A Tutorial on ADT semantics for LOTOS users
Part II: Operations on Types

José A. Mañas

Dept. Ingeniería Telemática
E.T.S.I. Telecomunicación
Ciudad Universitaria
E-28040 MADRID
SPAIN

jmanas@dit.upm.es

14 November, 1988

Quis Custodiet Ipsos Custodes?
Decimus Junius Juvenalis

Abstract

In a LOTOS specification there are many types that are build from more basic ones. The operations allowed for types are presented with plenty of examples.

1 Introduction

This tutorial intends to cover the gap between normal specifiers and algebricians. The question to answer is

what are the abstract data types of LOTOS?

LOTOS syntax is used, but not introduced in detail. Readers are referred to the standard [ISO, 88]. This tutorial is dealing with semantics.

The author is not a mathematician, nor an algebrician, but a plain computer scientist, with a very pragmatic interest on abstract data types.

The presentation will be straightforward, introducing only those theoretical concepts that are strictly required. Many examples will be presented, not big ones, but meaningful ones.

The theory of abstract data types is not a single theory, but there are as many as ADT languages. We shall concentrate on ACT ONE, the language used in LOTOS. When there are several choices, they will be briefly presented, but only the one followed by ACT ONE is carried on.

In real specifications there will be many types. These types can be combined in different forms. Namely, renamed, composed, extended and actualized.

2 Renaming

In LOTOS there is syntactic support to define morphisms. A new type NT may be derived from and old one OT by means of a morphism

$$m : OT \rightarrow NT$$

that is written as

```
type NT is OT renamedby
  sortnames
    ns  for  os
    . . . .
  opnnames
    no  for  oo
    . . . .
  endtype
```

where m is defined as

$$\begin{array}{ll} m(os) = ns & \text{for sorts} \\ m(oo) = no & \text{for operations} \end{array}$$

The syntax forces to preserve the functionality of the operations: number of arguments and format (prefix, infix) is not modified; just the names are consistently changed all over the signature.

LOTOS puts a requirement on these morphisms: they must be **injective**. This requirement is not very theoretical, but practical. LOTOS designers think that it is not very sensible to write morphisms that confuse things. But it may be very restrictive in certain cases[1]. However, there is not much more fun with renaming.

3 Composition

It is much more challenging to compose types. LOTOS supports two forms of composition, namely **union** and **"enrichment"**[2]. Union appears as multiple import of types, as in

```
type ... is A, B, ..., Z with ...
```

where types $A, B, ...Z$ are composed, that is

$$A + B + ... + Z$$

Enrichment appears when sorts and/or operations are introduced, as in

```
type ... is ... with
    sorts ... opns ... eqns ...
    endtype
```

Let's start with some theoretical background.

Let

$$A =< S, F, E >$$

be an algebra with sorts S, operations F and semantics defined by (positive conditional) equations E. We extend this algebra by adding more of everything,

$$A + A' =< S + S', F + F', E + E' >$$

[1]For instance, users cannot specify the mapping of Mod4 into Mod2, (see Part I).

[2]We shall briefly define a propriety called enrichment, so let us write the LOTOS construction within quotes.

where + stands for disjunctive set union (i.e. standard *or*). The use of set union is interesting in actual practice. Suppose two types include booleans. In the composition of them there will be only a single boolean, as usually expected.

3.1 Consistency

If in (the initial model of) algebra A there exist two terms t_1 and t_2 that are not equal, but they become equal in (the initial model of) algebra $A + A'$, then we say the extension is **inconsistent**.

In positive terms, an extension is **consistent** iff $\forall t_1, t_2 \in |A|$

$$E + E' \vdash t_1 = t_2 \Rightarrow E \vdash t_1 = t_2$$

To be read: if in $A + A'$ we can prove two terms are equal, it must be provable too in the original algebra. Consistency means equivalence classes are not collapsed.

For instance, when you write down types involving booleans, you must be careful that things are either true or false, but never

$$\text{true} = \text{false}$$

3.1.1 Example

Let's consider natural numbers

```
type Naturals
  sorts  nat
  opns   0 :       -> nat
         s : nat -> nat
  endtype
```

that we extend to have stacks of natural numbers[3]

```
type Inconsistent.Stack is Naturals with
  sorts  stack
  opns   empty :            -> stack
```

[3] We do not claim it is a useful stack at all. But be aware that the strange equation [3] is too closed to what normal specifiers write when they want a finite stack: something where nothing else can be added.

```
                push  : nat, stack -> stack
                top   :      stack -> nat
          eqns
[1]         top (empty) = 0 ;
[2]         top (push (n, s)) = n ;
[3]         push (n, s) = s;
          endtype
```

It can be easily shown this type is an inconsistent extension of naturals:

[4]	top (push (s (0), empty)) = s (0)	[2]
[5]	top (empty) = s (0)	[4] ← [3]
[6]	0 = s (0)	[5] ← [1]

We can easily demonstrate that

$$\forall x : nat, s(x) = 0$$

ALL the naturals are equal to zero. We have just ONE equivalence class.

However, don't be distressed by this terrible example. There are many practical cases where it is possible to live with inconsistencies. A bit more later on.

3.2 [Sufficient] Completeness

If for some sort $s \in A$, there exist terms in $|A + A'|s$ for which there is no equal term in $|A|s$, then we say the extension is **incomplete**.

In positive terms, an extension is **complete**[4] iff

$$\forall s \in A, \forall t \in |A + A'|s, \exists t_1 \in |A|s, E + E' \vdash t = t_1$$

Completeness means that we are not inventing new equivalence classes.

For instance, when you write down types involving booleans, it is important that for every *t: bool* you can prove

$$t = \mathit{true} \text{ or } t = \mathit{false}$$

[4]Some authors call it sufficiently complete. We shall not make any difference.

3.2.1 Example

Let's now try another definition of stacks of natural numbers[5].

```
type Incomplete.Stack is Naturals with
  sorts  stack
  opns   empty :             -> stack
         push  : nat, stack -> stack
         top   :      stack -> nat
  eqns
    top (push (n, s)) = n;
  endtype
```

It can be easily shown this type is an incomplete extension of Naturals. Here you have a few new equivalence classes for sort nat

top (empty)
s (top (empty))
s (s (top (empty)))
...

3.3 Enrichment

We say an extension is an **enrichment** iff it is consistent and complete.

Enrichment means there are the same equivalence classes before and after the extension. Neither less, nor more.

3.3.1 Example

Lastly, let's define a good stack of nats.

```
type Good.Stack is Naturals with
  sorts  stack
  opns   empty :             -> stack
         push  : nat, stack -> stack
         top   :      stack -> nat
```

[5]We do not claim it is a useful stack at all. But be aware that lack of previous equation [1] is too close to what normal specifiers write when ignoring errors.

```
            eqns
[1]           top (empty) = 0 ;
[2]           top (push (n, s)) = n ;
            endtype
```

It can be shown this type is an enrichment of Naturals:

1. Consistency
   (a) it is enough to consider sort *nat*, just because there is no other sort in Naturals.
   (b) there is only one operation producing *nat*: *top*
   (c) we have to prove that
   $\nexists s : stack,\ top(s) = s_1$ and $top(s) = s_2$
   and Naturals $\vdash s_1 \notin s_2$
      i. for $s = empty$, $top(empty) = 0$ is unique.
      ii. for $s = push(n, s)$, $top(push(n, s)) = n$ is unique too.
2. Completeness
   (a) we have to prove that $\forall t = top(s) \in nat$, $\exists n : nat$ such that $top(s) = n$
      i. $top(empty) = 0 \in |Naturals|nat$
      ii. $top(push(n, s)) = n \in |Naturals|nat$

3.4 Do we need enrichments?

LOTOS does not restrict extensions. Specifiers may write inconsistent and incomplete types freely. It's their full responsibility[6]. Inconsistent specifications may save paper. For example, let us suppose we have a very nice theory of groups. We can easily go into abelian groups

```
type Abelian.Group is Group with
  eqns
    x . y = y . x
  endtype
```

which is clearly inconsistent, because $a.b \neq b.a$ in group theory.

[6]For readers surprised by such a permissiveness, we must recognize that, in general, these proprieties are indecidable. That's the true reason to allow them: who sorts them out?

Incompleteness is very often used in relation with partial algebras. Coming back to that example of type *Incomplete.Stack*, the peculiar equation for *top(empty)* reflects that it is an exception. We can comfortably live with incomplete types, so far as we can prove (somehow) that the undesirable terms will be never produced. The issue is not trivial, and here we just have the room to mention it. When ADT languages are more oriented to produce working products (lotos is mostly devoted to algebra) it is usual to introduce "application conditions" or predicates that are tested on produced terms, and mut be fulfilled in order to the term to be accepted. It is closely related to exception mechanisms that raise a signal (in Ada terms) when the operands fail to comply with some requirement, as is the case for operation *top* on operand *empty*.

To put another example, the definition of a partial ordering relation will yield *TRUE*, *FALSE* or *UNKNOWN* (for unrelated pairs). It's clearly an incomplete extension of Booleans.

To summarize, LOTOS allows for inconsistent and incomplete types. Such types may be harmless, or cause troubles. That's up to the specifier. Nevertheless, it would be great having a tool to check it. Such a tools, are currently under study, but their applicability is quite restricted.

4 Parameterized Types: Introduction

Reusability is a major goal of modern software engineering. In order to achieve this goal, it is necessary that software be broken into components that are as resuable as possible; parameterization is a technique that can greatly enhance the reusability of components. For example, *lists of X*, *sets of X*, etc. Generic modules also greatly ameliorate the otherwise odious need for defining abstractions whenever they are used. Without some such facility, strong typing would not be tolerable in practice.

Before giving details, we shall informally introduce the concepts we are going to deal with. A free use of Pascal will help in fixing the wishes.

5 Parameterization: Informal Presentation

5.1 Just Abstraction

It is typical of Pascal programmers to define LISP lists as

```
TYPE list= record  item: integer; next: ^list  end;
```

And then code some functions and procedures to handle these lists, namely header, tail, concatenation, etc.

LOTOS uses normal ADTs for this business:

```
type Lists is Naturals with
  sorts  list
  opns   nil    :             -> list
         _ .. _ : nat,  list -> list    (* cons   *)
         hd     :       list -> nat     (* car    *)
         tl     :       list -> list    (* cdr    *)
         _ ++ _ : list, list -> list    (* append *)
  eqns
    hd (nil) = 0 ;                      (* exception *)
    tl (nil) = nil ;                    (* exception *)
    hd (n .. L) = n ;
    tl (n .. L) = L ;
    nil ++ L = L ;
    L ++ nil = L ;
    (n .. L1) ++ L2 = n .. (L1 ++ L2) ;
  endtype
```

Now, the Pascal programmer wants a sorting function:

```
function sort (L: list): list;
  begin
    (*  some wonderful algorithm  *)
  end
```

The LOTOS specifier just extends the previous type:

```
        type Sorts.Nats is Lists with
          opns
            sort   :       list -> list
            sorted :       list -> bool
            in     : X,    list -> bool
            hase   : list, list -> bool
              (* HAs Same Elements *)
          eqns
[1]         hase (L, L) = true ;
[2]         hase (L1 ++ (x .. L2),
```

```
                 M1 ++ (x .. M2)) = hase (L1 ++ L2, M1 ++ M2) ;
[3]           x in L1 = true,
              x in L2 = false =>
                  hase (L1, L2) = false ;
[4]           hase (L1, L2) = hase (L2, L1) ;

[5]           x in nil = false ;
[6]           x in (y .. L) = (x eq y) or (x in L) ;

[7]           sorted (nil) = true ;
[8]           sorted (x .. nil) = true ;
[9]           sorted (x .. (y .. L)) = (x <= y) and sorted (y .. L) ;

[10]          hase (L1, L2) = true,
              sorted (L2)   = true =>
                  sort (L1) = L2 ;

            endtype
```

While the Pascal programmer needed a parameterized function, the LOTOS specifier uses just an extended ADT.

Sorting may be specified in many ways. While the Pascal programmer has to go directly for a sorting algorithm,the LOTOS specifier doen't need any: he just states the properties of sorting, namely getting something sorted. The previous presentation is particularly independent of any sorting algorithm. The key ideas come from prof.dr. Manfred Broy, at Passau.

5.2 Parameterization

Now, our Pascal programmer wants something better, a list of anything, something he tries to write as

```
    TYPE list (X) = record item: X; next: ^list (X) end;
```

but the Pascal compiler shouts.

No problem for the LOTOS specifier:

```
        type Lists.of.X is
          formalsorts  X
```

```
formalopns   E :  -> X        (* for error *)

sorts  list
opns   nil    :             -> list
       _ .. _ : X,    list -> list    (* cons   *)
       hd     :       list -> X       (* car    *)
       tl     :       list -> list    (* cdr    *)
       _ ++ _ : list, list -> list    (* append *)
eqns
  hd (nil) = E ;                      (* the error *)
  tl (nil) = nil ;
  hd (n .. L) = n ;
  tl (n .. L) = n ;
  nil ++ L = L ;
  (n .. L1) ++ L2 = n .. (L1 ++ L2) ;
endtype
```

Now, for lists of naturals,

```
type Lists.of.Nats is Lists.of.X
  actualizedby Naturals using

  sortnames   nat for X
  opnnames      0 for E
  endtype
```

The poor Pascal programmer is still alive[7] to try

```
function sort (L: list (X),
               <=: function (X, X): boolean): list (X) ;
```

where he is trying to pass both a list of anything and a boolean function to compare those anythings. Of course, the Pascal compiler shouts more and more ...

In the meanwhile, our LOTOS specifier goes on without trouble ...[8]

```
type Sorts.Lists.of.X is Lists.of.X with
  formalsorts
    bool
```

[7] Incredible! Why didn't he yet learn LOTOS?
[8] He sings in the rain, you know.

```
formalopns
  _ <= _ : X,    X    -> bool
  _ eq _ : X,    X    -> bool
  true,
  false  :            -> bool
  _ and _,
  _ or _ : bool, bool -> bool
opns
      (* same as above in type Sorts.Nats *)
eqns
      (* same as above in type Sorts.Nats *)
endtype
```

If you want to sort natural numbers, just instantiate it ...

```
type Sorts.Lists.of.Nats is Sorts.Lists.of.X
  actualizedby  Naturals using

  sortnames  nat for X
  opnnames     0 for E
  endtype
```

Again, notice that those sorts and operations with the same names do not need to be explicitly stated. E.g. *bool for bool.* (We suppose Naturals include Booleans, as usual).

5.3 Interface Requirements

Readers may be shivering by the power of these constructions. While in poor Pascal strong type checking is good enough, here we are going far further. Is strong sort checking enough? Shouldn't we ask for more security?

Yes, we can go further. Notice you can build lists of anything, but you can only sort lists of things on which a total ordering is defined. LOTOS permits to impose conditions on actual parameters. For our case, we would like the following **formal semantics**[9]

```
(*  eq must be an equality relation  *)
[1]   a eq a = true ;      (* reflexive *)
[2]   a eq b = b eq a ;    (* symmetric *)
[3]   a eq b = true ,
```

[9]Semantics for the formal algebra, that is, restrictions to put on any potential actual algebra.

```
      b eq c = true =>
        a eq c = true ;    (* transitive *)

(*  <= must be a total ordering  *)
[4]   a <= a = true ;      (* reflexive *)
[5]   a <= b = true ,
      b <= a = true =>
        b eq a = true ;    (* antisymmetric *)
[6]   a <= b = true ,
      b <= c = true =>
        a <= c = true ;    (* transitive *)
[7]   (a <= b) or (b <= a) = true ;  (* total *)

(*  true and false must be different  *)
[8]   true <> false ;

(*  and must be as expected  *)
[9]   true and a = a ;
[10]  false and a = false ;
[11]  a and b = b and a ;

(*  or must be as expected  *)
[12]  true or a = true ;
[13]  false or a = a ;
[14]  a or b = b or a ;
```

LOTOS permits to state all these conditions under the section `formaleqns`. But there is an exception: only (positive conditional) equalities are allowed. Thus, [8] is not accepted.

In general ADT theory, the formal equations are needed only to *certify* an actual candidate is acceptable. Once accepted, they may be dropped. But in LOTOS, these formal equations are retained as part of the result type. Then the discussion above about existence of an initial model for algebras specified using unequalities comes back. LOTOS just request (positive conditional) equalities.

As perspicacious readers did already notice the difficulty arises when one has to prove that the conditions hold for some particular module. Of course, the simpler the conditions the better the chance of actually getting the proof. But be ready for hard work!

Automatic theorem provers are not very good in actual practice. But, it is usually the case that the formal equations are **syntactically equal** to the equations of the actual parameter. In these cases, the demonstration is trivial. For instance, the formal equations imposed on operations *and* and *or.* are very likely to be present in the actual type Booleans. That greatly simplifies life.

5.4 General Instantiation

We have already done some instantiations or actualizations of parameterized ADTs. They were rather conventional in the sense that the result is an algebra[10]. But let's go a little bit further and allow actual parameters to be parameterized ADTs as well. For instance, *lists of lists ...*

```
type Lists.of.Lists.of.X is Lists.of.X
  actualizedby Lists.of.X using

  sortnames  list for X
  opnnames    nil for E
  endtype
```

We get another parameterized ADT that can be instantiated. For instance, *lists of lists of natural numbers ...*

```
type Lists.of.Lists.of.Nats is Lists.of.Lists.of.X
  actualizedby Naturals using

  sortnames  nat for X
  opnnames     0 for E
  endtype
```

When several actualization steps are involved, the order of actualization is not relevant. That is, we can as well specify

```
type Lists.of.Lists.of.Nats is Lists.of.X
  actualizedby Lists.of.Nats using

  sortnames  list for X
  opnnames    nil for E
  endtype
```

5.5 Higher Order Programming

This last example will not introduce new concepts, but let's put it forward to show the power of LOTOS ADTs.

[10]Better say a category of algebras out of which we choose the initial model.

LISP and functional languages in general claim that they provide higher order functions. In simple terms, they treat functions as first class citizens that can be passed as arguments and returned as results. LOTOS is not a higher order language and arguments to operations cannot be operations. But with parameterized specifications, something can be done.

The next example shows a classical case of higher order programming. In LISP style it is expressed as

```
mkiter= function (star, one)
        return iter
        where
          iter= function (L)
                if (L = NIL)
                  then
                    return one
                  else
                    return star (car (L), iter (cdr (L)))
```

But we have something more than in LISP: we can impose conditions on the formals ...

```
        type On.Lists is Lists.of.X with
          formalopns
            1     :      -> X
            _ * _ : X, X -> X
            _ + _ : X, X -> X

          formaleqns
            a * 1 = a ;         (* right neutral *)

          opns
            iter  :       list -> X
            _ & _ : list, list -> list

          eqns
            iter (nil) = 1 ;
            iter (x .. L) = x * iter (L) ;

            L & nil = L ;
            nil & L = nil ;
            (x1 .. L1) & (x2 .. L2) = (x1 + x2) .. (L1 & L2) ;
```

This type may be instantiated with Naturals using

```
+  for *
0  for 1
f  for +
```

Then *iter* adds the elements of a list. And you can count the elements as

```
iter (L & (1 .. 1 .. 1 .. ... ))
```

Or you can use

```
+ for *
0 for 1
* for +
```

Then & corresponds to the internal product.

It is left as an exercise to the reader instantiating it to evaluate the Pythagoras' distance.

6 Parameterization: Formal Presentation

In general, a LOTOS ADT will include some formal sorts FS, some formal operations FF, some formal equations FE, some sorts S, some operations F and some equations E.

Let's introduce some notation,

$PSPEC$	is the parameterized ADT	
FA	is the formal parameter	$=< FSIG, FE >$
AA	is the actual parameter	$=< ASIG, AE >$
RA	is the result of instantiation	$=< RSIG, RE >$

6.1 Parameterized Specifications

A parameterized specification is defined as a pair of algebras

$$PSPEC =< FA, A >$$

where there is a *formal* and a *target* algebras

$$FA =< FS, FF, FE >$$

$$A =< FS + S, FF + F, FE + E >$$

FA states properties that actual parameters must have. A extends the result of importing an actual with more sorts, operations and equations.

6.2 Basic Actualization

Actualization is a total *morphism*

$$g : FSIG \rightarrow ASIG$$

Total *morphism* means that every formal sort and every formal operation must be instantiated. LOTOS does not support the so called partial actualizations where some formals are instantiated while others remain formal.

If the actual is a simple algebra AA, the result is defined as

$$RA =< AS + g(FS) + S, AF + g(FF) + F, AE + g(FE) + g(E) >$$

Notice we write $g(E)$ because formal operations may appear in the target part, and the actualization morphism have to be applied to them.

That expression may be simplified to

$$RA =< AS + S, AF + F, AE + g(FE) + g(E) >$$

because $g(FS)$ must be a subset of AS, and $g(FF)$, a subset of AF.

$$g(FSIG) \subset ASIG$$

With respect to equations, they could be dropped, but LOTOS holds them. (See requirement 2 below).

6.3 General Actualization

In general, the actual parameter may be another parameterized ADT

$$APSPEC = << AFS, AFF, AFE >< AFS + AS, AFF + AF, AFE + AE >>$$

then the instantiation is a nice game of unions

$$\begin{aligned} PSPEC(APSPEC) = \ & << AFS, AFF, AFE > \\ & < AFS + AS + g(FS) + S, \\ & AFF + AF + g(FF) + F, \\ & AFE + AE + g(FE) + g(E) >> \end{aligned}$$

With respect to requirements, see requirement 2 below.

6.4 Nonparameterized types

The types studied up to this section were not parameterized. They strictly correspond to parameterized specifications with an empty formal part

$$<<>, < S, F, E >>$$

They play a similar role to nullary operations: somehow they are *constants*.

6.5 Operations on parameterized ADTs

Parameterized ADTs may be renamed and composed. When renamed, the morphism applies both to the formal and target parts.

When composed, they are *or'ed* componentwise. I.e.

$$\begin{aligned} PSPEC1 + PSPEC2 = \ & << FS1 + FS2, FF1 + FF2, FE1 + FE2 >, \\ & < FS1 + FS2 + S1 + S2, \\ & FF1 + FF2 + F1 + F2, \\ & FE1 + FE2 + E1 + E2 >> \end{aligned}$$

6.6 Requirement 1

The formal part must be a self-contained ADT. That means that sorts used in FF and FE must belong to FS, and operations used in FE must belong to FF.

Referring back to the example of type *Sorts.Lists.of.X*, notice *bool* appears as a formal sort that will be later actualized by actual booleans.

6.7 Requirement 2

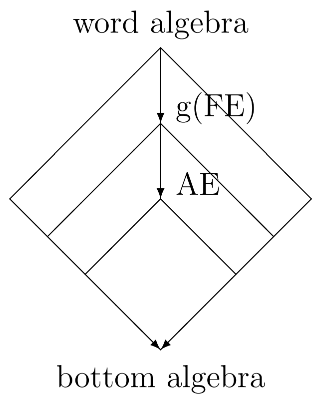
The formal equations must hold for the actual parameter. That is, FE must be a consequence of AE

$$AE \vdash g(FE)$$

in terms of categories,

$$Alg(ASIG, AE) \subseteq Alg(ASIG, g(FE))$$

in words, the actual parameter must be a subcategory of the formal algebra after mapping it to the actual signature.



When the actual is a parameterized ADT, then the formal equations must be guaranteed

$$AFE + AE \vdash g(FE)$$

6.8 Persistency

Let's consider every sort s in AS. When the target is added, we are adding new terms to that sort. After considering the whole set of equations $AE + g(FE) + E$, the number of equivalence classes in the sort s in the result algebra may be

- bigger than in AA

  that's similar to extending an ADT incompletely; we say AA does not **persists** through actualization;

- smaller than in AA

  that's similar to extending an ADT inconsistently; we say AA does not **persists** through actualization;

- equal

  that's similar to enriching an ADT; we say AA **persists** through actualization;

A PSPEC is **persistent** if for every actual algebra AA that have the properties required by FE, and for every sort in AA, the equivalence classes of s in AA are the same as in RA.

Usually, *pspecs* are not persistent for every valid actual algebra, but only for some of them. Persistency is similar to consistency and completeness between the actual parameter and the resultant algebra. The intuition is rather natural and not too complicated, but the mathematical treatment is too large to be covered in this tutorial.

6.8.1 Examples

Type Lists.of.X is persistent over any actual because

1. There is no formal condition FE, thus any actual type may be used for instantiation.
2. The equations do not make equal any term in X[11].
3. No new equivalence class is introduced in sort X.

Persistency in actualizations is very similar to consistency in extensions. For instance,

[11] It is not true by definition; you have to prove it, but the proof is very easy.

```
        type Stack.of.X is
          formalsorts  X
          formalopns   E  : -> X
          sorts  stack
          opns   empty :            -> stack
                 push  : nat, stack -> stack
                 top   :      stack -> nat
          eqns
[1]         top (empty) = E ;
[2]         top (push (x, s)) = x ;
[3]         push (x, s) = s;
          endtype
```

is not persistent for any actual, because it makes every term of sort X equal to E.

6.9 Do we need persistent types?

As with enrichments, persistency is a nice propriety, but very often it is too restrictive. So, it's reasonable that LOTOS does not require parameterized ADTs to be persistent. Nevertheless, it would be great having a tool to check it. Such a tools are currently under study, but its applicability seems to be quite restricted.

However, persistency is very important when libraries of ADTs are considered. Parameterized ADTs in libraries should be carefully documented, clearly specifying the interface requirements, the class of ADTs for which they are persistent, and what happens with other ADTs for which they are not persistent, but may still be useful.

Nobody would accept type *Stack.of.X* (as presented in the last section) in the standard library. But specifiers may accept a version that introduces new equivalence classes (something similar to the type *Incomplete.Stack* presented before). Of course, specifiers must be warned.

To summarize. Persistency is a nice property, LOTOS does not enforce it, it is not easily checkable, you can live without it, but it must be clearly documented, mainly for types in the standard library.

6.10 Composability

Persistency of PSPECS guarantees composability of actualizations. An example was already shown building

```
Lists.of.Lists.of.Nats = Lists.of.Lists.of.X (Naturals)
```

```
= (Lists.of.X (Lists.of.X)) (Naturals)

= Lists.of.X (Lists.of.Nats)
= Lists.of.X (Lists.of.X (Naturals))
```

and both routes yield the same result.

7 The Case for LOTOS

Most of the use of parameterized specifications in LOTOS has been already presented. Let's now introduce the last details with respect to scopes, in order to unsderstand the use of signatures and algebras in actual specifications.

7.1 Requirement

Surprisingly, LOTOS requires actualization morphisms to be injective for operations. So, you cannot actualize type *On.Lists* using $+$ both for $*$ and $+$. I wonder why.

7.2 Scope signatures

LOTOS specifications are organized in scopes that can be nested as usual in block structured languages as Pascal. There is a global scope that encompasses from the specification header down to the end of the specification. Gloal data types have this scope:

```
specification ...
    global data type definitions
  behaviour
    . . . .
  endspec
```

There are nested scopes called definition blocks. They are introduced after the behaviour key word (as in the previous pattern), and as the body of process definitions:

```
behaviour expression
where
  local data type definitions
```

When we refer to a sort identifier to qualify a formal parameter (of specification or process), or in a behaviour expression, we are referring to some signature that gives the syntax of the terms of that sort, and to some algebra that gives the semantics of values of that sort.

But, when there are several types with the same scope, ones being extensions of others,

which signature are we referring to?

Let's put an example ...

```
process P ...
    ... g ?x: SA ; BE

  where
    type A is
      sorts  SA
      opns    a : -> SA
      endtype

    type B is A with
      opns    b : -> SA
      endtype
  endproc
```

where type B is an incomplete extension of type A.

which are the valid values for $x : SA$?

the answer is: you build a signature per scope that is the union of all the types defined in that scope[12]. That signature is used to produce ground terms for x. In the previous example,

```
type this.scope is
  sorts  SA
  opns    a : -> SA
          b : -> SA
  endtype
```

[12]Standard rules for visibility and hiding of identifiers in nested scopes, as in Pascal, are used.

and

```
g ?x: SA ; BE
```

is equivalent to

```
  g !a ; BE [a/x]
[]
  g !b ; BE [b/x]
```

The procedure is systematic, but usually there are infinite ground terms, and you cannot expand ?-experiment offers in a finite piece of paper. But that's only a minor trouble. However, a bigger trouble rises when considering the following section. In-line expansion of ground terms is not always correct.

7.3 Canonical Data Type

Previous section presented the case for signatures, but we are also interested on the algebra. *Which algebra are we referring to when there are several types involved?* Surprisingly, we are not referring to the algebra resulting of the union of local algebras. Let's put another example,

```
hide g in
    g ?x: SA ; ...
  |[ g ]|
    P [g]

where
  type A is
    sorts  SA
    opns     a : -> SA
    endtype

  process P [g] ... :=
      g !b ; ...

    where
      type B is A with
        opns    b : -> SA
        endtype
    endproc
```

will actions on g synchronize?

According to the scope signature procedure, they would not, because term b is ignored outside process P, and thus `g ?x: Sa` would only expand into `g !a`. But that's an incomplete view of the story. In fact they synchronize because LOTOS uses ONLY ONE algebra for the WHOLE specification. This single algebra is called **canonical** and results of the union of ALL the types spread along the specification. Of course, only non parameterized types are considered. Parameterized ADTs were already used to build nonparameterized types.

7.4 Last comment

The procedures to build scope signatures and the canonical algebra imply that it is **EXTREMELY DANGEROUS** to extend types incompletely or inconsistently, or deal with nonpersistent parameterized ADTs.

For instance, imagine you extend some type including Booleans and, inadvertently, you make

$$\mathit{true} = \mathit{false}$$

You have destroyed Booleans everywhere!

But, as already mentioned, those *bad* types are useful. There is a systematic trick to deal with this business. If you want a new type NT that is based on an old one OT, and NT modifies the partition of OT sorts into equivalence classes, and you want OT to be preserved as it is, then do the extension (actualization) in two steps:

```
first rename OT:

        type ROT is OT renamedby
          sortnames
            . . . .
          opnnames
            . . . .
          endtype

then operate on ROT:

        type NT is ROT with
            ... your nasty business goes here
          endtype
```

When LOTOS joins $OT + ROT + NT$, NT destroys ROT, but OT is not disturbed.

8 Further Readings

The *bible* is [Ehrig, 85], but it is unreadable for beginners and covers too many topics and issues currently under research. Language ACT ONE, the ancestor of LOTOS for ADTs, is presented in chapter 9, with examples.

For LOTOS, the standard reference is [ISO, 88], both for syntax and precise semantics. This tutorial has not been too precise with syntax. If you really intend to write data types in LOTOS, have a look at the concrete syntax. For the semantics, I think that the only way through is letting the semantics analyser complain till it gets through. Then, use a simulator till you are confident with your types.

[Broy, 81] introduces the proprieties of composition from the point of view of category theory.

Parameterization is subject to very intensive research. A good introduction with reasonable examples can be found in [Goguen, 85a]. There is some philosophy in [Goguen, 85b]. It is interesting to read [Goguen, 79] where a working language is presented with many examples.

[Wagner, 81] is more formal, but still readable. This last paper was presented to a congress in [Tatcher, 82] in a more formal style. If the reader wants brain food, he may try [Ehrig, 81]. At least there are several interesting examples.

9 Acknowledgements

Presentations of ADTs use to be too trivial or too complicated for laymen. Understanding the theory and putting it in simple words with examples has been a time consuming task. Many people have helped in discussing issues, examples, papers and books.

I have to acknowledge the early help of Huub van Thienen in the Twente University of Technology[13], as well as the other members of the SEDOS Project, mainly those at the Technical University of Berlin.

Apart from SEDOS, I am very thankful to the Research Center of Standard Electrica at Madrid, for inviting to the highly illuminating lectures of prof.dr. Manfred Broy from Passau.

And last, but not least, I have to remind the members of the Department of Ingenierìa

[13]Current affiliation: Dpt. of Informatics, Faculty of Science, University of Nijmegen, The Netherlands.

Telemática, and the students attending to the courses on LOTOS at the ETSITM, who have stoically attended the lectures and provided fruitful feedback.

References

[Broy, 81] M. Broy, C. Pair and M. Wirsing. A Systematic Study of Models of Abstract Data Types. 1981.

[Ehrig, 81] H. Ehrig, H.J. Kreowski, J.W. Tatcher, E.G. Wagner and J.B. Wright. Parameter Passing in Algebraic Specification Languages. Workshop on Program Specification, Aarhus, 1981. LNCS vol. 134, pp. 322-369.

[Ehrig, 85] H. Ehrig and B. Mahr. Fundamentals of Algebraic Specification, Part 1. Springer Verlag, Berlin, 1985.

[Goguen, 85a] J. Goguen and J. Meseguer. EQLOG: Equality, Types, and Generic Modules for Logic Programming. Int. Summer School on Advanced Prog. Technologies, M.F. Verdejo (ed.). Fac. Informatica de San Sebastian, 1985. pp. 1-69.

[Goguen, 85b] J. Goguen. Parameterized Programming. Int. Summer School on Advanced Prog. Technologies, M.F. Verdejo (ed.). Fac. Informatica de San Sebastian, 1985. pp. 70-121.

[Goguen, 79] J.A. Goguen and J.J. Tardo. An Introduction to OBJ. IEEE Conf. Spec. for Reliable Software, M.I.T. April, 1979. pp. 170-189.

[ISO, 88] ISO - Information Processing Systems - Open Systems Interconnection - LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. IS 8807, 1988.

[Tatcher, 82] J. Tatcher, E. Wagner and J.B. Wright. Data Type Specification: Parameterization and The Power of Specification Techniques. Trans. Prog. Lang. and Systems, 4(4), 1982. pp. 711-732.

[Wagner, 81] E.G. Wagner. Lecture Notes on the Algebraic Specification of Data Types. IBM, Thomas J. Watson Research Center, Yorktown Heights, New York. RC 9203, Oct. 1981.