A Tutorial on ADT semantics for LOTOS users
Part I: Fundamental Concepts

José A. Mañas

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

jmanas@dit.upm.es

14 November, 1988

The limits of my language mean the limits of my world
Ludwig Wittgenstein

Abstract

An informal presentation of the semantics of Abstract Data Types as they are used in the specification language LOTOS. This paper covers the fundamentals, while a separate *Part II* covers the available operations.

1 Introduction

This tutorial intends to cover the usually big 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 definition [ISO, 88]. This tutorial is mostly 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. That helps making this paper introductory, but be sure it will not be very useful for mathematicians.

The magic word in this world is *ABSTRACT*. There are several reasons to say **abstract**. Some people say ADTs are implementation independent, and put as a counter example concrete data types as those in the language Ada. It is true that you can describe data without giving algorithms, thus providing an implementation-free specification, but it is also true that you can specify algorithms as ADTs too. In fact, most people specify ADTs by giving and evaluation algorithm in the style that is usualy referred to as *a rewrite system*.

Other people say ADTs are representation independent, as opposed to cluttering representation issues as in old FORTRAN. Ok, that's true as for any language where types may be defined by means of symbolic names, that is, for most modern programming languages. But be aware you can also describe bytes and bits as ADTs.

We prefer to say ADTs are a *methodology* to describe objects by itemizing the properties your data must have. It's up to specifiers which properties are described. You may wish to specify algorithms or even bit representations. Those are perfectly legal items to be specified. You may wish to specify the axioms of group theory, and that is perfectly legal too.

With respect to the contents of this paper, 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.

A note of warning about the universality of what is exposed in the following pages. 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. When there are differences between ACT ONE and LOTOS, the last one is preferred.

2 ADTs in LOTOS

In order to make clear what do LOTOS specifiers need from ADTs, let's start by revising LOTOS constructions.

2.1 Synchronization

Basic LOTOS activity is to synchronize on action denotations. There are three basic kinds of synchronization:

1. synchronization:

   `g !E1  |[ g ]|  g !E2`

   where expressions $E1$ and $E2$ must belong to the same sort.[1] It will succeed iff we can prove $E1 = E2$ from the specification of the common type.

2. value passing:

   `g !E  |[ g ]|  g ?x:S`

   where expression E must belong to the sort S. It will succeed, becoming $x = E$.

3. negotiation:

   `g ?x:S  |[ g ]|  g ?y:S`

   It will succeed, becoming $x = y$ for some value in sort S.

Cases 2 and 3 can be regarded as trivial with respect to the use of data type specifications. Case 1 introduces the notion of **proving**. It must be proved if two expressions are equal or not, in order to know whether the synchronization may succeed.

2.2 Guarded Behaviours

The behaviour

$$[E1 = E2] \rightarrow B$$

is equivalent to *stop* or B, depending on $E1$ being equal to $E2$ or not. Again, it must be proved. There is a shorthand for boolean expressions: $[E1]$ stands for $[E1 = true]$.

2.2.1 Predicates

The action denotation

$$a[E1 = E2]; B$$

will allow a to synchronize or not depending on the result of proving if $E1$ is equal to $E2$ or not. There is a shorthand for boolean expressions: $[E1]$ stands for $[E1 = true]$.

[1] Precise definitions are given below. For the time being, think of sorts as Pascal types.

2.3 Conclusion

The use of an ADT specification in LOTOS is basically finding out if two expressions are equal or not. Of course, valid expressions are only those constructed using the operations of specified types, with the standard rules. This is typical of every programming language, and we shall not say much more.

3 Very Basic Remind

Since readers are not expected to be working algebricians, we shall copy here some very very basic notions. To simplify the notation, where no confusion arise, we shall use the Pascal typing mechanism, writing $x : S$ for $x \in S$.

3.1 Functions

A function is a deterministic criterion to relate *EVERY* combination of elements of a domain of sets, into just one element of a range set. The standard notation is used:

$$f : s_1 \dots s_n \rightarrow s_r$$

where the domain is $(s_1 \dots s_n)$, and the range is s_r. Notice we say "... every ...", that is, we are dealing with **total** functions, those defined over the whole domain.

3.2 Surjective

A function is surjective if and only if for every element of s_r there exists a combination of elements in the domain whose image is that element in the range. The whole range set is covered. Formally,

$$\forall x_r : s_r, \exists x_1 : s_1 \dots x_n : s_n \text{ such that } f(x_1 \dots x_n) = x_r$$

3.3 Injective

A function is injective iff different combinations of elements in the domain are mapped into different elements of the range. Informally, it means that from the result we can infer the arguments.

$$f(a) = f(b) \Rightarrow a = b$$

For injective and total functions f there exists an inverse function f' such that

$$f \circ f' = I$$

$$f' \circ f = I$$

where $\circ$ is the composition of functions

$$(f1 \circ f2)(x) = f1(f2(x))$$

and I is the identity function

$$I(x) = x$$

3.4 Bijective

A total function that is both surjective and injective is then bijective.

4 Fundamental Concepts

Although we shall try to keep to a minimum, we need some formalism to proceed. ACT ONE is a language for specifying by means of algebras.

- *ALGEBRA:*

  An **algebra** is a set of elements with operations and properties between them.

This is very informal. Let's refine the concept.

4.1 Sort

- *SORT:*

  A **sort** is a set of elements with a name.

There are algebras with just one sort[2]. For instance, that of *Bool*. And there are algebras with more than one sort[3]. For instance, the algebra of the *stacks of integers*, where at least we have the sort of *integers* and the sort of *stacks*.

What we here call sorts, corresponds to the concept of **type** in most programming languages (e.g. in Pascal). But readers must be careful because later on we shall call types to algebras. But let's go step by step.

4.2 Operation

- *OPERATION:*

  An **operation** is a function with a domain and a range

$$op : s_1, s_2, \ldots s_n \rightarrow s_r$$

where $< s_1 \ldots s_n >$ is the domain[4], over which the operation is defined; and s_r is the range, the sort of the result. We shall restrict ourselves to *total functions*. That is, operations must be defined for *EVERY* possible combination of input arguments. They must be defined over the whole domain.

Formally, let $op : Sx \rightarrow Sy$

$$\forall x \in Sx, \exists y \in Sy \text{ such that } op(x) = y$$

There are ADT languages able to specify partial operations, but that is not the case for ACT ONE in LOTOS. There are languages with nondeterministic operations, but their algebraic support is extremely complicated. It is not the case for LOTOS too.

4.3 Ground Terms

- *GROUND TERMS:*

  The elements in the sorts are called **ground terms**[5]

[2] one-sorted algebras, homogeneous.
[3] many-sorted algebras, heterogeneous.
[4] Cartesian product of sorts.
[5] Some authors call them *carriers*, but the terminology is not common to everybody.

In an algebra A, including a sort s, we shall denote by $|A|s$ the set of ground terms in sort s. Sometimes we shall abuse of the notation and write $|A|$ to denote all the sets of ground terms of sorts belonging to $|A|$.

There is a simple procedure to construct all the terms in a sort:

1. seed: nullary operations

   $\forall op :\rightarrow s, op \in |A|s$

2. rest: operations on ground terms generate new ground terms

   Let $x_1 \in |A|s_1, \ldots, x_n \in |A|s_n$

   and $op : S_1 \ldots S_n \rightarrow S$

   then $op(x_1 \ldots x_n) \in |A|s$

Example. Let's consider the Booleans, with one sort *bool* and three operations

$$\begin{array}{lll} T & : & \rightarrow bool \\ F & : & \rightarrow bool \\ not & : bool & \rightarrow bool \end{array}$$

the following are ground terms

T, F, $not(T)$, $not(F)$, $not(not(T))$, $not(not(F))$, ...

Notice the set of ground terms $|Booleans|_{bool}$ is infinite. This is a very typical situation.

We shall be mainly concerned with ground terms, but the definition may be slightly extended to cover **variables**. Let X be a set of variables of sort s. We call **terms** of the sort s to the set formed by ground terms, as previously defined, and the elements of X. Notice that previous rule (2.) generates *more* new terms than before.

For Pascal educated readers, nullary operations are CONSTANTS, and ground terms are valid expressions formed according to the Pascal type compatibility. Terms are similar, including variables of the appropriate sorts.

4.4 Signature

- *SIGNATURE:*

  We shall call **signature** of an algebra to the pair formed by the set of sorts and the set of operations.

$$\begin{array}{lcl} SIG & = & < S, F > \\ S & = & \{sorts\} \\ F & = & \{operations\} = \{op : S^* \rightarrow S\} \end{array}$$

For those who know Modula-2, a signature is quite similar to a DEFINITION module. It says how functions are called, and which are their parameters and result. The whole syntax is known, but nobody knows what those functions do when called. There is no *semantics* in a signature, but its value should not be disminished. All the elements of the game are in the signature. The rules of the game come later, but there is no further opportunity to introduce more elements. Even more, the names of the elements of the game are extremely interesting for the reader: the intuition works fine. To put an example, while every body will have a quick idea of an operation named *plus* may do, the intuition will not help at all if the operation is named *opn*27, and it will work against normal understanding if named *minus*. Intuition works despite any correct semantics that are provided later on. While machines are very good with statements, despite concrete names; humans are much better with names.

4.5 Morphisms

- *MORPHISM:*

  We shall relate signatures by means of morphisms. **Morphisms** are mappings between signatures.

Given SIG_1 and SIG_2, we may define a mapping

$$g : SIG_1 \rightarrow SIG_2$$

that maps every sort in S_1 into some sort of S_2, and every operation in F_1, into some operation of F_2. We shall put a minimal requirement on morphisms: they must preserve functionality, that is, the mapping of operations must be such that

$$\begin{array}{lll} if & op: & s_1 \ldots s_n \rightarrow s_r \\ then & g(op): & g(s_1) \ldots g(s_n) \rightarrow g(s_r) \end{array}$$

4.6 Algebra

The **signature** gives all the syntactic knowledge. We just need to add **semantics** to have an algebra.

- *ALGEBRA:*

  $algebra =< signature, semantics >=< S, F, semantics >$

An *ABSTRACT DATA TYPE* is an algebra.

The semantics is needed for proving if two expressions (ground terms) are equal or not. That is the reason for many languages to describe the semantics in terms of $=$ and $\neq$.

We shall use the following notation to denote reasoning with algebras: $A \vdash t_1 = t_2$ that is read as: from (the semantics of) algebra A we can prove that terms t_1 and t_2[6] are equal.

Semantics breaks sorts into equivalence classes

$\forall x \in CE_i$ and $\forall y \in CE_j$

$$\begin{array}{rcl} i = j & \Rightarrow & x = y \\ i \neq j & \Rightarrow & x \neq y \end{array}$$

If two (ground) terms belong to the same equivalence class, then they are equal. If two (ground) terms belong to different equivalence classes, then they are unequal. There is no exception. Each equivalence class groups equal terms, by definition.

Let us present a simple example. We shall start using LOTOS notation, but readers are referred to the LOTOS definition for a precise and concise syntax definition. It is not our concern.

4.6.1 Naturals modulo 2

```
type Mod2 is
  sorts
    bit
  opns
    0, 1  :          -> bit
    _ + _ : bit, bit -> bit

  semantics
    + | 0 1
   ---+-----
    0 | 0 1
    1 | 1 0
  endtype
```

[6] Obviously, (ground) terms, i.e. generated from A's signature.

There is just one sort, *bit*, and three operations, 0, 1 and +. The ground terms are 0, 1, $0+0$, $0+1$, $1+0$, $1+1$, $(0+0)+0$, $(0+0)+1$, $(0+1)+0$, ... and there are just two equivalence classes, as may be easily proved

class 0	class 1
0	1
$0+0$	$0+1$
$1+1$	$1+0$
$(0+0)+0$	$(0+0)+1$
$(0+1)+1$	$(0+1)+0$
...	...

4.7 Models

There may be many algebras with the same signature. That implies that given a signature, we may think of many algebras. We shall indistincticly speak of **models** as a synonym for algebras. We shall denote by $Alg(SIG, semantics)$ the collection of models of a signature SIG that comply with a certain *semantics*. The whole collection of models for a signature, no semantics, are denoted as $Alg(SIG, \emptyset)$, or just $Alg(SIG)$ for short.

There are some very simple models. So simple that they exist for every signature.

- WORD algebra.

  Every (ground) term is different from any other. Equivalence classes are one-element sets.

- BOTTOM.

  All the terms are equal. There is just ONE equivalence class encompassing ALL the (ground) terms.

We do not argue if these algebras are useful or not. We just state their existence. We can even find a third trivial model when variables are considered

- FREE algebra.

  Every term is different from any other. It is a generalization of the word algebra.

4.7.1 Example: bottom algebra

The bottom algebra is a nice model for some stupid programs as

```
program nut (input, output);
  begin
  while true do
    begin
    readln;
    writeln ("Ha!");
    end
  end.
```

4.7.2 Example: word algebra

The word algebra is the normal model for simple specifications of natural numbers:

```
type Naturals
  sorts
    nat
  opns
    0 :      -> nat        (* zero *)
    s : nat -> nat         (* successor *)
  endtype
```

Where the ground terms are 0, $s(0)$, $s(s(0))$, $s(s(s(0)))$, ... and every ground term is different from each other.

4.8 Homomorphisms

Informally speaking, the semantics of an algebra allows us to *evaluate* terms, meaning that given a complex term we are able to find a simpler one that is equal *in some sense*. For instance, in type Mod2,

$$Mod2 \vdash (0 + 1) + 1 = 0$$

There may exist different algebras, with different signatures and different semantics, but such that similar *evaluations* may be performed on any of them.

Suppose there is a morphism $f : A_1 \to A_2$ with the property that if we *evaluate* a term t in A_1 and get α, then moving into A_2 and evaluating there, will yield $f(\alpha)$.

$$\begin{array}{lll} A_1 \vdash & t & = \alpha \\ A_2 \vdash & f(t) & = f(\alpha) \end{array}$$

We are looking for relations between algebras. And obviously we are thinking of morphisms. But not any morphism. We need, we wish, to preserve some properties. We introduce the homomorphisms.

- *HOMORPHISM:*

  An **homomorphism** is a morphism between (signatures of) algebras such that two terms of the same equivalence class in the original algebra are mapped into terms of the same equivalence class in the image algebra.

Since operations are proper functions (i.e. deterministic), it is enough to require for every operation that the result of applying it here and then mapping there is the same as first mapping arguments and then applying it there. Formally,

Let $A_1 =< SIG_1, SEM_1 >, A_2 =< SIG_2, SEM_2 >$,

$h : SIG_1 \rightarrow SIG_2$ is an homomorphism iff $\forall t_1, t_2 \in |A_1|$,

$$A_1 \vdash t_1 = t_2 \Rightarrow A_2 \vdash h(t_1) = h(t_2)$$

and $\forall (op : s_1 \dots s_n \rightarrow s_r) \in F_1, t_1 : s_1, \dots, t_n : s_n$,

$$h(op(t_1 \dots t_n)) = h(op)(h(t_1) \dots h(t_n))$$

We shall usually write $h : A_1 \rightarrow A_2$. In the previous (long) definition, please notice that the implication goes from left to right. That is, NOBODY requires an homomorphism to guarantee that

$$A_1 \vdash t_1 = t_2 \Leftarrow A_2 \vdash h(t_1) = h(t_2)$$

Let's put a familiar example. The most widely used homomorphism I know of is the KERMIT[7]. At my institution I have a VAX and a SUN. Let *mine.p* be a pascal program. Let us have two evaluation functions[8]

VAX: pascal-program	$\rightarrow$	result in algebra VAX
SUN: pascal-program	$\rightarrow$	result in algebra SUN

[7] Kermit is a widely used program to transfer data from one computer to another.

[8] Compile + link + execute.

The KERMIT is a morphism, $kermit : VAX \rightarrow SUN$, that allows me to evaluate in the VAX and bring the result to the SUN, $kermit(VAX(mine.p))$ or bring it to the SUN and evaluate in this last system $SUN(kermit(mine.p))$. $kermit$ is an homomorphism because

$$kermit(VAX(mine.p)) = SUN(kermit(mine.p))$$

The property of kermit is that any program that runs in the VAX, will run in the SUN too, and we obtain the same result everywhere.

At my institution, the Pascal version in the SUN is a superset of the version in the VAX. So, it is guaranteed that any program that runs in the VAX will run in the SUN, but not the other way round. If it were too in both directions, we would have a nice *isomorphism*.

4.8.1 Properties

- Identity.

  $\forall A =< SIG, sem >$, $\exists$ homomorphism I,

  $I : SIG \rightarrow SIG$

  that maps every (ground) term on itself.

  $$\forall t \in |A|, I(t) = t$$

- Transitivity.

  Let $h_1 : A_1 \rightarrow A_2$ and $h_2 : A_2 \rightarrow A_3$ be homomorphisms.

  Then $h_1 \circ h_2 : A_1 \rightarrow A_3$ is an homomorphism too.

- Associativity.

  $$h_1 \circ (h_2 \circ h_3) = (h_1 \circ h_2) \circ h_3$$

- Bottom.

  $\forall A$, $\exists$ homomorphism h,

  $h : A \rightarrow$ bottom algebra.

  It is the trivial homomorphism that collapses everything.

- Partial ordering.

  The previous properties permits to define a partial ordering between models for a given signature.

  $$A_1 \geq A_2 \Leftrightarrow \exists h : A_1 \rightarrow A_2$$

4.9 Isomorphisms

Some homomorphisms are bijective. That means there is an inverse homomorphism such that the composition of both gives the identity

$$h \circ h' = h' \circ h = I$$

Such homomorphisms with an inverse are called **isomorphisms**. An isomorphism $i : A_1 \rightarrow A_2$ guarantees

$$A_1 \vdash t_1 = t_2 \Leftrightarrow A_2 \vdash i(t_1) = i(t_2)$$

Going back to the example of the *kermit*, if the two systems support the same Pascal standard, the *kermit* becomes an isomorphism.

4.10 Examples

Let's put again our type $Mod2$.

```
type Mod2 is
  sorts   bit
  opns    0, 1  :          -> bit
          _ + _ : bit, bit -> bit
  semantics
          + | 0 1
          ---+-----
          0 | 0 1
          1 | 1 0
  endtype Mod2
```

And let's consider naturals modulo 4.

```
type Mod4 is
  sorts   n.4
  opns    0, 1, 2, 3  :          -> n.4
          _ ++ _      : n.4, n.4 -> n.4
  semantics
         ++ | 0 1 2 3
         ---+---------
```

```
          0 | 0 1 2 3
          1 | 1 2 3 0
          2 | 2 3 0 1
          3 | 3 0 1 2
endtype Mod4
```

We can easily find an homomorphism

$$h : Mod4 \rightarrow Mod2$$

There are many of them. One of the easiest would be

$$\begin{array}{lcl} h(n.4) & = & bit \\ \\ h(0) & = & 0 \\ h(1) & = & 1 \\ h(2) & = & 0 \\ h(3) & = & 1 \\ h(++) & = & + \end{array}$$

Notice the equivalence of *evaluating* in both algebras

Mod4	$h(Mod2)$	Mod2
$1++2$	$h(1)h(++)h(2)$	$1+0$
3	$h(3)$	1

Notice there is no inverse homomorphism, so h is not an isomorphism.

Now, let's consider the algebra of parities.

```
type Parity is
  sorts   parity
  opns    even, odd  :                -> parity
          _ * _      : parity, parity -> parity
  semantics
            *  | even odd
          ------+----------
          even | even odd
          odd  | odd  even
  endtype Parity
```

Here we can easily find an homomorphism that is an isomorphism,

$$\begin{array}{rcl@{\qquad}rcl} i : Parity & \rightarrow & Mod2 & i' : Mod2 & \rightarrow & Parity \\ \\ i(parity) & = & bit & i'(bit) & = & parity \\ \\ i(even) & = & 0 & i'(0) & = & even \\ i(odd) & = & 1 & i'(1) & = & odd \\ i(*) & = & + & i'(+) & = & * \end{array}$$

4.11 Representation Independence

Last example shows that the existence of an isomorphism between two algebras means both algebras are very similar. In fact, the difference between them is just a matter of names. The isomorphism is not much more than a translation cipher that preserves semantics.

Thus, it is usual to say that two algebras related by an isomorphism are just two representations of the same idea. Abstract data types are algebras defined up to isomorphism. So, we say they are representation independent. That's the reason they are tagged as **abstract**.

There are many examples of isomorphic algebras:

binary	decimal
arabic numerals	roman numerals
ascii	ebcdic
prefix notation	postfix notation

4.12 Categories

As already stated, given an signature *SIG*, there may be many models or algebras

$$A =< SIG, semantics >=< S, F, semantics >$$

that share the same signature. These many models are related by homomorphisms that establish a partial ordering. The collection of models and homomorphisms is known as a **category**.

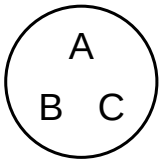
A model differs from others in the partition into equivalence classes.

4.12.1 Categories as DAGs

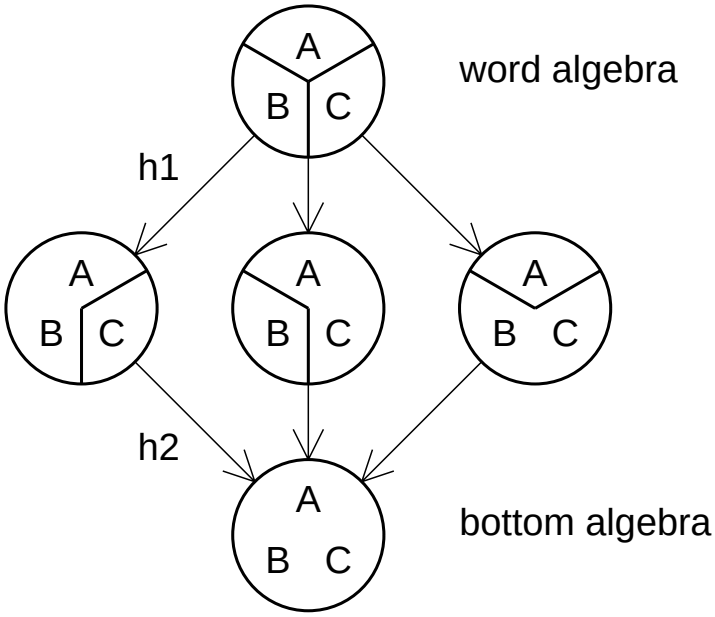
For the very simple type

```
type ST is
  sorts
    s
  opns
    A, B, C:  -> s
  endtype
```

The set of ground terms may be drawn as

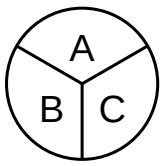


that may be partitioned in different ways

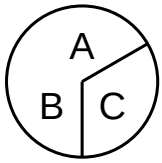


Each model having different properties[9]:

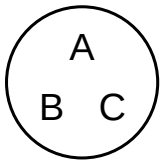
[9]We shall speak in terms of = and $\neq$ because that is what is needed in LOTOS.



A <> B, A <> C, B <> C



A = B, A <> C, B <> C



A = B, A = C, B = C

Since equality is transitive, we can just write $A = B = C$.

The point is that every single model can be uniquely determined by means of equalities and unequalities.

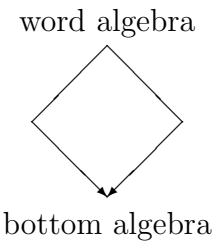
Models are ordered by homomorphisms (the arrows in the previous picture)

$$\begin{array}{lll} h_1: & h_1(A) & = A \\ & h_1(B) & = A \\ & h_1(C) & = C \end{array}$$

$$\begin{array}{lll} h_2: & h_2(A) & = A \\ & h_2(B) & = A \\ & h_2(C) & = A \end{array}$$

So, the category of models for a signature is a **dag** *(directed acyclic graph)* where the first node is the word algebra and the last, the bottom algebra.

Schematically, categories can be drawn as



4.12.2 Initial Algebra

For the slightly more complex type

```
type T is
  sorts
    s
  opns
    A, B, C, D:  -> s
  endtype
```

The set of ground terms is

.

that may be partitioned in 15 different ways. The total set of models, without caring for semantics, $Alg(SIG) or Alg(SIG, \emptyset)$ is as follows,

This category can be reduced by means of semantics. In the previous example we showed how to choose just one model by means of equalities and unequalities. More general situations may be devised.

If we impose just one equality, we restrict the category to

$$Alg(SIG, \{C = D\})$$

We can further reduce it by adding more equalities

$$Alg(SIG, \{C = D, A = C\})$$

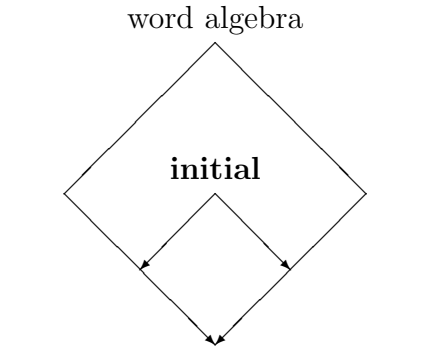
Notice there is a very closed relationship between imposing equalities and moving down along the graph of partially ordered models. The reason is obvious: an homomorphism cannot make distinct was was equal, but may make equal what was not specified.

Thus, there is always a sub-dag of partially ordered models. That dag has always a unique first node, by construction. That first node is called the **initial model** of the sub-category.

The initial model is the one that makes as many equivalence classes as possible. Any other model in the dag goes further identifying terms. In other words, in the initial model everything is different unless explicitly stated.

Because of homomorphism transitivity, we can formally define the initial model. Given a category Alg(SIG, sem), we say A is the initial model in the category iff for every model B in it there exists an homomorphism $h : A \rightarrow B$.

We can identify as well a last node in the dag. It is called the **final model**. For = specified sub-categories, it is ALWAYS the bottom algebra.



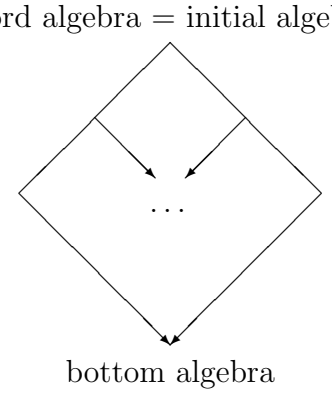
4.12.3 Unequalities

Semantics can be specified by means of unequalities. The effect on the category of models is to reduce it. For example,

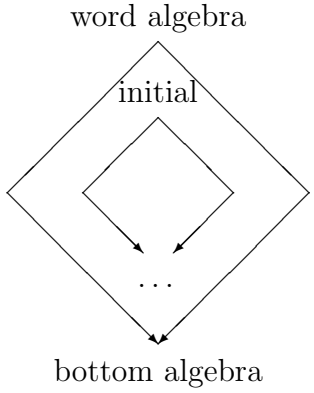
$$Alg(SIG, \{B \neq C, B \neq D, C \neq D\})$$

The initial model of sub-categories selected by unequalities is ALWAYS the word algebra.

Notice there is no unique last node. In this case, there are three. So, we say there is no final model.



If we use both equals and unequals to specify semantics, we reduce our dag from the top (by $=$) and from the bottom (by $\neq$). Nevertheless, by construction, we always have a first node, that is the initial model.



4.12.4 Positive Conditionals

It is usually of practical interest to use conditional axioms as

$$A = B \Rightarrow C = D$$

It must be read as follows: a model M agrees with this semantics if either $A \neq B$, or both $A = B$ and $C = D$. Notice that when $A \neq B$, C and D may be equal or not.

Let's suppose we just have equalities and positive conditionals as the one shown above. That is, there is no $\neq$ in the semantics.

The procedure to construct the sub-category is as follows. We have two kinds of rules

1. $A = B$
2. $C = D \Rightarrow E = F$

procedure build ();
begin
take the word algebra;
for each rule (1. $A = B$)
go *down* into the sub-dag $A = B$;
while $\exists$ (2. $C = D \Rightarrow E = F$)
if $C = D$ is in the current initial model
go *down* into the sub-dag $E = F$;
clear out this equation;
end;

Conditional axioms must be reconsidered every time we go into a sub-dag, because there is a new initial model to test. At the end, there may be some equations $C = D \Rightarrow E = F$ for none of which $C = D$ according to the current initial model. These ones are just dropped. This construction procedure guarantees there is always an initial model.

4.12.5 Negative conditionals

Negative conditionals may yield dags without an initial model. Just let's put an example.

$$Alg(SIG, \{B \neq C \Rightarrow A = B\})$$

The semantics may be expanded into

$$\begin{array}{l} B = C \\ B \neq C \quad and \quad A = B \end{array}$$

The sub-category is

Where there is no unique initial model.

4.13 The Model in LOTOS

In LOTOS only equalities and positive conditionals are allowed. Thus, there is always an initial model. There is always a final model too: the bottom algebra.

In LOTOS we just choose one model for each type, the *INITIAL MODEL*.

Notice we have to select one single model to make precise whether two terms are equal or not.

Sometimes we say LOTOS specifies types equationally with initial semantics. This expression is rather strange, but readers must be able to understand it so far.

5 Further Readings

[ISO, 88] is the standard reference for LOTOS, both syntax and semantics. It's hard to read, but it's definitely the last word about the language. A tutorial introduction may be found in appendix C of this document. A much better tutorial may be found in [Bolognesi, 87].

Nearly every paper on ADTs starts with a very brief presentation of signatures, sorts, etc. But every author has its own peculiarities and definitions, what creates lots of confusion

to beginners.

Pedagogic presentations can be found in [Wagner, 81] and [Burstall, 82]. They are intended for beginners.

An introduction to lattices (categories) of models can be found in [Broy, 81]. After introducing the objects, the authors go into demonstrating theorems. The standard reference for category theory is [MacLane, 72], but normal readers should not need it.

The last word on ADTs in ACT ONE[10] is [Ehrig, 85], but it is not really intended for beginners.

It has been shown in many papers, after [Majster, 77], that extra operations are usually needed to specify an ADT with a finite number of equations. These are called **hidden operations**. In LOTOS nothing is hidden, so if an extra operation is needed, it will be visible outside the ADT. Hidden operations arise the much more general question of **overspecification**: *are these the minimal requirements on the ADT?* or, *are we imposing more equivalences than required?* The answer to these questions is far from clear, is usually undecidable, and is definitely out of the scope of this tutorial.

References

[Bolognesi, 87] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems, 14(1), pp. 25-59, North Holland, Jan. 1987.

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

[Burstall, 82] R.M. Burstall and J.A. Goguen. Algebras, Theories and Freeness: An Introduction for Computer Scientists. Proc. 1981 Marktoberdorf NATO Summer School, Reidel, 1982. pp. 329-348.

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

[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.

[MacLane, 72] S. MacLane. Categories for the Working Mathematician. Springer, New York- Heidelberg- Berlin, 1972.

[10] The *mother* of LOTOS data types.

[Majster, 77] M.E. Majster. Limits of the "Algebraic" Specification of Abstract Data Types. SIGPLAN Notices, 12-10, pp. 37-42, October, 1977.

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