 function TPDone_sent (TPPM: tppm_info) : tppm_info is
      TPPM.{TDO := false}
 endfunc

 function add_SentRcvd (SP: ServicePrim, TPPM: tppm_info) : tppm_info is
      TPPM.{SENT := SENT + SP}
 endfunc

function change_pending_to_sent (K: LookUpKey, TPPM: tppm_info) : tppm_info is
      TPPM.{SENT := SENT + get (K, PEND), PEND := PEND - K}
 endfunc

 function add_Pending (SP: ServicePrim, TPPM: tppm_info) : tppm_info is
      TPPM.{PEND := PEND + SP}
 endfunc

 function remove_SentRcvd (K: LookUpKey, TPPM: tppm_info) : tppm_info is
      TPPM.{SENT := SENT - K}
 endfunc

 function remove_Pending (K: LookUpKey, TPPM: tppm_info) : tppm_info is
      TPPM.{PEND := PEND - K}
 endfunc

 function change_state (TS1: tppm_state, TPPM: tppm_info) : tppm_info is
      TPPM.{TS :=TS1, TDO := is_dec_cmt (TS1) or is_dec_rbk (TS1)}
 endfunc

 function Qing (Info: sf_info) : sf_info is
    Info.{QING :=true}
 endfunc

 function TokOwed (Info: sf_info) : sf_info is
    Info.{TO :=true}
 endfunc
```

```
  function BidRspSent (Info: sf_info) : sf_info is
     Info.{BS :=true}
  endfunc

  function TokReq (Info: sf_info) : sf_info is
     Info.{CTRrue}
  endfunc

  function NoQ (Info: sf_info) : sf_info is
     Info.{QING :=false}
  endfunc

  function NotTokOwed (Info: sf_info) : sf_info is
     Info.{TO :=false}
  endfunc

  function NotTokReq (Info: sf_info) : sf_info is
     Info.{CTR :=false}
  endfunc

  function set_did (D: dc_id, Info: sf_info) : sf_info is
     Info.{DID :=D}
  endfunc

  function change_state (S: sacf_state, Info: sf_info) : sf_info is
     if (S = FREE) then
          Info.{TO := false, BS := false, SKT := false, CTR := false, STATE := S}
     else
          Info.{STATE := S}
     endif
  endfunc

  function change_lbdsent (DCC: correlator_Opt, Info: sf_info) : sf_info is
       Info.{LBDSENT := DCC}
  endfunc

  function change_lbdrec (DCC: correlator_Opt, Info: sf_info) : sf_info is
       Info.{LBDREC := DCC}
  endfunc

 function inc_lbdsent (Info: sf_info) : sf_info is
       Info.{LBDSENT :=inc_dcc (LBDSENT)}
  endfunc

  function inc_next_did (SI: service_info) : service_info is
       SI.{NEXT_DID := inc (NEXT_DID)}
  endfunc
```

5.7 Oddness of language constructs

Although the idea of having functions and constructors with a single argument and a single result might seem elegant at first sight, it turns out to introduce a lot of strange details, which are likely to be confusing for potential E-Lotos users:

1. Although both the argument and the result of a function can be records, this is not the case for constructors: the result of a constructor cannot be a record.

2. To express functions without parameters, it is necessary to introduce a new kind of objects, the *constants*.

3. For practical use (e.g., to write lists of actual parameters for a function), records are not appropriate: one has to introduce another concept, *tuples*, which are derived (but distinct) from records.

4. Constructors without arguments are handled in different ways. In [Jef95], these constructors are considered to have a single parameter, the empty record. However, in SML, there is a special case for "nullary" constructors, which are considered to be different from constructors whose argument is the empty record.

5. The same problem occurs for (infix) binary operators, which break the rule of functions having a single argument. In [Jef95], they are not considered. In SML, they are handled as a special case of functions with *two* arguments.

6. If C is a constructor and E an expression, there is a subtle difference between terms "$C\ E$" and "$C\ (E)$": in the former case, C is applied to the (scalar) expression E; in the latter, C is applied to a tuple having expression E as its single component. Both expressions are not identical with respect to type-checking!

   Every time one wants to define a one-argument constructor, one has to choose between either a named parameter enclosed in a record:

   ```
   datatype GEOMETRICAL_SHAPE :=
      CIRCLE of {RADIUS:real}
    | ...
   endtype
   ```

   or a "flat", anonymous parameter:

   ```
   datatype GEOMETRICAL_SHAPE :=
      CIRCLE of real
    | ...
   endtype
   ```

   It does not seem possible to have a single flat, named parameter. However, if there are several parameters, they must be named and enclosed in a record. Therefore, the fact of adding a new parameter to a one-argument constructor may require tedious changes if a "flat" approach was chosen initially.

7. Similarly, if F is a function, both expressions "$F\ E$" and "$F\ (E)$" are not the same !

8. In record expressions, there is a shorthand that allows not to write "$L = E$" if L and E are syntactically identical. This allows to elide field denotations of the form "$L = L$". However, this notation is strangle because both L's belong to distinct semantic identifier classes: the left one is a label identifier, the right one is a variable identifier.

9. According to [Jef95], it is possible to declare anonymous records, but it is not possible to give names to them! For instance, one can declare two variables `A` and `B` whose type is the same anonymous record:

   ```
   A : {X:int, Y:bool}
   B : {X:int, Y:bool}
   ```

but, surprisingly, it is not possible to write instead:

```
datatype S := {X:int, Y:bool} endtype
A : S
B : S
```

because the declaration of `S` is syntactically illegal. One must introduce a constructor with a unique name (since overloading is not allowed), e.g., `pair`, and write instead:

```
datatype S := pair of {X:int, Y:bool} endtype
A : S
B : S
```

Besides the existence of *anonymous types*, there also exist *"un-namable" types*, i.e., types which can never be given a type identifier! Such a bizarre feature denotes a non-orthogonal type system.

We notice that this impressive collection of oddities is not justified by any increased expressive power. If included in E-LOTOS*, it is to be feared that the "learning curve" of the language will be steep, and that teachers will spend their time trying to find an explanation for such unusual concepts.*

5.8 Complexity of semantics

Introducing anonymous records would make the static and dynamic semantics of E-LOTOS more complex:

- It introduces the notion of *type expressions*. In standard LOTOS, the type of each expression is simply a sort identifier. With anonymous records, the situation is different: the type of each expression is a *type expression*, a type expression being either a sort identifier, or a record of type expressions. Nested tupling is thus allowed.

- It introduces the notion of *structure equivalence* for type expressions. In standard LOTOS, two expressions have the same type iff their sort identifiers are identical (this is *name equivalence*). In SML, however, the situation is less simple, since a mixture of name equivalence and structure equivalence is used. For instance, in the following example:

  ```
  A : {X:int, Y:{Z:bool, T:bool}}
  B : {X:int, Y:{Z:bool, T:bool}}
  C : {Y:{T:bool, Z:bool}, X:int}
  datatype S := pair of {X:int, Y:{Z:bool, T:bool}} endtype
  D : S
  ```

  the three variables `A`, `B`, and `C` have the same type (which means that the compiler has to identify nested records that differ by field permutation), whereas `D` doesn't have the same type as `A`, `B`, and `C`, because it has the named type `S`.

Replacing sort identifiers with type expressions, and name equivalence by a mixture of name and structure equivalence might have side effects in the module part as well as in the behaviour part. These potential side effects should be carefully investigated.

5.9 Subversion of software methodologies

We observe that anonymous records and structure equivalence go against the recommended methodologies for software design and programming.

It is to be feared that *lazy* programmers and specifiers may take advantage of these features to write poorly documented descriptions, in which complex data are structured in (nested) records and manipulated by functions without naming the types of these data (as it is often seen in LISP programs). Such data turn out to be low-level *concrete types* rather than high-level abstractions.

This is a common concern for most programming languages designers: N. Wirth chose name equivalence in Pascal for this reason. It is to be noticed that the same decision was made for algorithmic languages like Ada, C, C++, etc.

Also, anonymous records and structure equivalence go against object-oriented methodologies for designing complex systems, which recommend to give explicit names to every data structure (i.e., object) manipulated by the software under design.

Finally, as [Jef95] does not allow to name anonymous records (see Section 5.7), re-engineering a description with anonymous types is a tedious process, implying many modifications. For instance, to document the following piece of code:

```
A : {X:int, Y:bool}
B : {X:int, Y:bool}
...
if (A = {X=0, Y=false}) and (B = {X=1, Y=true}) then ...
```

one must introduce a new constructor `pair` in many different places:

```
datatype S := pair of {X:int, Y:bool} endtype
A : S
B : S
...
if (A = pair {X=0, Y=false}) and (B = pair {X=1, Y=true}) then ...
```

5.10 Absence of compatibility with standard LOTOS

Compatibility of E-LOTOS with respect to LOTOS is an explicit requirement stated in the scope of the New Work Item.

As the E-LOTOS Committee decided to replace ACTONE with a new datatype language, this compatibility requirement becomes the following: the description obtained by taking a valid LOTOS description and translating its data types into E-LOTOS data types should be a valid E-LOTOS description.

This approach is exemplified in one of AFNOR's contributions to the Liege meeting: it should be possible to update existing LOTOS descriptions (such as those of the CCR protocol, the OSI-TP protocol, the LAPB protocol, etc.) by updating their data part only, leaving the behaviour part as is.

However, it is not sure that the approach presented in [Jef95] meets these requirements, since it makes no provision to support:

- constructors and functions with several arguments: it seems that every constructor (or function) call of the form $C(E_1, ..., E_n)$ has to be replaced with something like $C\{L_1 = E_1, ..., L_n = E_n\}$;
- infixed functions: it seems that every constructor (or function) call of the form $E_1\ C\ E_2$ has to be replaced with something like $C\{L_1 = E_1, L_2 = E_2\}$;
- overloading (see Section 3 above).

Moreover, in standard LOTOS, processes may have several gate parameters and several value parameters. Requiring that all functions have a single parameter would create a dissymmetry between the data part and the behaviour part, unless the same requirement also applies to processes. Logically, this would entail the introduction of (nested) gate records; this is all the more true if gates are promoted to the status of "first-class citizen". Again, a clear migration path for existing LOTOS descriptions should be provided.

5.11 Absence of interoperability with other ISO languages

To be a successful and widely used language, E-LOTOS should interoperate smoothly with other main computer languages. We have to keep in mind the experience of the language Hermes developed from 1986 through 1992 at the IBM Research Center: Hermes designers identified the "lack of interoperability" as one of the fatal flaws of their language [KG95]:

> "Calling C routines from Hermes processes, and vice versa, is difficult. To call C, the programmer must write arcane 'C-Hermes' processes which need to be linked into the Hermes run-time system. Because of the many details involved, an experienced C programmer might need two to six hours to write and debug his first C-Hermes processes for a simple system call. Subsequent C-Hermes processes could be written more quickly, but the programmer would still require about one hour to write a simple call. Cumbersome interoperability with other languages was a major stumbling block to implementing production programs in Hermes."

By considering all computer languages currently standardized within ISO/IEC JTC1/SC22 — Ada, APL, C, C++, Cobol, Fortran, LISP, Modula-2, Pascal, and Prolog — we notice that all of them (perhaps with the exception of APL) support functions with multiple arguments.

Similarly, the languages standardized within ISO/IEC JTC1/SC21 such as ESTELLE, LOTOS, and IDL also allow functions with multiple arguments. This is also the case of the ITU-T language SDL.

The decision of having functions with a single argument would make of E-LOTOS *a language different from all other* ISO *languages.* E-LOTOS *will not gain acceptance by selecting unusual features instead of standard ones.*

Also, selecting the approach proposed in [Jef95] would put serious restrictions on the interoperability of E-LOTOS with other languages:

1. First of all, the record type of SML and [Jef95] is essentially "order-free": it is defined as a mapping from field labels to values. For instance, both record values $\{X = 1, Y = 2\}$ and $\{Y = 2, X = 1\}$ are equal (which implies that they have the same type).

   We are afraid that order-free records might not map easily to external types (such as C's structures) in which the order of fields is significant. This problem occurs if one wants, in an E-LOTOS description, to import, modify, and export a data structure specified externally (e.g., structures handled by UNIX system calls).

2. Also, it would not be easy to implement in C, C++, Ada, etc. an "external" function whose profile is specified in E-LOTOS. This would imply a run-time conversion from the E-LOTOS record of arguments to the list of arguments accepted by the external function, which is tricky and slow.

   As records are order-free, the order in which actual parameters are passed (through records) to an E-LOTOS function is not significant. Therefore, problems would occur when interfacing E-LOTOS to other languages, in which the order of function arguments is meaningful. There would be several possible translations from argument records to argument lists. The same problem occurs, for instance, if one wants to invoke, from an E-LOTOS program, a function belonging to a library of C functions (e.g., the mathematical library).

3. Also, the approach proposed in [Jef95] only allows parameters to be passed by value, whereas most computer languages allow more general types of parameter passing. For instance, IDL, C++, Ada, etc. allow "**in**", "**out**", and even "**inout**" parameters.

   The lack of more such parameter modes will prevent to invoke, from an E-LOTOS description, routines defined in libraries of C functions, such as most of the UNIX system calls.

   Moreover, it will not be possible to use E-LOTOS to specify the behaviour of systems whose interface is specified in IDL, thus limiting the usefulness of E-LOTOS for the description of ODP systems.

Adopting the anonymous tupling and single-argument function paradigms for E-LOTOS *would make of* E-LOTOS *an isolated language, unable to interoperate with computer languages like C, C++ and Ada used in real applications. This would also make the "plug-in" approach impossible and would prevent the interconnection with* IDL*.*

5.12 Inefficient implementations

As regards efficiency of code generated by compilers, it is clear that having functions with "**out**" parameters is better than functions returning records.

The latter solution requires that structures are put on the execution stack, thus implying an run-time overhead in terms of memory. There is probably also a time overhead, as the called function must assign the fields of these structures and, later, the calling function must extract these fields. "**out**" parameters can be handled in a more efficient way.

6 Conclusion

For the reasons stated above, we conclude that the first approach advocated in [JGL+95] is indeed better that the second one.

We therefore advise the E-LOTOS Committee not to include the following undesirable features:

- Polymorphism
- Type inference
- Anonymous tupling
- Single-argument constructor and functions

into the future E-LOTOS addendum.

Acknowledgements

We would like to thank Radu Mateescu for his comments regarding this paper.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[Bak82] T. P. Baker. A one-pass algorithm for overload resolution in Ada. *ACM Transactions on Programming Languages and Systems*, (4):601–614, April 1982.

[Jef95] Alan Jeffrey. Semantics for a fragment of LOTOS with functional data and abstract datatypes. ISO/IEC JTC1/SC21/WG7/1.21.20.2.3 Enhancements to LOTOS. Input document of the edition meeting, Liège (Belgium), December 1995.

[JGL$^+$95] Alan Jeffrey, Hubert Garavel, Guy Leduc, Charles Pecheur, and Mihaela Sighireanu. Towards a proposal for datatypes in E-LOTOS. Annex A of ISO/IEC JTC1/SC21/WG7 Second Working Draft on Enhancements to LOTOS. Output document of the edition meeting, Ottawa (Canada), July, 20–26, 1995, October 1995.

[KG95] Willard Korpfhage and Arthur P. Goldberg. Hermes Language Experiences. *Software — Practice and Experience*, 25(4):389–402, April 1995.

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

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

[WM94] R. Wilhelm and D. Maurer. *Les compilateurs, théorie, construction, génération.* Manuels informatiques Masson. Masson, Paris, 1994.