DIAGRAMMES

S. K. LELLAHI

Categorical abstract data type (CADT)

Diagrammes, tome 21 (1989), exp. n° 6, p. SK1-SK23

<http://www.numdam.org/item?id=DIA_1989__21__A6_0>

© Université Paris 7, UER math., 1989, tous droits réservés.

L'accès aux archives de la revue « Diagrammes » implique l'accord avec les conditions générales d'utilisation (http://www.numdam.org/legal.php). Toute utilisation commerciale ou impression systématique est constitutive d'une infraction pénale. Toute copie ou impression de ce fichier doit contenir la présente mention de copyright.

NUMDAM

Article numérisé dans le cadre du programme
Numérisation de documents anciens mathématiques
http://www.numdam.org/

DIAGRAMMES Volume 21, 1989

CATEGORICAL ABSTRACT DATA TYPE (CADT)

Received July 1, 1988

S.K. LELLAHI

Université. Paris 11, LRI Bât. 490
91405 Orsay Cedex FRANCE

Category theory is more and more used in studying abstract data types. Since long time, some authors used the notion of *sketch* to study the syntax and semantics of mathematical structures. This notion introduced and developed by Ehresmann [Ehre 68], is more powerful than the signature one, since it uses limits in categorical sense. Indeed, the signature approach uses nothing but products, while the notion of sketch allows not only to specify algebraic structures but also non algebraic ones such as fields.

A rigorous study of data structures shows that not all of them have an algebraic character; this is why we believe that the concept of sketch is a natural framework for the extension of algebraic abstract data types.

In this paper we show, in a simplified framework, how one may translate and extend the classical notions of algebraic abstract data types using sketches. The extension is called *categorical abstract data type* (*CADT*).

Keywords : Specification, Abstract data type, Sketch theory, Generalized algebraic data type, Categorical semantics.

INTRODUCTION

The algebraic concept of abstract data type (ADT) is nowadays considered as an efficient tool for the construction of programs which are reliable, reusable, extensible, less costly and easy to develop. The aproach is based on the mathemathical theory of multisort algebras and their specification by equations or other axioms [EhMa 85]. This theory is issued from the works of Birkoff [Birk 38] and Cohn [Cohn 65] on abstract and universal algebras. The abstraction of these works, by the application of technicques borrowed from category theory, leads to Lawver's theory [Lawv 68].

Thanks to the work of authors such as Zill [Zill 74], Guttag [Gutt 75] and specially the ADJ group [GTWW 77] the multisort algebras have found a great application ground in computer science.

A.M.S. Sub. class. : 68 B 05 18 C 10 68 B 10 18 A 10

An algebraic abstract data type is defined at two levels : syntax and semantics. The syntax is given by a signature, i.e. a set of names for abstract objects (sorts) and a set of names for links between these objects (operations). The semantics is defined by a set of axioms and the notion of algebra which consists of an interpretation for the objects and the links of the signature.

At the same time as Cohn and Lawvere, Ehresmann, following his research in differential geometry, introduced the notion of sketch [Ehre 66] as a basic notion for the specification of any axiomatic structure. This notion allows to specify not only algebraic structures but also structures whose laws are partially defined. Further work of Ehresmann and his students, mainly Lair [Lair 77] and Guitart-Lair [GuLa 80],[GuLa 82] have shown that sketch theory is a powerful tool for the study of the syntax and semantics of nearly all mathematical structures (see [Ehre 85]). These results are nowadays considered as a theorical foundation for a categorical study of first order theories [MaPa 87].

From a syntactical point of view, a sketch is independent from any interpretation. Intuitively a sketch is given by a set of abstract objects, a set of abstract links between objects, an abstract law which allows chaining links and a choice of a class of objects and links to characterize global properties of objects. A model of a sketch is then an interpretation for the objects, links and global properties in a more semantically known universe.

The intuition behind the concept of ADT and sketch shows the analogy of these two concepts at the syntactic and semantic level as well. The goal of this paper is to make this analogy explicit and to show that not only sketch theory is a natural extension of abstract data types, but also it becomes essential for a rigorous study of problems concerning abstract data types.

As a matter of fact, on the one hand authors working on abstract data types realize more and more that an informal and not rigorous application of category theory to abstract data types may lead to unexpected disapointments [Bern 87]. On the other hand, in the theory of universal algebras, multisort algebras and their applications, the only limits considered (in the categorical sense) are products. However, it has been proved that products are not sufficient for describing the syntax of theories whose axioms are not universally quantified or whose axioms need the use of "$\vee$" and/or "$\neg$". To handle such theories one must resort to other kinds of limits such as pullback, pushout or equalizer, etc. The consideration of these limits in sketch theory has allowed to specify other mathematical structures, such as fields, which are not specifiable by methods of universal algebras.

We argue that such structures are often met in the area of abstract data types. This is the case when the axioms of a data type contain inequalities or when one wants to handle error and exception cases. Consequently, a rigorous study of ADT needs the use of

limits other than products. This necessity becomes obvious when we intend to specify much more complex data types like those used in databases.

In order to make the paper self contained we recall, in Section 1, the basic notions of sketch theory. We suppose that the reader is familiar with the elements of category theory. Our terminology is similar (but simpler) than that of [Lair 87]. In Section 2, we reconsider some classical examples of data types and we present them in our approach. These examples show how one can specify a data type using sketches. An informal language of specification for sketches is also given in this section. The semantics is studied in Section 3. The notion of algebra in a specification is compared to the notion of model in a sketch. A sub section is reserved to the study of initial semantics. We show how it is possible to deduce results on initial semantics from well known results in general sketch and category theories. For instance; if a sketch uses colimits then the initial semantics may not exist. This is the case when the axioms of a data type contain inequalities [Kapl 87]. But in this case every connected component of the category of models of such data structure has an initial semantics.

In forthcoming papers we shall explore further the use of sketches . More precisely we shall use it for parametrized specification and abstract implementation and apply these concepts to specify a database. Gray [Gray 84] proposed tensor products of sketches to describe parametric data types but only special paramertric types can be defined in this way. We believe that the study of hierarchical structures and also stratified semantisc , introduced by Bidoit , [Bido 88] can be formalized using sketches.

1 BASIC CONCEPTS AND DEFINITIONS

In this section we introduce our terminology and we recall some basic definition from category theory.

1.1 C-graph and Category

In the following table the terminology used in this paper is compared with the one of graph theory.

in this paper	in graph theory
arrow	arc
object	vertex, node
domain	head
codomain	tail

In what follows we will often use lower case identifiers for arrows and upper-case

identifiers for objects. By a *loop* in a graph we mean an arrow whose domain and codomain is the same object. Any arrow x with domain A and codomain B is denoted by $x : A \rightarrow B$.

The *domain selection* and the *codomain selection* functions associated with a digraph G = <Arr_G, Obj_G> are denoted by *d* and *c* respectively. Thus :

$$d : \text{Arr_G} \longrightarrow \text{Obj_G} \quad ; \quad c : \text{Arr_G} \longrightarrow \text{Obj_G}.$$

If there is no risk of ambiguity we write *d*x for *d*(x) and *c*x for *c*(x). This kind of notation is used for all functions through the paper.

Unless otherwise specified, by a graph we mean a "polyadic digraph" (i.e. many arcs between two nodes are allowed) with loops such that for each object X, one of all possible loops on X is distinguished . This loop is denoted by 1_X and it is called the *identity* of X. Identities are called *trivial arrows* , whereas other arrows are called *proper arrows* . It is sometimes useful to confuse object X and its associated identity 1_X. Such consideration allows us to see a class of objects as a *discrete graph* (i.e. a graph with no proper arrows) and also permits us to drop identities when we draw a graph.

Let x and y be two arrows. We say that <x, y> is a *consecutive ordered pair* of G iff the domain of y is equal to the codomain of x.

Definition 1 A graph G is a *compograph* (*C-graph* for simplicity) iff a family Comp_G of consecutive ordered pairs of G is chosen in such a way that the following axioms are verified :

(i) For every arrow x the ordered pairs $<1_{dx}, x>$ and $<x, 1_{cx}>$ are in Comp_G.
(ii) For every <x, y> in Comp_G an element of G, denoted y.x, is chosen in such a way that *d*(y.x) = *d*x and *c*(y.x) = *c*y. Moreover $x.1_{dx} = 1_{cx}.x = x$.

Every graph G , can be considered as a *trivial C-graph* i.e. a graph in which Comp_G contains only the ordered pairs of form (i).

C-graphs were introduced and studied several years ago by Ehresmann [Ehre 65] and developed by many other authors working in category theory [Cope 78]. Such structures can be regarded as a graph G provided with a "partial binary operation" on its arrows for which every element has a right and a left identity. For this reason, if <x, y> is an ordered pair of arrows in G, we say <x, y> *composable* , or y.x *defined* , iff <x, y> belongs to Comp_G. Then the arrow y.x is the *composite* of <x, y> and y.x=z is called an *axiom* (or an *equation*) of C-graph G. A composite that has an identity as a factor is called *trivial*. If G is a finite C-graph then the operation of G can be represented by a table in which some entries may stay undefined. A C-graph is

associative iff for every x, y, z in G whenever one of the two composites z.(y.x) and (z.y).x is defined so is the other and they are equal.

Definition 2 A *category* is an associative C-graph in which every consecutive ordered pair is composable.

An arrow of a category is usually called a *morphism* .

Remark 1 In the previous definitions and in the rest of this paper the size of graphs can be " finite", "small" or "large". More precisely, a graph G is *finite*, *small* or *large* iff Arr_G is respectively a finite set, an arbitrary set or a class. Let G(A, B) be the set, or the class, of all arrows from A to B. We say that G is *locally small* if G is large but every G(A, B) is small. Note that the sets (classes) G(A, B) are always disjoint. All these considerations are valid also for C-graphs and categories. In a category $\mathcal{C}$ the notations Hom(A, B) or $\mathrm{Hom}_{\mathcal{C}}(A, B)$ are also used instead of $\mathcal{C}(A, B)$.

Definition 3 A *C-diagram* from a C-graph G to a C-graph G' is a correspondance who carries the structure of G on the structure of G'.

More precisely, $F : G \longrightarrow G'$ is a C-diagram iff :

i) F associates every object X of G to an object FX of G' and every arrow of G to an arrow of G' in such a way that $1_{FX} = F1_X$.
ii) $\forall \langle x, y\rangle \in$ Comp_G, $\langle Fx, Fy\rangle \in$ Comp_G' and $F(y.x) = F(y).F(x)$.

If G and G' are graphs (i.e. C-graphs with only trivial equations) then F is called a *diagram* and if G' is a category then F is called a *functor*. We can define the notions of *sub-graph*, *sub-C-graph* and *sub-category* as usual. That is a part of a graph, a C-graph or a category such that the canonical inclusion is a diagram, a C-diagram or a functor respectively. G' is a *full sub-C-graph* of G iff G' is a sub C-graph such that :

$$\forall A, B \in \mathrm{Obj_G'}, G'(A, B) = G(A, B).$$

Note that a full sub C-graph is entirely determined by its objects.

Remark 2 We draw the reader's attention to our definition of category as a particular graph . This definition, due to Ehresman [Ehre 65], differs from the usual one [MacL 71] . It allows to see a category as an "algebraic structure" with a partial and associative binary operation. It is a generalized monoid, so it can be manipulated like a monoid. The same analogy persists between functors and homomorphisms of monoids.

Usually categories are large graphs and have the same name as the generic name of their objects. For instance the category having sets as objects is named the *category of sets* (see example 2 below).

1.2 Examples

The following examples were chosen for their usefulness in the rest of the paper.

Example 1 Let $\leq$ be a reflexive and transitive binary relation on a set E. We can associate canonically a small category E to $<E, \leq>$ in the following way :

objects : $Obj_E = E$;

arrows : $Arr_E = \cup_{A,B \in E} Hom_E(A, B)$ and
$Hom_E(A, B) =$ if $A \leq B$ then $\{<A, B>\}$ else $\emptyset$.

composition : $<y, z> . <x, y> = <x, z>$.

In particular, for every set X the powerset of X with the inclusion relation $\supseteq$ defines a small category $\mathcal{Sub}_X$ (the subsets of X).

Example 2 $\mathcal{Set}$ is the category whose objects are all the sets of some universe, whose morphisms are mappings between these sets with usual composition of mappings. $\mathcal{Set}$ is a large category. There are many interesting subcategories of $\mathcal{Set}$. We denote by $\mathcal{D}set$ the full subcategory of $\mathcal{Set}$ having countable sets as objects.

Example 3 Three other large categories will be considered :

$\mathcal{G}raph$: the category of small graphs;
$\mathcal{C}$-$\mathcal{G}raph$: the category of small C-graphs.
$\mathcal{C}at$: the category of small categories.

In these three categories the morphims are respectively diagrams, C-diagrams and functors. The composition of morphisms is defined by composition of their underlying functions.

1.3 Cones and cocones.

Let us G be a C-graph. We can associate to G a C-graph $G^{\downarrow}$ such that G is a sub C-graph of $G^{\downarrow}$ and :

i) there exists only one object $V^{\downarrow}$ of $G^{\downarrow}$ not belonging to G;
ii) for every object A in G, $G^{\downarrow}(V^{\downarrow}, A)$ has exactly one arrow say $\pi_{V\downarrow,A} : V^{\downarrow} \longrightarrow A$
iii) for every arrow $\alpha : A \longrightarrow B$ in G , $< \pi_{V^{\downarrow},A} . \alpha >$ is composable in $G^{\downarrow}$ and

$$\alpha . \pi_{V^{\downarrow},A} = \pi_{V^{\downarrow},B}.$$

The C-graph $G^{\downarrow}$ is entirely determined by G and $V^{\downarrow}$.

Definition 4 Let Σ be a C-graph. A (projective) *cone* on Σ is a 3-tuple $\mu = \langle G, D, V \rangle$ in which :

- Σ is a C-graph and V is an object of Σ;
- G is a small C-graph and $D : G^{\downarrow} \rightarrow \Sigma$ is a C-diagram such that $D\,V^{\downarrow} = V$.

From now on, the following terminology and notations will be used.

- V is the *vertex* of μ;
- D, or more intuitively the image of G by D, is the *base* of μ;
- G is the *indexation* graph of μ, whereas the objects of G are indices;
- if A is in G then $D(\pi_{V^{\downarrow}_{,A}})$, denoted by $p_{V,A}$ or p_A, is a *A-projection* or a *A-coordinate* of μ.

Therefore for every $x : A \dashrightarrow B$ in G we have $Dx.\ p_A = p_B$. Note that other arrows, different than p_A can exist from V to DA in C-graph Σ and also several projections can exist with the same codomain. This is the main reason for introducing the indexation graph and the base, in the definition of a cone .

Cocone is the dual of a cone and can be defined in a similar manner by inverting the projection's direction (see figure 1)

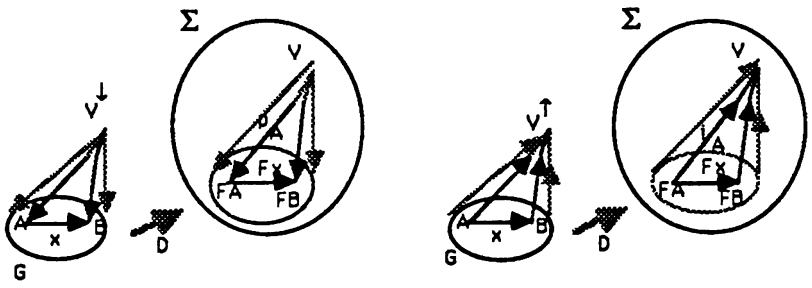


Fig. 1 : cones and cocones

An arrow like i_A in the figure is called an *A-coprojection* .

Generally G and D are implicitely defined. In this case, the image of $G^{\downarrow}$ (resp. $G^{\uparrow}$) by D is viewed as a cone (resp. cocone).

1.3.1 Special cones and special cocones.

In practice special cones and cocones are considered. We define and list them below :

Terminal object generator (TG) cone is a cone in which $G = \varnothing$. A TG is determined by the choice of an object in Σ as vertex. A TG cone is considered identical to its vertex and it is denoted by λ in this paper.

Product generator (*PG*) *cone* is a cone in which G is isomorphic to the trivial graph **n** = {1, 2,...,n} for n>1. Such a cone is determined by the choice of n arrows in Σ with the same domain : the vertex.

Pullback generator (*PULG*) *cone* is a cone in which G is a finite trivial C-graph formed by a family of arrows with the same codomain. Such a cone is determined by the choice of a family of commutative triangles in Σ, indexed by Obj_G, with a common side (figure 2a). A particular PULG in which G has only two arrows and in which two projections are identities (figure 2b) is called a *monic generator* (*MG*) *cone*.

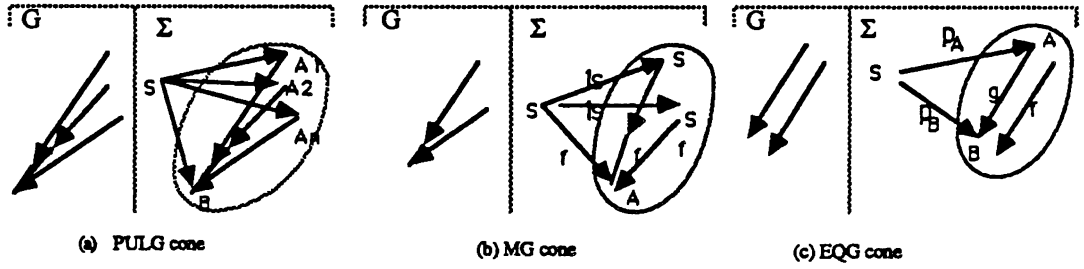


Fig. 2 : Special cones

Equalizer generator (*EQG*) *cone* is a cone in which G is formed by two distinct parallel arrows i.e. with the same domain and the same codomain (figure 2c).

Note that in the figure 2c some objects and arrows of Σ are duplicated for more clarity.

The special cocones, dual of the cones cited above are called *initial object generator* (*IG*), *coproduct generator* (*CPG*), *Pushout generator* (*PUSG*), *epi generator* (*EPG*) and *coequalizer generator* (COEQG) cocone, respectively.

1.4 Sketches

Definition 5 A sketch on Σ is a 3-uple $\sigma = < \Sigma, \mathcal{P}, \mathcal{J} >$ such that :

i) Σ is a C-graph;
ii) $\mathcal{P}$ is a family of cones on Σ;
iii) $\mathcal{J}$ is a family of cocones on Σ.

We can consider different kinds of sketches : *projective sketches* are those for which $\mathcal{J} = \emptyset$, *mixed sketches* are those for which $\mathcal{J} \neq \emptyset$ and $\mathcal{P} \neq \emptyset$ and *finite sketches* are those for which all items are finite sets. In this paper we consider only finite sketches.

Sketches were introduced and studied several years ago by Ehresmann [Ehre 66]. His aim was to specify through this notion, any axiomatic structure, not only the algebraic ones. For instance, using a mixed sketch one may specify the structure of a field. It is well known that this structure is non specifiable by the classical approach of abstract and universal algebras using signature and positve conditional axioms [EhMa 85]. There is an extensive litterature on sketches due to Ehresmann and his students ([Ehre 85]).

In our opinion, algebraic abstract data types and functional programming seem the natural area where the results of the theory of sketches can be applied. J.W. Gray uses this theory for parametrized specification and algebraic semantics [Gray 84] , [Gray 86]. In this paper we relate sketches to the specification theory. Our aim is to show how one can use the well known results of the theory of sketches in specification theory. For this reason we compare , at first, the terminologies of sketch theory and specification theory and we choose then the specification terminology [EhMa 85]

In the following sections we present some well known examples of data types but interpreted in our approach. More details and examples can be found in [Lell 87].

2 SKETCH AS A SYNTAX OF ABSTRACT DATA TYPES

Roughly speaking a sketch determines local and global properties of a theory. These properties are syntactic, so they are independent of any implementation of this theory.

Research on sketches revealed that every finite first order theory is sketchable [GuLa 82]. On the other hand any data structure can be defined by a first order theory. Following these results we can establish the fact below :

FACT 1. *The syntax of any data structure can be defined by a finite (perhaps mixed) sketch in which all operations are "total".*

Henceforth, such skecth will be called a *categorical specification* (*CSPEC* for simplicity). In comparison with the classical algebraic approach, we can regard some objects of a sketch as sorts, some arrows of a sketch as operations and some non trivial composites as particular axioms.

More precisely, we call the attention on two kinds of object in a CSPEC. We call *sort* any object which is not the vertex of any distinguished cone or cocone except a TG cone, a MG cone, a IG cocone or a EPG cocone. The other objects will be called *constructible sorts*. We distinguish also the arrows which are projections, coprojections or identities from other arrows. The later are called *operations*.This terminology is chosen in order to facilate the comparaison with algebraic approach of specification.

An operation whose domain is a TG cone (i.e. λ) and whose codomain is X, is called a *constant* of sort X. If c is a constant of sort X and $\lambda_A : A \to \lambda$ is an arrow then we suppose $< \lambda_A , c>$ is always a composable ordered pair. The composite is called a *constant operation with value* c and it is denoted by $\lambda_{A,c}$. The arrows of the form λ_A , $\lambda_{A,c}$, projection and coprojection and identities will be often dropped when we draw a sketch.

Remark 3 One may regard the distinguished cones and cocones of a CSPEC as type constructors. They are the internal constructors of the data type to be defined. This mean that they are the only tools allowing to construct new types from the ones named by the sorts. For this reason henceforth we call *sort constructor* any distinguished cone and *sort coconstructor* any distinguished cocone in a CSPEC. They become internal type constructors and internal type coconstructors in the semantic level (see section 3). Thus an object of CSPEC which is the vertex of a sort constructor (coconstructor) is called a *constructed* (*coconstructed*) *sort*. The reason for this terminology and consideration will be more clear in section 3 when we will define the semantics of a CSPEC.

As a conclusion of the above discussion we emphasize that a rigorous method for specifying a data type is to supply :

a set of sorts, a set of operations, a set of sort constructors,
a set of sort coconstructors
and/or a table which define the relations between operations (i.e. composition law).

The following section gives examples which will make clear these definitions.

2.1 Canonical CSPEC of an algebraic specification.

Let <S, W> be a multi-sort signature. We can canonically associate to <S, W> a projective categorical specification $\sigma = <\Sigma, \mathcal{P}>$ by the following procedures :

Obj_Σ :

- Obj_$\Sigma := \emptyset$;
- for every sort s of $< S, W >$ add s to Obj_Σ;
- for every $u \in S^* = \cup_{n>1} S^n$ such that $W_{u,s} \neq \emptyset$ for some $s \in S$, add u to Obj_Σ;
- add to Obj_Σ an object λ distinct from the others.

Arr_Σ :

- Arr_$\Sigma := \emptyset$;
- add every operations of <S, W> to Arr_Σ;
- for every X in Obj_Σ add to Arr_Σ an arrow $1_X : X \to X$;

- if $u = s_1 s_2 \ldots s_n$ ($n \geq 2$) and $W_{u,s} \neq \emptyset$ for some $s \in S$, then add to Arr_Σ the arrows : $p_{u,1} : u \rightarrow s_1$, $p_{u,2} : u \rightarrow s_2, \ldots, p_{u,n} : u \rightarrow s_n$

Composition :

- Arrows of the form 1_X are identies.The only composites are the trivials ones .

Sort constructors :

- a TG sort constructor (i.e. a sort constructor determined by a TG cone) with λ as vertex;
- for every u in S^* of arity $n \geq 2$ such that $W_{u,s} \neq \emptyset$, a PG sort constructor (i.e. a sort constructor determined by a PG cone) with **n** as indexation graph and $p_{u,1}$, $p_{u,2}$, , $p_{u,n}$ as projections.

Sort coconstructors :

- There are no sort coconstructors.

The CSPEC defined above is a projective CSPEC. If we have an enrichement of this signature by a system of axioms we must then enrich our CSPEC by adding some non trivial composite, other sort constructors and/or other sort coconstructors. The example 4 shows this technique.

Before studying the classical example STACK, we introduce some usefull notation.

If $G = \{s_1, s_2, \ldots, s_n\}$ is a discrete graph then by definition of $G^{\downarrow}$(resp. $G^{\uparrow}$) there exists only one object of $G^{\downarrow}$ (resp. $G^{\uparrow}$) which is not in G. This object is denoted by $s_1 s_2 \ldots s_n$ (resp. $s_1 | s_2 | \ldots | s_n$). The same notation is used for a PG cone and a CPG cocone respectively. If $s_1 s_2 \ldots s_n$ (resp. $s_1 | s_2 | \ldots | s_n$) is present in the C-graph of a CSPEC then this sketch contains, implicitely, a PG sort constructor (resp.a CPG sort coconstructor) with vertex $s_1 s_2 \ldots s_n$(resp. $s_1 | s_2 | \ldots | s_n$).

Example 4 Let us now consider the classical data type "STACK" [EhMa 85], [Bern 86]. STACK has a signature <S, W>, where S ={d,s} and W ={push, pop, top, empty, err}. The associated CSPEC of this signature is $<\Sigma, \mathcal{P}>$. This CSPEC is shown in figure 3 where $\mathcal{P} = \{D_1, D_2\}$ is the set of sort constructors with $D_2(0) = \lambda$, $D_1(1) = d$, $D_1(2) = s$, $D_1(12) = ds$, $D_1(1) = p_d$, $D_2(2) = p_s$.

Here s, d and λ are sorts and the only constructed sort is ds. Let us suppose that we want to enrich this signature. For instance, a specification of STACK is made by adding to this signature the following usual system of axioms :

$\forall x, \forall s,\ pop(push(x, s)) = s$; $\forall x, \forall s, top(push(x, s)) = x$;
$pop(empty) = empty$; $top(empty) = err$.

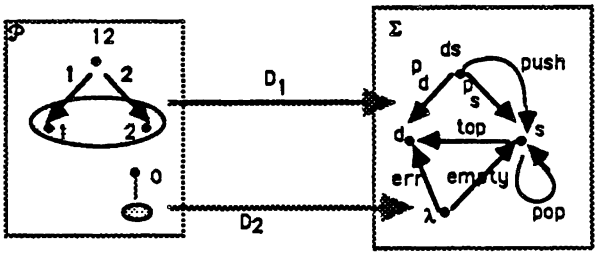


Fig. 3 : CSPEC of STACK

In our aproach this specification is obtained by adding the following non trivial axioms to the underlying C-graph Σ of the CSPEC above :

$$\text{pop.push} = p_s \quad ; \quad \text{top.push} = p_d \; ;$$
$$\text{pop.empty} = \text{empty} \; ; \quad \text{top.empty} = \text{err}.$$

The above axioms do not treat errors. Usually STACK with errors use predefined types NAT and BOOL and the new operations :

$$\text{empty?} : \text{STACK} \rightarrow \text{BOOL} \; ; \; \text{height} : \text{STACK} \rightarrow \text{NAT}$$

with some new axioms [BiGa 83], [Bern 86]. In our approach, using mixed sketches, a formal specification of STACK can be defined without any predefined types other than the data type "data",with the sort d, as parameter data type. We see STACK as a coproduct of empty stack and the set of non empty stacks. We treat then the pop and top operations slightly different on these two parts. This leads us to consider the sort s as a coconstructed sort. The associated CSPEC is shown in figure 4. In this figure "ne_" refers to non empty.

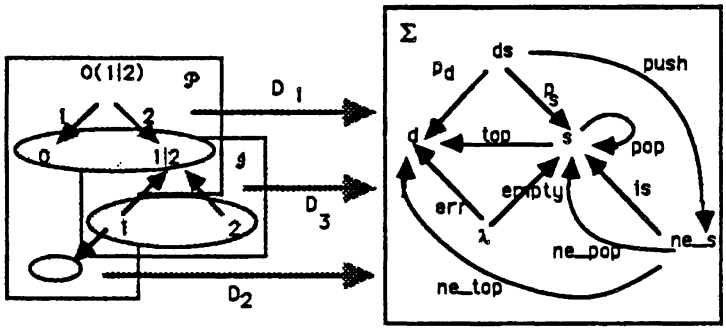


Fig. 4 : CSPEC of STACK with error

This CSPEC is obtained by adding to the previous sketch some trivial arrows and objects, adding a CPG coconstructor and to consider the following axioms :

ne-pop.push = p_s ;	pop.is = ne-pop;	pop.empty = empty
ne-top.push = p_d ;	top.is = ne-top;	top.empty = err.

We emphasize that the method for checking if a given stack is empty or not, can differ when we use predefined CSPEC. For instance, let BOOL be a predefined CSPEC in which b is the vertex of a CPG defined by two constants T and F. Thus b is a coconstructed sort. BOOL has also other usual operations, constructors and axioms (see [Lell 87]). Using BOOL and our notation we can make another CSPEC for STACK from the previous one as follows :

- drop ne_s, is, ne_pop,ne_top in C-graph but add s as a sort and empty?: s→b as an operation (b is the sort wich define BOOL),
- delete the sort coconstructor,
- replace in the axioms ne_pop by pop, ne_top by top and is by 1_S ,
- add the axioms empty?.empty = T and empty?.push = $\lambda_{ds,T}$.

Using variables these axioms are the classical axioms of specification of stack with errors [BiGa 83].Thus the CSPEC of stack with errors is a mixed sketch but all operations are total. In our approach we have not needed the use of a partial operation. By a technique like the one mentioned in this example, we can transform an exception or error case to a CSPEC with inductive part. i.e with coconstructors. The problems concerning errors and exceptions are not discussed in this paper. Mixed sketches are more complicated to handle.That is the case also for error handling. The reason is the existence of inductive part.

Remark 4 The above example shows that, we do not use variables to formulate axioms. Thus, the usual semantics part of a data type, i.e. the axioms, can be seen here as a "syntactic part". The real semantics is the instanciation and it will be studied in the section 3 of this paper. On the other hand, our explicit axioms are all of the form g.f = h where g, f and h are proper or trivial operations.

2.2 An informal language for CSPECs

Using this terminology we are led to the following informal language for specifying a CSPEC. This language uses the symbol "=" and some keywords defined in the last sections. For this informal presentation we follow closely specification laguages like OBJ2 and PLUS. The presentation below is more like the definition of module in OBJ2. Figure 5 summarize this language.

In such a representation, the absence of one of the parameters means that it does not exist in the CSPEC. Moreover if one sort is the vertex of another sort constructor (or of another sort coconstructor) then this sort is defined by the declaration of that constructor or that coconstructor. The example of stack with error in figure 6 illustrates this situation.

```
<name of CSPEC> =def
   uses : <list of eventual predefined CSPECs used>;
   sorts : <list of sorts not occuring in predefined CSPECs >;
   sort constructor
      <name of sort constructor> = def  {not occuring in predefined CSPECs}
         sorts : <list of sorts in the base not occuring in predefined CSPECs>
         operations : <list of operations in the base>;
         vertex : <name of the vertex>;
         projections : <names of projections >;
      end; { name of sort constructor}
   ...
   end; { sort constructors }
   sort coconstructors :    {not occuring in predefined CSPECs}
      <name of coconstructor> = def
         sorts : <list of sorts in the base> ;
         operations : <list of operations in th base>;
         vertex : <name of vertex>;
         coprojections : <names of coprojecctions >;
      end; { name of sort coconstructor}
   ...
   end; { sort coconstructors }
   operations :<list of operations not defined previously>;
   axioms : <list of axioms not defined previously>;
end.
```

Fig. 5 : syntax for CSPEC.

Example 5 Applying this informal language for the CSPEC defined in example 4 we obtain figure 6.

3 MODELS OF SKETCH AS SEMANTICS OF ABSTRACT DATA TYPE

Intuitively speaking, the semantics of a syntax is given by all possible meanings or interpretations of this syntax in a better known universe. In ADT theory, a semantics for a specification is generally determined by a class of algebras of this specification. This class can be an arbitrary class (i.e.loose semantics), the class of all algebras (i.e.the most loose semantics) or the class of initial algebras (i.e.initial semantics). Initial semantics is caracterized by an algebra isomorphic to the quotient algebra of terms modulo axioms. An algebra is a meaning of syntax defined by specification. In our approach the notion of algebra is replaced by the model of sketch. Consequently, we define semantics of a CSPEC as a full subcategory of a category of its models. But a full

subcategory is entirely determined by its objects. Thus, a semantics for a CSPEC is a class of its models. This shows the analogy with the algebraic approach. But what does it mean a model of a sketch?. Intuitively this is an interpretation of the syntax in a category having appropriate limits and colimits.

```
STACK = def
   uses : data
   sorts : { λ, ne_s};
   sort coconstructors :
      coproduct =
         sorts : λ, ne_s ;
         vertex : s ; { this is λ|ne_s }
         coprojections : empty : λ → s , is : ne_s → s;
      end; {coproduct}
   end; {coconstructors}
   sort constructors :
      terminal = def
         vertex : λ ;
      end; {terminal}
      product= def
         sorts : d,s ; {s is the vertex of coproduct, d is the sort of data }
         vertex : ds;
         projections : pd : ds → d , ps : ds → s ;
      end; {product}
   end ;{constructors}
   operations :
      err : λ → d      , push : ds → ne_s   ,   top : s → d ,
      ne_top : ne_s → d , ne_pop : ne_s → s   ,   pop : s → s ;
   axioms :
      ne_pop.push=ps ,     pop.is = ne_pop ,  pop.empty=empty ,
      ne_top.push= pd ,    top.is = ne_top ,  top.empty=err ;
end. {STACK}
```

Fig. 6 : STACK with errors.

3.1 Limit and colimit in a category

Let $\rho = <G, D, V>$ and $\rho' = <G, D', V'>$ be two cones in a category $\mathcal{C}$, with the same base. That is they have the same indexation graph G and $D(x) = D'(x)$ for any object or arrow x in G . We say ρ' *is factorized through* ρ *by a factor* $f : V \to V'$ iff $\rho' = \rho f$, that is for every A in Obj_G we have $pr_{V',A} = pr_{V,A}.f$.

Definition 6 ρ is a *limit cone* iff every cone ρ' with the same base as ρ can be factorized by a unique factor through ρ.

When ρ is a limit cone we note $V = \lim \rho$ and $f = \lim_{\rho} \rho'$. A cocone being *colimit cocone* is understood dually (see [MacL 71] for more detail). Limits and colimits are defined up to isomorphism. If a special cone (cocone) is a limit cone (cocone) it will take the same name as the cone (cocone) that generates it. For instance, if a product generator cone is a limit cone we call it a *product*, if a coproduct generator is a colimit cocone we call it a *coproduct* and so on (see [Lell 87]).

3.2 Categorical abstract data type

Let $\mathcal{C}$ be a category. Generally $\mathcal{C}$ is the category $\mathcal{S}et$ or the category $\mathcal{D}set$ of denumerable sets. Nevertheless the following definitions and results remain valid for any suitable $\mathcal{C}$.

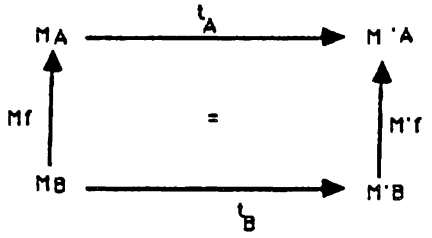
Let us consider the notation of last section and let $\rho = \langle G, D, V \rangle$ be a cone, G' be a graph and $M : G \longrightarrow G'$ be a diagram. It is obvious that $\langle G, MD, MV \rangle$ is a cone in G'. This cone will be denoted by $M\rho$.

Let $\sigma = \langle \Sigma, \mathcal{P}, \mathcal{I} \rangle$ be a sketch.

Definition 7 A $\mathcal{C}$*-model* of σ (called "réalisation " in [Ehre 66]) is a functor M from Σ to $\mathcal{C}$ that turns every distinguished cone into a limit cone and every distinguished cocone into a colimit cocone. A *model* of σ will mean a $\mathcal{S}et$-model. If s is a sort of σ the object Ms of $\mathcal{C}$ is called a $\mathcal{C}$*-carrier* of s.

Models allows to translate, in the category $\mathcal{C}$, the syntactic properties of σ. The category $\mathcal{C}$ is then considered as a well known universe.

Definition 8 Let M and M' be two $\mathcal{C}$-models of a sketch σ. A *morphism from* M *to* M' is a $t : \text{Obj_}\Sigma \longrightarrow \text{Arr_}\mathcal{C}$ such that for every A and B in Obj_Σ and every $f : A \rightarrow B$ in Σ the diagram below commutes :



In this diagram t_A denotes t(A). That is :

i) for every A in Obj_Σ , t_A : MA $\longrightarrow$M'A;
ii) for every f :A $\longrightarrow$B in Σ , M'f. $t_A = t_B$.Mf .

This definition of morphism is analogous to the definition of natural transformation between functors in category theory [MacL 71]. Thus morphisms between models of a sketch form a category. This category is called the *category of $\mathcal{C}$-models* denoted by $\mathcal{C}$-Mod(σ) . The category $\mathcal{C}$-Mod(σ) inherits some of the properties of the category $\mathcal{C}$. The category $\mathcal{C}$-Mod(σ) can be empty but if the set of constants of σ is not empty this category is not empty either (see next section). Mod(σ) will stand for the category $\mathcal{S}et$-Mod(σ) and Modd(σ) will denote the category $\mathcal{D}\mathcal{S}et$-Mod(σ). In this paper our interest will be Mod(σ) and Modd(σ).

From now on we suppose that σ is a CSPEC i.e. finite sketch. A $\mathcal{C}$-model of σ is called a *$\mathcal{C}$-categorical abstract data type ($\mathcal{C}$-CADT)* with syntax σ and a model of σ is called a *categorical abstract data type* (CADT) with syntax σ .

In most cases one provides $\mathcal{C}$ with a canonical method for construction of limits and colimits. For instance in $\mathcal{S}et$ we know how we can construct canonical product, coproduct, equalizer, etc...[Lell 87] .In this case, a $\mathcal{C}$ -model of σ is entirely determined by its restriction to sorts and operations of σ. In fact the image of a constructible sorts (i.e. vertex of some sort constructor or sort coconstructor) can be constructed in $\mathcal{C}$ by limit or colimit construction. This is the reason to call such a sort constructible. We call this restriction the *explicit part* of the $\mathcal{C}$-model. If σ is a sketch associated to an algebraic specification (section 2.1) with equational axioms then an explicit part of a model corresponds to an algebra which satisfies the axioms. More generally from well known results of sketch theory (see for example lemma of 3.2.2 of [MaPa 87]) we can assert the following fact.

FACT 2 *There exists a one to one and onto correspondence between projective CSPEC and algebraic specifications whith conditional positive axioms. Moreover if* SPEC = <S, Σ, E> *is a specification and* σ_{SPEC} = <Σ_{SPEC},$\mathcal{P}_{SPEC}$> *is the associated CSPEC then the two categories* Alg(SPEC) *and* Mod(σ_{SPEC}) *are isomorphic. It is the same for the categories* Mod(σ) *and* Alg($SPEC_\sigma$) *where σ is a CSPEC and $SPEC_\sigma$ is the associated specification.*

3.3 Semantics of a CSPEC

Intuitively speaking, $\mathcal{C}$ is a well known universe and models convey a meanning, in $\mathcal{C}$, for every syntax obtained by σ, in such a way that distinguished cones and cocones become limit cones and colimit cocones, respectively.

Definition 9 A $\mathcal{C}$-semantics of σ is a full sub-category of the categoy of its $\mathcal{C}$-models.

Whereas a full sub-category is entirely determined by the class of its objects, so a $\mathcal{C}$-semantic for σ is determined by a class of CADTs. This class is itself determined by a property on $\mathcal{C}$-mod(σ). The *initial $\mathcal{C}$-semantics*, if it exists, is obtained by the choice of the class of all CADTs isomorphic to an initial object of $\mathcal{C}$-Mod(σ). The initial semantics will be studied in the next section. Other semantics are *loose $\mathcal{C}$-semantics*. The category of all $\mathcal{C}$-CSPEC is the *most loose $\mathcal{C}$-semantics*.

The above definition and remark show that not every category K can be viewed as semantics for a given σ. The category K must have previously some properties. A powerful theorem of sketch theory (see [Lair 81] caracterizes the categories which can be isomorphic to a category of models. Lair's theorem is general and introduce large categories and large sketches. It is interesting to study the behaviour of this theorem for CSPEC (i.e. finite case) but this is beyond the scope of this paper.

3.4 Forgetful functor and synthesis functor

We first present a brief reminder about the forgetful and synthesis functors. Let $\sigma' = < \Sigma', \mathcal{P}', \mathcal{I}' >$ be a CSPEC. A C-diagram $u : \Sigma' \longrightarrow \Sigma$ will be called a *sketch morphism* iff $\mathcal{P} \supseteq u(\mathcal{P}')$ and $\mathcal{I} \supseteq u(\mathcal{I}')$.

Every sketch morphism $u : \sigma' \longrightarrow \sigma$ defines a functor $U : \mathcal{C}\text{-Mod}(\sigma) \longrightarrow \mathcal{C}\text{-Mod}(\sigma')$ by composition (i.e. $U(M) = M \circ u$ wher $M : \sigma \longrightarrow \mathcal{C}$ is a $\mathcal{C}$-CADT ; $U(t) = tu$ where $t : M \rightarrow M'$ is a morphism of $\mathcal{C}$-Mod(σ) and $(tu)_S = t_{u(s)}$ for s in Obj_Σ'). Using well known results of sketch theory we can stablish the fact below.

FACT 3 *If σ' and σ are projective (i.e. $\mathcal{I} = \mathcal{I}' = \emptyset$) then for every u, the correspondig U has a left adjoint when $\mathcal{C}$ is a "suitable category".*

Never mind what "suitable category" means. $\mathcal{S}et$ is a suitable category. Every category of models of a projective CSPEC is suitable. This is enough for many applications. In the general case a suitable category must have enough limits and colimits and some commutation properties of limits and colimits. Let us suppose that $\mathcal{C}$ is a suitable category.

For our purposes the interesting case is when σ and σ' are hierarchically dependent. That is $\Sigma \supseteq \Sigma'$, $\mathcal{P} \supseteq \mathcal{P}'$ and $\mathcal{I} \supseteq \mathcal{I}'$. In this case u is the identity on σ' and an adjoint , if it exists, is called a *$\mathcal{C}$-synthesis functor*.

3.5 Initial semantics

Let us now suppose $\sigma' = dis\sigma = \langle S_\sigma, \emptyset, \emptyset \rangle$ where S_σ is the set of sort of σ considered as a C-graph. In this case $\mathcal{C}$-Mod(σ') is isomorphic to the cartesian product of $|S_\sigma|$ copies of $\mathcal{C}$. Thus if $\mathcal{C}$ has an initial object I then $\mathcal{C}$-Mod(σ') has an initial object obtained by $|S_\sigma|$ copies of I. On the other hand left adjoint carries colimits, and initial objects are colimits. Consequently from fact 3 we deduce the following fact :

FACT 4 *If σ is a projective CSPEC and if $\mathcal{C}$ has an initial object then $\mathcal{C}$-initial semantics exists for σ.*

The proof is trivial. With these conditions $\mathcal{C}$-Mod(disσ) has an initial object I and a synthesis functor F exists, so F(I) is initial in $\mathcal{C}$-Mod(σ). In the particular case where $\mathcal{C}$ = *Set* the empty set is initial, so initial semantics exist for every projective CSPEC.

Definition 10 Let σ be a projective CSPEC. We call *prototype* of σ ([EHRE 68]) the smallest category Ptyp(σ) containnig σ such that every sort constructor of σ is a limit cone in Ptyp(σ).

The fact below gives a formal construction of Ptyp(σ).

FACT 5 *Ptyp(σ) exists and can be obtained by the following procedure :*

- Construct a category Lσ *which has the same objects as* σ *and the following morphisms :*

- *every arrow of σ is a morphism;*
- *if $\langle f, g \rangle$ is a consecutive ordered pair in* Lσ *and if g neither f are identity then add a morphism gf to Lσ as composite of g and f;*
- *if μ is a constructor of σ with vertex S and if μ' is a cone in* Lσ *with the same base as μ then add a morphism* $\lim\mu\mu'$ *to* Lσ *with appropriate limit conditions;*
- *nothing else is a morphism in* Lσ.

- Obtain Ptyp(σ) by the two rewriting rules below :

rule 1 *If g and f are arrows of σ and if k=g.f is defined then rewrite gf in k (not confuse gf and g.f).*

rule 2 *If S is the vertex of a constructor μ in σ and if S' is the vertex of cone μ' in* Lσ *with the same base asμ and if there exits $f : S' \rightarrow S$ such that $\mu f = \mu'$ then rewrite $\lim_{\mu'}\mu$ on f.*

3.5.1 Terms of a CSPEC. When σ contains the TG sort constructor λ, for every sort $s \neq \lambda$ the set $Term_\sigma(s) = Hom_{Ptyp(\sigma)}(\lambda, s)$ is called the set of *ground terms* of sort

s. Thus $Term_\sigma = \cup_{S\sigma} Term_\sigma(s)$ is the classical ground terms modulo equations. If f : $s_1 ... s_n \longrightarrow s$ is an operation and if t_i is a term of sort s_i then $f(t_1, ..., t_n)$ corresponds to $f(\lim_\mu' \mu)$ where μ and μ' are cones with the discrete base $\{s_1,...,s_n\}$ and wit vertex λ and s repectively.

Example 6 Let Nat_0 be the CSPEC defined below.

Nat$_0$=
sorts :{ s, λ}
sort constructors:
final =
vertex : λ
end;
operations : 0 : $\lambda \rightarrow$ s ; suc : s $\rightarrow$ s
end.

The category Ptyp(s) is then defined as follows :

Obj_ Ptyp(*Nat*$_0$) = {λ, s};
Hom(s, s) = { 1_S, $\underline{0}$} $\cup$ {suc^n | n$\geq$1} $\cup$ {$\underline{n}$ | n$\geq$1};
Hom(λ, s) = { 0 } $\cup$ {suc^n 0 | n$\geq$1}; Hom(λ, λ) = { 1_λ }; Hom(s, λ) = { λ_S}.

where suc^n = suc ...suc, $\lambda_S = \lim_\lambda s$, $\underline{0} = \lambda_S 0$, $\underline{n} = suc^n \underline{0}$. We see that $Term_{Nat_0}$ is isomorphic to N.

Example 7 Let Nat be defined as below.

Nat =
uses : *Nat*$_0$
operations : 1: $\lambda \longrightarrow$s ;
axioms : suc.0 = 1
end.

Using rewriting rules, suc^{n-1}1 is rewriten to suc^{n}0 and $Term_{Nat}$ = $Term_{Nat}$(s) is isomorphic to N (see [Lell 87] for more details).

We associtae now a set X_s of variables to every sort s and we call $X = (X_s)_{s\in S\sigma}$ a *system of variables* for σ. We can consider a new CSPEC σ(varX) as below :

σ(varX) =$_{def}$
uses : σ;
operations : x : $\lambda \longrightarrow$ s for each s $\in S_\sigma$ and each x$\in X_s$;

end.

A ground term of $\sigma(varX)$ is a *term with variables* of σ. Thus $Term_{\sigma(varX)}(s)$, denoted by $Term_{\sigma}(varX,s)$, is the set of terms with variables of sort s and $Term_{\sigma(varX)}$, denoted by $Term_{\sigma}(varX)$, is the set of all such terms. It is tedious but easy to prove the following fact.

FACT 6 *For every projective CSPEC σ which contains the TG constructor λ and for every system X of variabes there exists a model $Term_{\sigma}X$ of σ defined by :*

- $Term_{\sigma}X(s) = Term_{\sigma}(varX,s)$ for each sort s;

- if $f : s \longrightarrow s'$ and if t is a term of sort s then $Term_{\sigma}X(f)(t) = ft$ where ft is the composite of f and t in $Ptyp_{\sigma(varX)}$.

Like algebraic specification $Term_{\sigma}X$ is the free structure generated by X.

Remark A paper of Wells and Barr ([WeBa 87]) gives another method to construct terms for particular sketches. The above construction of terms is a natural generalisation of usual recursive construction of terms in algebraic specification. Therefore the proof by structural induction can be applied in this case..

CONCLUSIONS

In this papper we introduced an approach for studying syntax and semantics in abstract data types. In this approach a sketch incorporates both, the signature and axioms of an algebraic approach. This allows a treatement without variable, so this point of view can be seen as a more syntactic than algebraic view. Categorical abstract data types introduced here generalize algebraic data types by taking into account other limits and colimits (in the sense of category theory) in their definition, whreas the classical approach is based on only products as limits. Through this paper our approach is compared with classical algebraic approach. We showed how some well known result of sketch theory and category theory can be used for a rigourous theory of data types. We think this approach opens new rigorous formalism for error handling, parametrized data types and other problems in this area. For the data type having a system of positive conditional axioms, most semantics problems can be interpretated by free structure and adjoint functor. But for a data type for which the specification uses other colimits (coproduct, epimorphism, ...) other tools as locally free structures are needed. In a forthcoming paper we use this concepts to specify a data base.

ACKNOWLEDGEMENTS

I am grateful to Christian Lair for his help and remarks during many fruitful discussions. I wish to thank Nicolas Spyratos for his comments and suggestions on an earlier draft of this paper.

REFERENCES :

[Bern 86] G. BERNOT. *Une semantique pour une spécification différenciée des exceptions et des Erreurs,....* Thèse de doctorat de 3e cycle, Université Paris-Sud (Orsay), 1986.

[Bern 87] G. BERNOT. *Good Functor ... Are Those Preserving Philosophy* ,Proceedings Category theory and Computer Science, Springer Verlag, LNCS 283, 1987.

[Bido 88] M. BIDOIT, *The Stratified Loose Approach : A generalisation of Initial and Loose Semantics*, Rapport de Recherche N° 402 LRI Univ. Paris-Sud (Orsay).

[BiGa] M. BIDOIT, M.C. GAUDEL *Spécification des cas Exceptions dans les Types Abstraits Algébriques : Problèmes et perspectives*, Rapport de Recherche N° 146 LRI, Univ. Paris-Sud (Orsay) 1983.

[Birk 38] G.BIRKOFF, *Structure of Abstract Algebra* , Proc. Cambridge Philosophical Society, 31,433-454, 1938.

[Cohn 65] P.M. COHN. *Universal algebra* , Harper and Row, New-York, 1965.

[Cope 78] L. Coppey, *Algèbres de décompositions et précatégories*, Thèse de doctorat d'état, Université Picardie, Amiens 1978.

[Ehre 65] Ch. EHRESMANN, *Catégories et structures* , Dunod, Paris, 1965.

[Ehre 66] Ch.EHRESMANN, *Introduction to the Theory Of Structured Categories* ,Tech. Report 10, Univ. of Kansas, Lawrence, 1966.

[Ehre 68] Ch. EHRESMANN, *Esquisses et Types de Structures Algébriques*; Bul., Institu., Polit., Iasi, XIV, 1968.

[Ehre 85] C. EHRESMANN, *Oeuvre complète et commentés*, part I, édité par A.C. Ehresmann, Amiens 1985.

[EhMa 85] H. EHRIG B.MAHR; *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics*, Springer-Verlag , 1985.

[Gray 84] J.W. Gray; *Categorical Aspects of Parametric Data Types*; Seminabericht 20, Fern Universitat, Hagen, 1984.

[Gray 86] J.W. GRAY, *The Category of Sketches as a Model for Algebraic Semantics*, (preprint version).

[GTWW 77] ADJ : I.A.GOGUEN, J.W.THACHER, E.G.WAGNER, J.B.WRIGHT, *Abstract Data Types and Initial Algebra and the Correctness of Data Representations.* Proceedings of the Conference on Computer Graphics, Pattern Recognition and Data Structures, 1975.

[GuLa 80] R. GUITART, C. LAIR, *Calcul Syntaxique des Modèles et Calcul des formules internes*; Diagrammes, Vol. 4, 1980.

[GuLa 82] R. GUITART, C. LAIR, *Limites et Colimites pour Représenter les*

Formules, Diagrammes 7, Paris, 1982.
[Gutt 75] J.V.GUTTAG, *Abstract data types and the Developement of data : Abstraction, Definition, and Structure* , SIGPLAN Notices, 8, no 2, 1976.
[Kapl 87] S. KAPLAN, *Positive/Negative Conditional Rewriting*, Proceedings Category theory and Computer Science, Springer Verlag, LNCS ???, Orsay 1987.
[Lair 75] C. LAIR, *Etude Générale de la catégorie des esquisses*, Esquisses Mathématiques 23, Paris 1975.
[Lair 77] C. LAIR, *Esquisse des Structures Algèbriques*; Thèse de doctorat d'état en math., Univ. de Picardie 1977.
[Lair 81] C. LAIR, *Catégories Modelables et Categorie Esquissable*, Diagramme 6, paris 1981.
[Lair 87] C. LAIR, *Esquisses, topos et Modèles* , Cours de DEA, Univ. Paris VII, 1987.
[Lawv 68] F.W.LAWVERE, *Some Algebraic Problems in the Context of Functorial Semantics of Algebraic Theory* , Lect. Notes in Math. 61, Springer-Verlage, 1968.
[LeJo] S.K. LELLAHI, G. Jomier; *An Algebraic Approach To Relational data Bases;*Proceedings of the Third Symposium on Computer and Information Sciences (ISCIS); Cesme, Izmir, Turkey, 1988.
[Lell 87] S.K. LELLAHI, *Types abstraits catégoriques : Une Extension des Types Abstraits Algébriques*, Rapport de Recherche N° 63 ISEM, Univ. Paris-Sud (Orsay), Novembre 1987.
[MacL 71] S. MAC LANE, *Categories for the working Mathematician*, Springer-Verlag , New York, 1971.
[MaHu 85] P.MATEI and F. HUNT, *Precision Descriptions Of Software Designs : an example* , IEEE Compsac, 130-136, 1985.
[MaPa 87] M. MAKKAI, R. PARE, *Accessible Categories : The fondation of categorical Model Theory* , Technical Report, Dept. of Math. and Stat. , Univ. Mc Gill, 1987.
[SSE 87] A. SERNADA, C. SERNADA, H.D. EHRICH, *Object-Oriented Specification of Data-Bases, An Algebraic Approch* , VLDB 1987, 107-116.
[Wagn 86] E. WAGNER. *Algebraic Theories, Data Types, and Control Constructors* ; Fundementa Informatica IX(1986), 343-370.
[WeBa 87] C. WELLS, M. BARR, ; *A Formal Description of Data Type Using Sketches;* Proceedings Category theory and Computer Science, Springer Verlag, LNCS 283, 1987.
[Zill 74] S.N. ZILLES, *Algebraic specification of data types*, Project MAC progress Report 11, MIT, Cambridge, Mass, 28-52, 1974.

Author's alternative adress :
Université Paris 13 IUT,
Av.Jean.Batist. Clément
93430 Villetaneuse FRANCE.