AAECC **14**, 187–215 (2003)
Digital Object Identifier (DOI) 10.1007/s00200-003-0129-1

AAECC
Applicable Algebra in Engineering, Communication and Computing

An Object-oriented Interpretation of the EAT System

Laureano Lambán, Vico Pascual, Julio Rubio
Universidad de La Rioja, Luis de Ulloa s/n Edificio Vives, Logroño (La Rioja), España
(e-mail: {lalamban, mvico, jurubio}@dmc. unirioja.es)

Received: June 12, 2002; revised version: May 6, 2003

Abstract. In a previous paper we characterized, in the Category Theory setting, a class of implementations of Abstract Data Types, which has been suggested by the way of programming in the EAT system. (EAT, Effective Algebraic Topology, is one of Sergeraert's systems for effective homology and homotopy computation.) This characterization was established using classical tools, in an unrelated way to the current mainstream topics in the field of Algebraic Specifications. Looking for a connection with these topics, we have found, rather unexpectedly, that our approach is related to some object-oriented formalisms, namely hidden specifications and the coalgebraic view. In this paper, we explore these relations making explicit the implicit object-oriented features of the EAT system and generalizing the data structure analysis we had previously done.

1 Introduction

1.1 The role of modelling in Symbolic Computation

Formal methods, understood as the application of mathematical formalisms to software engineering, are nowadays a central part in many methodologies of software development [32], [5], [2]. Even if the formal methods are now mainly related to specification and validation tasks, there is a wide consensus on the idea that formal methods should be also used in other areas. In particular, there is an increasing interest in the application of formal methods in *quality control*.

Though the mathematics could be the same in both cases, there are several differences when a formalism is used to specify a system or when it is used to analyze system features. A first obvious difference is a temporal one: the

Partially supported by MCyT, project TIC2002-01626 and by Comunidad Autónoma de La Rioja, project ACPI-2002/06.

specification occurs at *early stages* of the software development process and the quality control at *later stages*. In other words, the specification takes place with non-yet-existing systems while the quality control is performed on existing systems. Let us express this idea by saying that, in the first case, the formal methods are used to *design* a system, and in the second one they are used to *model* a system[1]. This simple remark has important consequences. The first one is that the design is a *prescriptive* task, while modelling is a *descriptive* task. The former indicates how things *should be*, while the latter concentrates on how things *are*. This means that in the design the degree of freedom is much higher than in modelling. Another consequence has to do with the *abstraction levels* allowed in each task. In the design, there is a wide range of abstraction levels which can be used. In an ideal scenario, starting from a very general formal specification, successive refinements give rise to a very detailed design, very close to the running code. Several proposals have been documented in the literature (see [48] for instance) to obtain executable programs from formal specifications. Nevertheless, a formal specification is usually a very abstract artifact (an algebraic specification, for instance) in which some algorithmic aspects can be overriden. On the contrary, in the modelling, even if several levels of abstraction can also be used, it is clear that the formalism must deal with *real* properties of the system under observation. In particular, the implementation programming language can be elided in the design, but it is mandatory to deal with its peculiar characteristics in the modelling.

As a general rule, it can be said that the more formal the initial specification of a system, the easier its modelling. In systems where the development process has been done almost without reference to documents of formal requirements, the modelling task is more challenging and, also, more necessary. This is the case, generally speaking, of the symbolic computation systems, where the early stages of the life cycle of software may be poorly structured. One reason for this trend could be the (erroneous) belief that, since a computer algebra system deals with formal mathematical objects, this is unnecessary to formally prepare the software development. However, the object representing a mathematical object in the computer memory is not *a* mathematical object itself (or rather it is a mathematical object of a very different nature than the initial one). This point of view is always present in our approach. Another pragmatic reason for this kind of development of non-commercial symbolic computation programs is that they are frequently written in academic contexts, where the feasibility of the algorithms (usually found by the same persons who write the programs) and the illustration of their theoretical interest are put before the software engineering guidelines. Anyway, it is clear that the modelling in symbolic compu-

[1] It is worth noting that the term *modelling* is sometimes also used in software engineering associated to early stages: a model is built and then is implemented. We are using the term modelling in a broader sense: to make a (mathematical) model of a situation or fact. In our case, we construct models on an already existing software system (namely, the EAT system).

tation becomes a main issue in order to increase the reliability of the systems. This point is especially important in systems capable of computing algebraic objects whose correctness has not been, up to date, confirmed (or refuted!) by other theoretical or mechanical methods. This is the situation of EAT [42] and Kenzo [15], Sergeraert's systems devoted to symbolic computation in Algebraic Topology. A brief overview of the EAT system is given in the next subsection.

1.2 Symbolic Computation in Algebraic Topology

Algebraic Topology does not seem to be, in principle, a *computational* discipline. Symbolic computation in Algebraic Topology presents some particularities which make difficult its realization on actual machines. On the one hand, the theory in Algebraic Topology works with very rich mathematical structures (chain complexes, simplicial sets and so on), whose translation to computer data structures is far from obvious. On the other hand, the algorithms in this field require that mathematical structures are constructed at runtime, as intermediary steps of the calculations. Hence, the programmer is confronted to the problem of handling complex data structures at runtime, with the additional difficulty posed by the fact that some of these intermediary mathematical structures are of infinite nature. Let us illustrate these points with a sample of an EAT session.

The implementation language of EAT is Common Lisp [49], and this is also the interface language. Let us assume that we are interested in the homology group $H_5(\Omega^2 S^3)$, that is to say, the fifth homology group of the second loop space of the sphere of dimension 3, S^3. The loop space of a topological space X is, roughly speaking, the space of continuous functions from the 1-sphere S^1 (the circle) to X which preserve the base point, endowed with a convenient topology (the compact-open topology). This space, usually denoted by ΩX, is infinite dimensional. The construction can be iterated: we can consider the loop space of ΩX, denoted by $\Omega^2 X$ and so on.

In EAT [42] we can compute the abovementioned homology group by typing:

```
> (cc-homology (oeh-ecc (loop-space-eh (ess-sseh
(sphere 3)) 2)) 5)
Component Z/3Z
Component Z/2Z
---done---
```

Here, the Common Lisp expression following the prompt symbol > has been typed by the user; then EAT displays the text `Component...` This text explains to the user that $H_5(\Omega^2 S^3) = \mathbb{Z}_2 \times \mathbb{Z}_3$, an abelian group with 6 elements. Apparently, this interaction is not very different from standard Maple or Mathematica experiences: we type some predefined utilities and we introduce some standard data. Namely, three integers: the dimension of the sphere (3), the

degree of iteration (2) and the degree of the sought homology group (5). Then, the system responds with some standard data or, as in this particular case, with a text displayed on the screen.

During the previous computation process, several (computer representations of) mathematical structures have been constructed, at runtime. To be precise, in this particular example it has been needed to construct 94 (computer representations of) chain complexes, 328 chain complexes morphisms and up to 9 different kinds of mathematical structures.

Furthermore, if we give details on the process, some more peculiarities appear. We will start assigning to the symbol `sphere3` the sphere of dimension 3, S^3.

```
> (setf sphere3 (sphere 3))
[SS-4]
```

The returned result indicates that a Simplicial Set has been constructed: the object (displayed) `[SS-4]` is (the machine representation of) that topological space. Now, we are going to briefly set out some details on the EAT way of representation. Using the Common Lisp function `inspect` (see [49], page 698) it can be seen how these mathematical structures are stored in the computer.

```
> (inspect sphere3)
```

The previous typing shows us the picture below, where we can see the structure of the object `sphere3`.

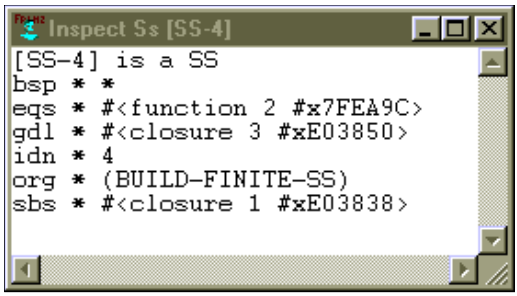

Fig. 1. Inspect of s3

This example shows that a mathematical structure is coded in EAT essentially by as a tuple of functions. The main slots in the previous object are of functional nature, either primitive functions (`#<function ...>`) or lexical closures (`#<closure ...>`) defined by the program; the rest of slots are interesting from the software engineering point of view but irrelevant for the modelling.

The object `[SS-4]` represents a topological space of *finite type*, it is an implementation of a finite mathematical structure. Sergeraert called this kind

of object *effective object* (this terminology appeared published for the first time in [41]). The EAT system is capable of obtaining the list of simplices of this object in, for instance, dimension 3:

```
> (sbs sphere3 3)
(<S3>)
```

Hence, there is only a (geometric) simplex in dimension 3 (internally denoted by the symbol `<S3>`). But, in the computation of the homology group $H_5(\Omega^2 S^3)$, another kind of spaces are involved, for instance the second loop space of S^3:

```
> (setf l2s3 (loop-space (loop-space sphere3)))
[SS-6]
```

The object (displayed) `[SS-6]` is also (the representation of) a simplicial set, but of *infinite nature*, a *locally effective object* in the terminology of [41]. The object `l2s3` includes the non-finite nature of the mathematical structure that is being implemented and gathers all the information needed to carry out the calculations. For instance, we can compare simplices, we can apply the face operators, etc. The following expression allows us to compute the 0-face of a simplex of `l2s3`.

```
> (gdl l2s3 0 3 (asm nil (loop2 (asm nil (loop3 nil
'<S3> 1)) 1)))
<ASM 1-0 <<LOOP *>>>
```

But unlike what happens with the object `sphere3`, the set of simplices is not available in the computer, since there are dimensions with an infinite number of elements. This fact can be seen in the picture below, which corresponds to the EAT representation of the object `l2s3`. In this case, there is no function in the slot `sbs`, but the label `:locally-effective`.

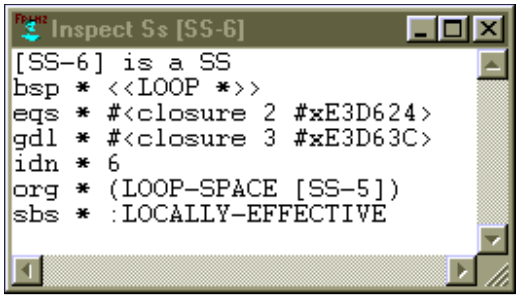

Fig. 2. Inspect of l2s3

Therefore, in the case of the object `l2s3`, it is not possible to compute its simplices in a given dimension:

```
> (sbs l2s3 3)
;; Error: The simplicial set [SS-6] is locally-
effective
```

However, EAT incorporates complex homological algorithms (see details in [47], [40]) that allow the program to obtain relevant information on `l2s3`, such as the fifth homology group computed above. As previously explained, EAT has provided homology groups which up to date could not be calculated by any other means (this is not the case of $H_5(\Omega^2 S^3)$ which is well-known; see more interesting examples in [40], [43]). Since no formal specification of the software system was written *before* developing the programs, the reliability of the system (apart from the exhaustive testing already done) relies on its formal *modelling*. Taking into account the handling of infinite (locally effective) objects, it is clear that the modelling is not a trivial task. As a mathematical tool to undertake this task we have chosen algebraic specifications.

1.3 Algebraic Specification of the EAT system

It is well-known that algebraic specifications provide an appropriate means to develop correct software, and in fact, they are one of the main formalisms to the above mentioned *design* task. Algebraic specifications can also be used as *modelling* tools, enabling the analyst to reason, in an abstract way, on the data handled by the programs. If the system to be analyzed is a symbolic computation package, the algebraic specifications provide, through the notions of signature and algebra, the necessary link between the data structures of the programs and the mathematical structures (based on universal algebra concepts in the field of Algebraic Topology) which are the ultimate goal of the calculations.

In [27] we undertook the analysis of the data structures appearing in the EAT system [42]. This program was designed to calculate the effective homology [47] of some complex topological spaces, namely iterated loop spaces (as it has been shown in the previous subsection). These spaces are of an infinite nature, and thus, some data structures in EAT must be designed to encode these infinite spaces (in a finite way, of course). From our first attempts, we were convinced that the EAT way of working with these structures follows a *pattern* with certain generic properties (it should be considered a universal pattern), in such a way that it could be exported to other systems of similar characteristics. The search for these properties took us to the definition of an operation between abstract data types (called $()_{Imp}$) and to a characterization of the EAT data structures as final objects in suitable categories of implementations [27].

This result was proved within a framework based on the well-known 1972 Hoare's paper [24], and far from other more algebraic approaches (some of them are surveyed in [34]). Looking for relations between our work and the current trends in the field of Algebraic Specification, we have found that our approach was close to the theoretical studies on object-oriented programming.

This finding was unexpected because, in our minds, the object-orientation in the effective homology setting began with the next-generation Sergeraert's system, called Kenzo [15], which was developed in CLOS (Common Lisp Object System). Thus, in principle, EAT depends on functional programming (as stressed, and necessarily used, in [27]) and Kenzo depends on object-oriented programming (but functional programming is still present in Kenzo). In this paper we will show that, in fact, EAT is also object-oriented or, at least, a simplified object-oriented theory provides tools to easily specify its data structures. More precisely, object-oriented features appear in EAT due to the handling of *families of implementations* of Abstract Data Types (ADTs from now on).

The relation between EAT and the object-oriented paradigm is then twofold. On one hand, the EAT programming methods are close to certain formalisms (variants of λ-calculi) introduced by Cardelli and other authors (see, for instance, [1], [8], [7]), in order to model object-oriented programming languages. On the other hand, the operation on Abstract Data Types which we introduced in [27] with the aim of explaining EAT data structures can be interpreted in terms of object-oriented algebraic specifications, more precisely in the context of the hidden specifications (see, for instance, [20], [6]) and in the coalgebraic framework (see, for instance, [39], [44]).

1.4 Scope and Organization of the Paper

In this paper it is shown that the ideas suggested by certain formalisms for the study of object-oriented programming can be applied to analyze a concrete symbolic computation system, namely the EAT system. As a consequence, some results extending the previous work [27] are introduced.

As far as its originality, this paper can be seen as an application from symbolic computation to the theoretical study of object-oriented programming. On one hand, because two main trends in the formalization of object-oriented programming appear in the very concrete problem of analyzing a symbolic computation system for Algebraic Topology. On the other hand, this relationship allows us to present another descriptions of the final objects appearing in object-oriented specifications and studying them in larger contexts.

The main theoretical contribution of the paper consists of the results in Sections 4 and 6 and the techniques used to derive them.

The paper is organized as follows. Section 2 introduces some object-oriented terminology and some clues on the implicit object-oriented nature of the EAT system. Section 3 deals with the algebraic specification of the EAT data structures, introduces the key notion of Σ_{Imp}-algebra and explains our primary approach to this notion, namely from a programming point of view. The notion of Σ_{Imp}-algebra is then studied from three additional perspectives. First, Section 4 illustrates our proposal for interpreting Σ_{Imp}-algebras in the context of Category Theory. The two previous views are compared with hidden

specifications and with coalgebras in Section 5. Then, we go back to symbolic computation and in Section 6 the results on EAT data structures of [27] are generalized. The paper ends with a section of conclusions and further work.

2 Object-oriented Features

When talking about object-oriented programming, one can usually think of several concepts: objects, of course (that is, entities which share the characteristics of data and behavior), classes, methods, message passing, local state or object identity (the possibility of changing attribute values in an object, while the object is still "the same"), inheritance, modularity and/or information hiding, generic functions, polymorphism, etc. However, it is quite unlikely to find an actual object-oriented system supporting all these features. The three necessary items for object-orientation are (see [1], [52]): objects, state and inheritance.

Let us present a simple example. To work with *points* (pairs of integer numbers) we can consider methods to access to the first and the second component, and, for instance, an operation to move points. This leads us to introduce three functions (expressed as a C-like signature):

```
int first ();
int second ();
point move (int,int);
```

as the ingredients of a point. So, a typical example to introduce the underlying ideas of the object-oriented programming could be the following simple signature for points:

$$\begin{array}{rcllll} cons & : & int & int & \rightarrow & p \\ first & : & p & & \rightarrow & int \\ second & : & p & & \rightarrow & int \\ move & : & p & int\ \ int & \rightarrow & p \end{array}$$

Ignoring the constructor *cons* (this kind of operations deserves a different treatment), we observe that these two signatures are closely related. This relation will be tackled later in this paper. In an informal way, we can state that the latter is a signature for a *class* and the former is a signature for *objects* (*i.e.*, instances of the class). The last operation, *move*, is a simplified way to deal with *updating operations*, but without considering *side-effects*. This allows the analyst to work within a pure functional programming framework which is easier to model (this decision is justified because the subsequent inclusion of side-effects has become a matter of routine; see [1]). These simplifications are shared in the field of programming languages (see, for instance, [1]) and in the field of algebraic specifications (see, for instance, [20]), where this kind of examples are usually chosen as a starting point for more elaborate work. But, what is the relation between such an example and the complex structures

of a symbolic computation program such as EAT? Three clues lead us to the relations between EAT and object-oriented programming.

The first clue is Cardelli's model of "objects as records of functions" [7]. Now, let us recall how algebraic structures are encoded in EAT (this point has been already illustrated in Subsection 1.2). For example, chain complexes (that is to say, differential graded free abelian groups) are encoded by a Common Lisp record (struct) which starts as ([42], page 9):

```
(defstruct cc eqc dp ...)
```

where `cc` is the name of the structure, the field `eqc` is intended to be a function (the equality test between generators), the field `dp` will contain another function (the differential operator) and so on. It is very tempting to reverse the metaphor and consider these "records of functions" as "objects". In doing so, we find very special kinds of objects (due to implementation decisions in EAT). For instance the signature below is a suitable simplification (idealization) of the actual interface for chain complexes in EAT:

$$\begin{array}{lcllll} cc-eqc & : & cc & gnr & gnr & \rightarrow \; bool \\ cc-dp & : & cc & cmb & & \rightarrow \; cmb \\ \ldots & & & & & \end{array}$$

Here, *cc* is the sort for *chain complexes*, *gnr* is the sort for *generators*, *cmb* is the sort for *linear combinations* on generators (see [42] for details) and, as usual, *bool* is the sort for boolean values. It is important to note that this kind of signature has no *updating operations*, even in the abovementioned simplified functional sense. They are objects whose behaviour is always to be observed, or, let us say, *immutable objects*. This fact shows that the division between *immutable data* and *mutable objects*, which is quoted in several papers in the hidden specifications domain (see, for instance, [30]), is not so sharp. The difference is rather between data (which are usually specified as immutable entities) and elements sharing data and behavior, namely objects (which can be mutable or not), as it is frequently explained in books on object-oriented programming. From our point of view, an immutable object is nothing but the representation of an *implementation of an ADT*. Or, rather, the representation of the *physical part* (that is to say, the part which is present in the computer memory) of such an implementation. For example, an instance of the struct `cc` would be the minimal information to recover (the implementation of) a chain complex (see Section 6 for details). And this minimal information indeed looks like an object, in Cardelli's sense.

The second clue is related to the kind of result we found in [27]: EAT data structures were characterized by means of a *final* object in a category. This fact suggests that our work is closer to the final semantics setting (final algebra semantics was defined by Wand [51] and, in some sense, even previously in [17]) than to the initial semantics one. In the same way it is reasonable to suppose that we will be able to express our results in terms of hidden specifications

([6], [20]) and also (and not in an independent way) within the coalgebraic framework ([39], [44], [25]). Recall that one of the goals of these two research domains is to reach the object-oriented systems for which the initial models are not convenient.

The third and last clue is related to the nature of the mathematical structures used in effective Algebraic Topology. Another explicit goal of the coalgebraic approach is to deal with infinite data structures, but this was also one of Sergeraert's objectives from the beginning of his research in effective Algebraic Topology [46]. Coalgebraic authors think in *potentially* infinite sequential structures (infinite lists, streams, generators and other mechanisms related to lazy evaluation strategies) and not in the representation of infinite algebraic or combinatorial structures (chain complexes, simplicial sets, simplicial groups, and so on) which are needed for symbolic computation in Algebraic Topology. However, it is very appealing to search a relationship between these two apparently distant topics.

The rest of the paper is mainly devoted to the second clue, to explaining, in an elementary way, this unexpected relationship and to suggesting the consequences it might have on symbolic computation, through the EAT system.

3 Algebraic Specification of the EAT Data Structures

In the EAT system [42], two different layers of data structures coexist. On one hand, there are data types based on symbol lists, integer arrays, and so on; these data structures can be specified in the usual way using algebraic specification and considering the initial semantics. On the other hand, there are data types corresponding to mathematical structures (such as chain complexes, simplicial sets, etc.). The instances of the former appear in EAT as *elements* of instances of the latter (this is the relation between *cmb* and *cc* in our previous discussion on chain complexes). In [27], we started the formal analysis of this second kind of data structures, showing how initial semantics is no longer useful and introducing an operation on ADTs which seems suitable for explaining the essence of these structures.

Although a group is a very elementary structure, simpler than the complex structures used in EAT, it will allow us to introduce the syntactical aspects of this operation on ADTs. Let us consider the signature `GRP` with only one sort g and three operations:

$$\begin{array}{lcccc} prd & : & g\ g & \rightarrow & g \\ inv & : & g & \rightarrow & g \\ unt & : & & \rightarrow & g \end{array}$$

This signature is obviously the basis for an algebraic specification of one group, whose underlying set is abstracted by the sort g. But when this is translated to a computer, something is missing. As Cook pointed out in [11]: "[another] type is

present, but invisible, within an ADT; this is the type of the concrete representation of the abstract type values". Now, for a specific implementation of an ADT, it can be considered that the sort g abstracts these concrete values (for instance, g could be the concrete type of Common Lisp symbols). But if, as usual in Symbolic Computation packages, several groups sharing the same concrete type have to be handled, an ingredient is still missing: the type of the groups represented on g is present though is invisible in the signature `GRP`. Making explicit this invisible (or *hidden*) type, we obtain a signature where the operations are:

$$\begin{array}{lcl} imp\_prd & : & imp\_g \;\; g \;\; g \;\; \rightarrow \;\; g \\ imp\_inv & : & imp\_g \;\; g \;\; \rightarrow \;\; g \\ imp\_unt & : & imp\_g \;\; \rightarrow \;\; g \end{array}$$

The prefix *imp* has been chosen because this new signature is not intended to specify *one* group, but to deal with *implementations* of groups. We can observe how the relationship between these two signatures is similar to the one which can be established between the two signatures for points presented in Section 2.

This construction is generalized for any signature in the following way. Let $\Sigma = (S, \Omega)$ be a signature, where $\Omega = (\omega_1, \ldots, \omega_m)$ is an enumeration of the operation symbols (each operation ω_j has associated an arity $s_j^1 \ldots s_j^{k_j} \rightarrow s_j^{k_j+1}$, where s_j^r belongs to S for each $r \in \{1, \ldots, k_j + 1\}$). A new signature $\Sigma_{Imp} = (S_{Imp}, \Omega_{Imp})$ is defined as follows:

- $S_{Imp} = \{imp\_s\} \cup S$, where imp_s is a fresh symbol.
- $\Omega_{Imp} = \{imp\_\omega_1, \ldots, imp\_\omega_m\}$ in which for each operation ω with arity $s_1 \ldots s_n \rightarrow s$ in Ω, an operation imp_ω with arity $imp\_s \; s_1 \ldots s_n \rightarrow s$ is included in Ω_{Imp}.

As usual, given a signature Σ, we denote by $Alg(\Sigma)$ the category of (total) Σ-algebras and Σ-homomorphisms. In this paper the categories $Alg(\Sigma_{Imp})$ are going to be interpreted from several points of view.

The first perspective we are going to consider is that related to real programming issues. This was our primary perspective, since we were interested in formally explaining the implementation strategies used in an *actual* software product: the EAT system. The construction $()_{Imp}$ allowed us to reach this goal, as shown in [27]. Nevertheless, the context in which [27] worked was more complex than the category $Alg(\Sigma_{Imp})$ in at least three aspects:

- The categories $Alg(\Sigma)$ are too big. An ADT is usually identified with a subcategory of $Alg(\Sigma)$ (closed by isomorphisms), the subcategory being explicitly defined by properties of the Σ-algebras which belong to it or by an axiomatic specification which determines a subclass of Σ-algebras (those that satisfy the axioms, called the *models* of the specification).
- In programming, we are frequently faced with *partiality* matters (because algorithms are rarely well-defined for each syntactically correct input) and then the interest focuses on *partial* Σ-algebras rather than on the *total* ones.

– Finally, $Alg(\Sigma)$ is related to the *specification* of an ADT, while in real programming we are interested in the *implementation* of ADTs. This is the reason why in [27] we relied on Hoare's classical notions [24], which are closely related to the actual programming matters.

The first two points can be tackled with little effort. The third item represents a more radical change which will be dealt with in Section 6. In the following sections we focus on the easier case of $Alg(\Sigma_{Imp})$, since our main objective is to clearly explain the relations between our approach and others stemming from the object-oriented approach. We will go back to implementation matters and to symbolic computation applications in Section 6.

In any case, we illustrate the programming meaning of Σ_{Imp} by means of examples based on the signature GRP previously introduced, and we assume that all the models in the examples will be groups and not only GRP-algebras. Let n be a natural number and for each example we consider the set $D = \{0, 1, \ldots, n-1\}$. Then we define a $\mathtt{GRP}_{Imp}$-algebra A with D as the carrier set A_g of sort g, and $A_{imp\_g} = \{(n_1, \ldots, n_k) \mid k \geq 1, n_i \in \mathbb{N}, n_i > 1, \forall i = 1, \ldots, k, \text{ and } n = n_1 * \ldots * n_k\}$. The idea is that a tuple $(n_1, \ldots, n_k)$ encodes the finite abelian group $\mathbb{Z}/n_1\mathbb{Z} \times \ldots \times \mathbb{Z}/n_k\mathbb{Z}$. Introducing a bijection between the elements of this group and the set D, it is straightforward to complete the definition of a $\mathtt{GRP}_{Imp}$-algebra. So, this algebra represents a family of groups. For instance, if $n = 12$, the tuples $(2, 2, 3)$, $(2, 6)$ and $(4, 3)$ represent three groups (the first two are isomorphic). It is obvious that any representation for abelian groups defined on D can be mapped on this $\mathtt{GRP}_{Imp}$-algebra A. And it is also obvious that the representations of non-abelian groups fall beyond the scope of this particular representation. Anyway, from a programming point of view, the most relevant aspect is that each implementation of the $\mathtt{GRP}_{Imp}$-algebra A gives us the means to work with every (abelian) group such that its underlying set of elements is (a representation of) D.

Then, a suitable formal setting for our work is the category, denoted by $Alg^D(\mathtt{GRP}_{Imp})$, whose objects are those $\mathtt{GRP}_{Imp}$-algebras such that their carrier set of sort g is D and whose morphisms are the identity map on D (remaining free the map in the carrier set of sort imp_g). The $\mathtt{GRP}_{Imp}$-algebra A belongs to $Alg^D(\mathtt{GRP}_{Imp})$.

To cover the non-abelian case, we will define another $\mathtt{GRP}_{Imp}$-algebra B with $B_g = D$ and $B_{imp\_g} = \{n \times n \text{ matrix} \mid \text{the matrix is the multiplication table of a group on } D\}$. Since D is finite, the multiplication table allows us to obtain the inverses and the unit, and in that way the definition of the $\mathtt{GRP}_{Imp}$-algebra B can be completed. It is clear that *any* representation of groups on D can be mapped on B and, if the underlying data set D remains unchanged, this $\mathtt{GRP}_{Imp}$-homomorphism is unique. So, B is the most "general" algebra among the representations of groups on D.

The problem of finding such a "most general" representation for any signature Σ and any domain D has an easy solution. We will see it in the following

section, where it will be nicely (almost trivially) described. To introduce the subject, let us observe that carrier sets such as A_{imp_g} and B_{imp_g} can be interpreted in a more "set-theoretical" way: instead of seeing an element of A_{imp_g} as a representation of a group, we can consider it as an index for a group, in such a way that a Σ_{Imp}-algebra is considered as an *indexed family* of Σ-algebras.

4 The Category Theory View

In order to formalize the interpretation of Σ_{Imp}-algebras as indexed families of Σ-algebras, we place ourselves in the more general setting of Category Theory. If $\mathcal{C}$ is a category, its object class will be denoted by $Obj(\mathcal{C})$, its morphism class by $Mor(\mathcal{C})$ and, if $A, B \in Obj(\mathcal{C})$, the morphisms from A to B will be denoted by $Mor_{\mathcal{C}}(A, B)$.

Given a category $\mathcal{C}$, we define a new category, denoted by $\mathcal{C}_{Set}$, to capture the notion of "indexed family of $\mathcal{C}$-objects". So, an object of $\mathcal{C}_{Set}$ is a pair (A, α_A), where A is a set and $\alpha_A{:}A \rightarrow Obj(\mathcal{C})$ is a mapping from A to $Obj(\mathcal{C})$. A morphism in $\mathcal{C}_{Set}$ between two objects $\alpha_A{:}A \rightarrow Obj(\mathcal{C})$ and $\alpha_B{:}B \rightarrow Obj(\mathcal{C})$ is a pair (h, H), where $h{:}A \rightarrow B$ is a map and $H{:}A \rightarrow Mor(\mathcal{C})$ is such that $H(a) \in Mor_{\mathcal{C}}(\alpha_A(a), \alpha_B(h(a)))$, $\forall a \in A$. These categories $\mathcal{C}_{Set}$ are particular cases of indexed or fibered categories [50], as explained below.

Note that if we consider a set A as a discrete category and a map $\alpha_A{:}A \rightarrow Obj(\mathcal{C})$ as a functor from A to $\mathcal{C}$, then the map H can be understood as a natural transformation from the functor α_A to the functor $\alpha_B \circ h$. In this context, the category $\mathcal{C}_{Set}$ can be defined in a more formal way as the *flattened category* [50] obtained from the indexed category $\mathcal{C}{:}Set^{op} \rightarrow CAT$ which associates to each set A the category $\mathcal{C}^A$ of functors from A to $\mathcal{C}$ (where Set denotes the category of sets and maps, and CAT the category of categories and functors). It is very easy to prove that $\mathcal{C}_{Set}$ is endowed with *general* coproducts, which are induced by the disjoint union of the index sets. For finite coproducts, if α_A and α_B are two objects in $\mathcal{C}_{Set}$, their coproduct is $[\alpha_A, \alpha_B]{:}A \sqcup B \rightarrow Obj(\mathcal{C})$, where the map is defined in an obvious way.

There is a canonical embedding of $\mathcal{C}$ into $\mathcal{C}_{Set}$. It is enough to define a functor $F{:}\mathcal{C} \rightarrow \mathcal{C}_{Set}$ in such a way that each $\mathcal{C}$-object is indexed by the only element of a singleton. This functor will be called *embedding functor* of $\mathcal{C}$ in $\mathcal{C}_{Set}$.

Now, if $\mathcal{C}$ is a small category (and therefore the *class* $Obj(\mathcal{C})$ is a *set*), there is a very singular object in $\mathcal{C}_{Set}$, namely, the identity $\mathbb{1} : Obj(\mathcal{C}) \rightarrow Obj(\mathcal{C})$. In a sense, $\mathbb{1}$ represents the category $\mathcal{C}$ itself. This modest (even trivial) object turns out to be a close relative of all the final objects and families of final objects which appeared in the recent years in the field of the object-oriented algebraic specifications (see [6], [20], [25], [39], [9]). The object $\mathbb{1}$ is characterized in Category Theory by the following evident property.

Theorem 1. *If $\mathcal{C}$ is a small category, the object $\mathbb{1}$ is the coproduct in $\mathcal{C}_{Set}$ of the objects in the image of the embedding functor of $\mathcal{C}$ in $\mathcal{C}_{Set}$. In addition, there is at least one morphism from each object in $\mathcal{C}_{Set}$ to $\mathbb{1}$.*

The second part of this result seems clearly linked to the notion of final object. But, in general, the distinguished morphism from an object α_A to $\mathbb{1}$ is not unique: the degree of freedom is related to the definition of the map between the sets of indexes (it can be different from α_A) and to the morphisms in $Mor_{\mathcal{C}}(\alpha_A(a), \alpha_A(a))$ for each $a \in A$.

Let us go back to signatures and algebras. Let $\Sigma = (S, \Omega)$ be a signature, being S the set of sorts and Ω the set of operations of Σ. We consider the corresponding categories $Alg(\Sigma_{Imp})$ and $Alg(\Sigma)_{Set}$. It is clear that a functor $I{:}Alg(\Sigma_{Imp}) \to Alg(\Sigma)_{Set}$ can be defined, fulfilling the intuitive idea of interpreting a Σ_{Imp}-algebra as an indexed family of Σ-algebras. To be precise, for each Σ_{Imp}-algebra $A = (A_{imp\_s}, (A_s)_{s\in S}, (imp\_\omega_A)_{\omega\in\Omega})$, the family of Σ-algebras $I(A) = (A_{imp\_s}, \alpha)$ is defined as follows. For each $x \in A_{imp\_s}$, $\alpha(x) = ((A_s)_{s\in S}, (\omega_{A_{(x)}})_{\omega\in\Omega})$, where $\omega_{A_{(x)}}(d_1, \ldots, d_n) = imp\_\omega_A(x, d_1, \ldots, d_n)$. Observe that this functor is not surjective on objects (for a given Σ_{Imp}-algebra A, all the Σ-algebras of the family $I(A)$ have the same carrier sets, namely $(A_s)_{s\in S}$, and this is not true for each family in $Alg(\Sigma)_{Set}$).

Now, in order to formalize the ideas of Section 3, all the elements of the structures must belong to a same set, and this implies that some subcategories of $Alg(\Sigma)_{Set}$ and $Alg(\Sigma_{Imp})$ have to be considered. We fix an S-set $D = (D_s)_{s\in S}$ and define a subcategory $Alg^D(\Sigma_{Imp})$ of $Alg(\Sigma_{Imp})$. The objects of $Alg^D(\Sigma_{Imp})$ are the Σ_{Imp}-algebras A such that $A_s = D_s$ for each sort s different from imp_s (the carrier set A_{imp_s} remains free). Its morphisms are those Σ_{Imp}-morphisms such that they are identities on D (the map between the carrier sets of sort imp_s remains free). In Section 3 we have shown the computational interest of this kind of categories, and now we are going to look for an adequate corresponding subcategory of $Alg(\Sigma)_{Set}$.

Given a category $\mathcal{C}$, we denote by $\mathcal{C}^{\{\}}$ the category with the same objects as $\mathcal{C}$ and whose morphisms are only the identities. Note that the category $\mathcal{C}^{\{\}}$ is nothing but a class of objects (and when $\mathcal{C}$ is a small category, it is just a set). This seems to take meaning away from this category (this explains the notation $\{\}$), but this radical situation appeared in a natural way in our approach [27] and also in the field of the hidden specifications (see, for example, [20]). The underlying idea is that EAT (and other Symbolic Computation systems) does not need to represent a category of mathematical structures, for instance the category of groups, but rather the *class* of groups and, sometimes, the *class* of group homomorphisms. This is exactly the situation in EAT which provides the user two kinds of data structures, one for the objects and another one for the morphisms. We will use $\mathcal{C}^{\{\}}_{Set}$ to denote the category $(\mathcal{C}^{\{\}})_{Set}$. Let us note that $(\mathcal{C}^{\{\}})_{Set}$ is different from $(\mathcal{C}_{Set})^{\{\}}$, since the morphisms in $(\mathcal{C}^{\{\}})_{Set}$ from (A, α_A) to (B, α_B) are maps $h{:}A \to B$ such that $\alpha_A = \alpha_B \circ h$. From now on we will

think of $\mathcal{C}^{\{\}}_{Set}$ as a subcategory of $\mathcal{C}_{Set}$. Then, we trivially obtain the following consequence (that, besides, is a well-known fact, since $\mathcal{C}^{\{\}}_{Set}$ is nothing but the slice category $Set/Obj(\mathcal{C})$).

Corollary 1. *If $\mathcal{C}$ is a small category, the object $\mathbb{1}$ is final in $\mathcal{C}^{\{\}}_{Set}$.*

Now, given a signature $\Sigma = (S, \Omega)$ and fixed an S-set $D = (D_s)_{s \in S}$, we consider the category, denoted by $Alg^D(\Sigma)$, whose objects are those Σ-algebras A such that $A_s = D_s$, for each sort $s \in S$, and whose morphisms are only the identities. Then, we have the corresponding category $Alg^D(\Sigma)_{Set}$. This category can be considered as a subcategory of $Alg(\Sigma)_{Set}$ and then it is clear that, when restricting the functor I to the subcategory $Alg^D(\Sigma_{Imp})$, we obtain an isomorphism between this category and $Alg^D(\Sigma)_{Set}$.

We denote by $\mathbb{1}^D$ the object of $Alg^D(\Sigma_{Imp})$ which corresponds to the final object $\mathbb{1}$ of $Alg^D(\Sigma)_{Set}$. Hence, the carrier set for the sort imp_s in $\mathbb{1}^D$ is given by the set of Σ-algebras whose S-set (its carrier sets) is D. The interpretation of an operation imp_ω in $\mathbb{1}^D$ is given by: $imp\_\omega_{\mathbb{1}^D}(A, d_1, \ldots, d_n) := \omega_A(d_1, \ldots, d_n)$, being $s_1 \ldots s_n \to s$ the arity of $\omega \in \Omega$, and being ω_A the interpretation of the operation ω in A.

Moreover, there is a canonical embedding of the category $Alg^D(\Sigma)$ into $Alg^D(\Sigma_{Imp})$. Each Σ-algebra can be interpreted as a Σ_{Imp}-algebra (it suffices to consider the singleton $\{*\}$ as carrier set of sort imp_s). The next result obviously follows.

Theorem 2. *The object $\mathbb{1}^D$ is the coproduct in $Alg^D(\Sigma_{Imp})$ of the objects in the image of the embedding functor of $Alg^D(\Sigma)$ in $Alg^D(\Sigma_{Imp})$. In addition, the object $\mathbb{1}^D$ is final in the category $Alg^D(\Sigma_{Imp})$.*

This final object $\mathbb{1}^D$ is isomorphic, in certain particular cases (immutable objects), to the final object which appears in the hidden specification context, and this is fairly simpler and more intuitive. But before comparing these two objects, two final remarks must be made.

First, in $Alg^D(\Sigma_{Imp})$, there is an object isomorphic to $\mathbb{1}^D$ which can be presented in a more practical way. Since D is fixed, giving a Σ-algebra is equivalent to giving the functions realizing each operation of Σ. So, the carrier set of sort imp_s in the object isomorphic to $\mathbb{1}^D$ can be described as the set of tuples of functions realizing the operations in a Σ-algebra. For instance, in the case of the signature GRP, the carrier set for imp_g would be $\{(f_1, f_2, f_3) \mid f_1 : D \times D \to D,\ f_2 : D \to D,\ f_3 : \{*\} \to D\}$. The relation between these tuples of functions and the records of functional objects (and therefore with the final object) described in [27], is clear. The resemblance to the models introduced by Cardelli and other authors ([1], [8]) to study theoretical aspects in object-oriented programming is also very appealing. Moreover, if we restrict ourselves

to the case of the *groups* (and not general GRP-algebras), the final object can be described through the tuples (f_1, f_2, f_3) satisfying the group axioms. Then, if we go back to the example where $D = \{0, 1, \ldots, n-1\}$, it is quite clear that this final object is isomorphic to the Σ_{Imp}-algebra B introduced in the previous section.

Second, even in the larger subcategory $Alg^{\{\}}(\Sigma_{Imp})$ of $Alg(\Sigma_{Imp})$ (which is obtained by considering only morphisms that are the identities on the carrier sets for the sorts different from imp_s) the class of objects $\mathbb{1}^D$ can be characterized by a universal property, namely by a *multi-colimit* of a particular kind. The notion of multi-colimit was introduced by Diers in [12] and was used in the context of Algebraic Specifications by Cîrstea [10]. (See these papers for detailed definitions.) For our purpose it is enough to know that, given a category $\mathcal{C}$, a *final family in* $\mathcal{C}$ is a class $\{O_j\}_{j\in J}$ of objects of $\mathcal{C}$ such that for any object A of $\mathcal{C}$, there exists only one $j \in J$ such that there is a morphism from A to O_j and, in addition, this morphism is unique. To guarantee that $Alg(\Sigma)$ is a small category, we are going to consider only Σ-algebras such that their carrier sets are subsets of a fixed set U (the *universe of data*). This constraint, which is very natural in a programming context, will not be explicitly pointed out in the statements. Therefore, we can index the family $\{\mathbb{1}^D\}$ on *all* S-sets D. In a slightly inappropriate way (since D is an S-set and not a set), we denote this family by $\{\mathbb{1}^D\}_{D\in\mathcal{P}(U)}$. Under these conditions and notations, the following result is obtained.

Theorem 3. *The family* $\{\mathbb{1}^D\}_{D\in\mathcal{P}(U)}$ *is final in the category* $Alg^{\{\}}(\Sigma_{Imp})$.

5 The Hidden Specifications View

Hidden specification is a special case of behavioural or observational specifications which were introduced by Reichel [38] and further studied by other authors [45], [35], [33], [23], [21]. The terminology *hidden specification* appeared in [18] for the first time and its theory has been extensively developed in recent years (see [19], [20], [6], [9]). Although the definitions vary slightly from one paper to another, we present a version which seems to summarize the essence of the notions.

Let $V\Sigma = (VS, V\Omega)$ be a signature. The elements of VS are called *visible sorts* and those of $V\Omega$ *visible operations*. Let us fix a $V\Sigma$-algebra D and let us include in $V\Omega$, as constants, the elements of the carrier sets of D which do not correspond to constants present in $V\Omega$. The $V\Sigma$-algebra D is called *data domain*. Then a *hidden signature* on $V\Sigma$ and D is a signature $H\Sigma = (S, \Omega)$ such that:

- $S = HS \sqcup VS$; the elements of HS are called *hidden sorts* of $H\Sigma$.
- $\Omega = H\Omega \sqcup V\Omega$, and for each operation $\omega : s_1 \ldots s_n \to s$ in $H\Omega$, the following two properties hold:

- there is at least one hidden sort in $s_1, \ldots, s_n, s$.
- if there is a hidden sort in $s_1, \ldots, s_n$, it is unique (it is assumed, by convention, that the hidden sort appears in the first position).

This last constraint is roughly equivalent to working with the "message passing" model instead of the "generic function" model for object-oriented programming (see [22]). The idea underlying this definition is that a hidden sort is used to name a set of objects, while a visible sort names a set of data.

The operations in $H\Omega$ of a hidden signature are classified in two classes:

- the operations $\omega : s_1 \ \ldots \ s_n \rightarrow s$, such that s is a hidden sort, are called *constructors*;
- the rest (that is, operations ending in a visible sort) are called *deconstructors*.

In turn, constructors are classified into:

- constructors *from data* when $s_1, \ldots, s_n$ are visible sorts;
- constructors *from objects* (and data) when s_1 is different from s (both, of course, being hidden sorts); and
- *updating operations* when $s_1 = s$.

These last operations correspond to the functional way of representing the updating of the *local state* of an object, as pointed out in Section 2. In the example of the *points* in that section, there is only one hidden sort p, only one visible sort int, the data domain is implicitly assumed to be $\mathbb{Z}$, and there is one constructor from data ($cons$), one updating operation ($move$) and two deconstructors ($first$ and $second$).

Remark (on terminology): In the area of hidden specifications, the constructors from data are called *generalized hidden constants*, the rest of the constructors are called *methods*, and the deconstructors are called *attributes*. The deconstructors correspond to the *observers* in the classical theory of algebraic specifications [16], if every hidden sort is declared a *distinguished sort*. In these operations, our terminology is a variant of Reichel's terminology [39] who called them *destructors*.

A *hidden algebra* A for a given hidden signature $H\Sigma$, on $V\Sigma$ and D, is a $H\Sigma$-algebra such that $A_{V\Sigma} = D$ (in other words, the reduct of A to the visible part is equal to the data domain D). A *hidden morphism* between two hidden algebras is a $H\Sigma$-homomorphism f such that f_D is the identity on D.

Hidden algebras for $H\Sigma$, on $V\Sigma$ and D, and hidden morphisms define a category, which will be denoted by $HAlg^D(H\Sigma)$. The following result can be found in [20].

Theorem 4. *If $H\Sigma$ is a hidden signature without constructors from data, then the category $HAlg^D(H\Sigma)$ has a final object.*

In [20] an explicit description of the final object is given and its carrier sets are defined by so-called "magical formula" (see [20]). This object will be introduced below, but first we will go back to the Σ_{Imp}-algebras.

It is clear that a signature Σ_{Imp} can be considered a hidden signature which satisfies the following properties:

- it only has one hidden sort (the sort imp_s);
- it has no visible operations;
- each operation is a deconstructor.

Besides, each hidden signature H where the three properties we have just introduced hold, is isomorphic (i.e., equal up to renaming) to a signature Σ_{Imp} for a suitable signature Σ. This signature Σ is obtained by erasing the hidden sort wherever it occurs in H. This way of obtaining Σ from H corresponds to the definition of an *interface for objects*, as pointed out by Cook in [11], page 169. (This is also the operation we have implicitly applied in Section 2 between the two signatures for points.)

Now, one can wonder what is the interest of this kind of hidden signatures since they have no constructors. However, note that the same objection could be raised regarding the hidden signatures for which the last theorem applies, since without constructors from data, we are not able to generate any object from scratch. The underlying idea is that the constructors from data are very special methods which must be tackled separately. This point of view seems to be well-established in every object-oriented programming language. In our particular case, there are no updating operations because we are interested in a *pure* functional programming approach, and constructors from objects are considered part of the system as a whole, but they are considered as algorithms and processes outside any particular algebraic structure. For instance, in EAT [42], there is an operator constructing (the implementation of) a chain complex from (the implementation of) a simplicial set. Would such an operator belong to the simplicial sets signature, or rather to the chain complexes signature? Discarding the possibility of specifying the complete system by means of a huge signature, the option of including only operations handling *elements* of a concrete structure has been chosen. In other words, our signatures gather only the *behaviour* of the implementations, considering every kind of constructors as *external* to any signature. Needless to say that to complete the system modelling, we should integrate standard signatures and signatures Σ_{Imp}, through constructors, but it is also clear that this is a different (and subsequent) task.

In any case, the relevant remark is that the category $HAlg^D(\Sigma_{Imp})$ is isomorphic to the category $Alg^D(\Sigma_{Imp})$ which has been defined in Section 4 and, therefore, that the final object in the previous theorem is isomorphic to $\mathbb{1}^D$. Note that $\mathbb{1}^D$ has no "magical" aspect and looks rather natural (but we must stress again: this only applies to a very particular case). It could be interesting

to establish an explicit isomorphism between these two final objects. In order to do it, some auxiliary definitions are needed (extracted from [20]).

Given a hidden signature $H\Sigma$ and a hidden sort h, a *$H\Sigma$-context of sort h* is a visible sorted $H\Sigma$-term (in other words, it is an expression where the last operation applied is a deconstructor) having a single occurrence of a variable, this variable z being of sort h.

The intuitive idea is the following. In the absence of constructors from data, the variable z (the name of this variable is not relevant, of course) is used as the starting point for all the possible observations of the "local state" of an object of sort h. Even without any more variables, there are enough $H\Sigma$-contexts thanks to the technical condition that makes the elements of D to be considered constants in the hidden signature (in the hidden specification framework, semantics is based on behavioural satisfaction; that is, two terms with the same visible behaviour are considered as equivalent). The set of $H\Sigma$-contexts of sort h is denoted by $\mathcal{C}_{H\Sigma}[z_h]$ and can be considered as an VS-set since it is stratified by the sort of the deconstructor applied (remember that VS stands for the visible sorts of $H\Sigma$).

Then, the so-called "magical formula" providing the carrier set for a hidden sort h in the final object $F_{H\Sigma}$ is given by:

$$F_{H\Sigma,h} = \prod_{v \in VS} [\mathcal{C}_{H\Sigma}[z_h]_v \to D_v]$$

where, as usual, we write $[A \to B]$ for the set of maps from A to B.

Instead of further explaining these formulas (which are described in [20] as "a kind of continuation"), we focus on our particular case of hidden signatures, looking for an explicit comparison with our final object $\mathbb{1}^D$. First, note that if there is no constructor, the $H\Sigma$-contexts are very simple: they are expressions $\omega(z, d_2, \ldots, d_n)$, for each operation $\omega : h\ v_2\ \ldots\ v_n \to v$, where $d_i \in D_{v_i}$, for $i = 2, \ldots, n$. (We are relying here again on the condition that the elements of D_v are constants in the signature; this strong condition implies that, for the sake of expressiveness, the operations of the algebra D can be skipped.) Thus, in the case of the signatures Σ_{Imp} in which, in addition, there is only one hidden sort, the set $\mathcal{C}_{\Sigma_{Imp}}[z_{imp\_s}]_v$ can be seen as a sum of sets (one set for each operation ω of sort v) and then a map from $\mathcal{C}_{\Sigma_{Imp}}[z_{imp\_s}]_v$ to D_v can be unfolded as a tuple of maps corresponding to the v-sorted functions in $\mathbb{1}^D$ (here we are referring to the "practical presentation" of $\mathbb{1}^D$ introduced in Section 4). When the symbol v ranges over the visible sorts, it is clear that every function in $\mathbb{1}^D$ is reached. This bijection easily extends to a hidden Σ_{Imp}-isomorphism between the "magical final object" and $\mathbb{1}^D$. For example, in the case of GRP, there is only one visible sort g, and each context $imp\_prd(imp\_g, d_1, d_2)$ contributes to the definition of $f_1{:}D \times D \to D$, each context $imp\_inv(imp\_g, d)$ to $f_2{:}D \to D$ and the context $imp\_unt(imp\_g)$ to $f_3{:}\{*\} \to D$.

It is well-known that hidden algebras are closely related to coalgebras. Hence, it is reasonable to suppose that we will be able to express the categories $Alg^D(\Sigma_{Imp})$ as categories of coalgebras.

The coalgebraic approach provides a general framework to study object-oriented features (in a very close way to hidden specifications) and to specify infinite data structures. The original ideas appeared in [39], and they were developed in [44], [25], among others.

Let $F{:}Set \to Set$ be an endofunctor of Set, the category of sets. Then a *F-coalgebra* is a pair (A, α_A), where A is a set and $\alpha_A{:}A \to F(A)$ is a map. A *morphism between F-coalgebras* (A, α_A), (B, α_B) is a map $f{:}A \to B$ such that $F(f) \circ \alpha_A = \alpha_B \circ f$.

For a given endofunctor F, F-coalgebras and morphisms between F-coalgebras form a category, which is denoted by $CoAlg(F)$. Given a fixed set A, a simple example of a category of coalgebras is the abovementioned *slice category* Set/A. The category Set/A corresponds to the functor $F_A{:}Set \to Set$ constant onto A (a functor $F{:}Set \to Set$ is said to be *constant onto a set* A if for each set X, $F(X) = A$, and for each map f, $F(f)$ is the identity map 1_A, that is, the image of a constant functor is the smallest non-empty category).

Now, we are going to briefly explain the relations between the hidden specifications and the coalgebraic perspective, which were studied in [30] and [9], for example. Let $H\Sigma = (S, \Omega)$ be a hidden signature on $V\Sigma = (VS, V\Omega)$ and data domain D, without constructors from data and without visible operations (that is, $V\Omega = \emptyset$ and thus $\Omega = H\Omega$). To make the notation easier, let us focus on the particular case where $H\Sigma$ only has one hidden sort h. Under these conditions, we consider an enumeration of $\Omega = (\omega_1, \ldots, \omega_k, \omega_{k+1}, \ldots \omega_m)$ where ω_i are deconstructors for $i = 1, \ldots, k$ and ω_i are updating operations for $i = k+1, \ldots, m$. In addition, $\omega_i : h\ v_2\ \ldots\ v_{n_i} \to v$ for $i = 1, \ldots, k$ and $\omega_i : h\ v_2\ \ldots\ v_{n_i} \to h$ for $i = k+1, \ldots, m$, where the symbol v refers to *visible* sorts. Let us denote by α_i the set of visible sorts $\{v_2, \ldots, v_{n_i}\}$ corresponding to the operation $\omega_i, \forall i = 1, \ldots, m$. Then we write $D^{\alpha_i} = D_{v_2} \times \cdots \times D_{v_{n_i}}, \forall i = 1, \ldots, m$. Now, we are ready to define an endofunctor $F_{H\Sigma}{:}Set \to Set$ from such a hidden signature $H\Sigma$. Given a set X, we define $F_{H\Sigma}(X) = \{(f_1, \ldots, f_k, f_{k+1}, \ldots, f_m) \mid f_i{:}D^{\alpha_i} \to D_v, \forall i = 1, \ldots, k \text{ and } f_i{:}D^{\alpha_i} \to X, \forall i = k+1, \ldots, m\}$ and, given a map $g{:}X \to Y$, we define $(F_{H\Sigma}(g))((f_1, \ldots, f_k, f_{k+1}, \ldots, f_m)) = (f_1, \ldots, f_k, g \circ f_{k+1}, \ldots, g \circ f_m), \forall (f_1, \ldots, f_k, f_{k+1}, \ldots, f_m) \in F_{H\Sigma}(X)$. It is not difficult to prove that the category $CoAlg(F_{H\Sigma})$ is isomorphic to the hidden category $HAlg^D(H\Sigma)$.

Our case, that is to say the case of Σ_{Imp}-algebras, is very close to the hidden signatures $H\Sigma$ above. Such a hidden signature, but *without updating operations*, can be considered as Σ_{Imp} for a convenient signature Σ, namely its interface of objects. Moreover, if we consider a hidden signature Σ_{Imp}, since it has no updating operations, the functor $F_{\Sigma_{Imp}}$ is the constant functor onto the

set F of functional tuples $F = [D^{\alpha_1} \to D_{v_1}] \times \cdots \times [D^{\alpha_m} \to D_{v_m}]$. Hence, an object of $Alg^D(\Sigma_{Imp})$ can be described as a $F_{\Sigma_{Imp}}$-coalgebra, that is, a map from a set X into the set of tuples of functions which have the arity of the operations of Σ (each one of these tuples of functions corresponds to the operations of a Σ-algebra with carrier sets D).

The section finishes with a remark about the existence of final object in $CoAlg(F_{\Sigma_{Imp}})$. First, the following result, due to Barr [4] (and which has a constructive proof), implies that the category $CoAlg(F_{\Sigma_{Imp}})$ has a final object.

Theorem 5. *If F is a polynomial functor, then the category $CoAlg(F)$ has a final object.*

The definition of polynomial functor can be found, for instance, in [25]. For our purpose, it suffices to know that a constant functor is a polynomial functor.

Second, we have shown that there is an isomorphism between $HAlg^D(\Sigma_{Imp})$ and $CoAlg(F_{\Sigma_{Imp}})$. Then, we can consider in $CoAlg(F_{\Sigma_{Imp}})$ the object which corresponds to the final object $\mathbb{1}^D$ in $HAlg^D(\Sigma_{Imp})$. Finally, since $F_{\Sigma_{Imp}}$ is a constant functor, its respective category of coalgebras is the slice category Set/F and then, the final object is given by the identity map $1_F{:}F \to F$ (this gives an expression which corresponds exactly to the "practical presentation" for the final hidden algebra which has been introduced in this section).

To sum up, it has been seen that the categories $Alg^D(\Sigma_{Imp})$, $HAlg^D(\Sigma_{Imp})$ and $CoAlg(F_{\Sigma_{Imp}})$ are the same. These isomorphisms allow us to use different presentations for the final object of these categories. The "practical" presentation, that is, the coalgebraic one, is the most appropriate for the implementation perspective.

6 Applications to Symbolic Computation

In the previous sections we have seen that the EAT constructions and representations are closely related to object-oriented programming (and also, and not in an independent way, to functional programming). This claim sets a framework to continue the analysis of the EAT data structures.

But, besides, the previous discussion allows us to extend the results given in [27] on the EAT implementations of mathematical structures. We must emphasize that the main goal of that work was to formalize the implementation pattern used in EAT to deal with mathematical structures, since we were convinced that this pattern had suitable properties from a programming point of view and could probably be used in other Symbolic Computation systems with similar requirements. So, the starting point was an actual software system and the analysis of its data structures led us to different formal frameworks. The underlying idea

is that in the implementation of a Symbolic Computation system, algebraic structures are data which are accessed through references, which can be considered as indexes (or as values of a hidden sort for a suitable kind of hidden signatures). Now, once the appropriate theoretical ideas have been developed, we can describe the implementation issues studied in [27] in a better and more general way.

The important fact is to realize that in order to deal with data which are mathematical structures, a software system has to be able to handle classes (families) of implementations of these structures and that these classes of implementations are given by objects of appropriate categories $Alg^{D}(\Sigma_{Imp})$.

Remember that the category where the EAT implementation pattern was characterized was denoted in [27] by $Imp^{\underline{\mathtt{T}},\underline{\mathtt{S}}}(\mathcal{T}_{Imp})$. In order to introduce this category some preliminary definitions are needed. The key notion is that of *implementation of an algebra*. An implementation has two basic parts: the representations of the carrier sets of the algebra and the programs which implement the operations. Now, we are going to briefly describe these notions. A detailed development of these issues can be found in [36]. (We suppose the reader is familiar with the elementary Common Lisp terminology.)

A (Common Lisp) *program* is a tuple $\mathtt{p} = (\mathtt{c}^{\mathtt{p}}, \mathtt{S}^{\mathtt{p}}, \mathtt{T}_1^{\mathtt{p}}, \ldots, \mathtt{T}_{\mathtt{n}}^{\mathtt{p}}, \mathtt{T}^{\mathtt{p}})$ where: $\mathtt{T}_1^{\mathtt{p}}, \ldots, \mathtt{T}_{\mathtt{n}}^{\mathtt{p}}, \mathtt{T}^{\mathtt{p}}$ are (Common Lisp) concrete types (a *concrete type* is a decidable set of Common Lisp objects together with an equality); $\mathtt{S}^{\mathtt{p}}$ is a subset of $\mathtt{T}_1^{\mathtt{p}} \times \cdots \times \mathtt{T}_{\mathtt{n}}^{\mathtt{p}}$ (called *definition domain of the program*); $\mathtt{c}^{\mathtt{p}}$ is a functional object (called *program code*) which can be applied to the data types $\mathtt{T}_1^{\mathtt{p}}, \ldots, \mathtt{T}_{\mathtt{n}}^{\mathtt{p}}$ (this means that, given $(\mathtt{d}_1, \ldots, \mathtt{d}_n) \in \mathtt{T}_1^{\mathtt{p}} \times \cdots \times \mathtt{T}_{\mathtt{n}}^{\mathtt{p}}$, the evaluation of the expression `(funcall` $\mathtt{c}^{\mathtt{p}}$ $\mathtt{d}_1 \ldots \mathtt{d}_n$`)` does not signal an error either "wrong number" or "wrong type" of arguments), and the evaluation of `(funcall` $\mathtt{c}^{\mathtt{p}}$ $\mathtt{d}_1 \ldots \mathtt{d}_n$`)` finishes for all $(\mathtt{d}_1, \ldots, \mathtt{d}_n) \in \mathtt{S}^{\mathtt{p}}$ returning an element of $\mathtt{T}^{\mathtt{p}}$. This element will be denoted by $\mathtt{c}^{\mathtt{p}}(\mathtt{d}_1, \ldots, \mathtt{d}_n)$. In addition, $\mathtt{c}^{\mathtt{p}}$ must preserve the concrete types equalities.

A program $\mathtt{p} = (\mathtt{c}^{\mathtt{p}}, \mathtt{S}^{\mathtt{p}}, \mathtt{T}_1^{\mathtt{p}}, \ldots, \mathtt{T}_{\mathtt{n}}^{\mathtt{p}}, \mathtt{T}^{\mathtt{p}})$ defines a (partial) function, which will be denoted by $F(\mathtt{p})$, $F(\mathtt{p}){:}\mathtt{T}_1^{\mathtt{p}} \times \cdots \times \mathtt{T}_{\mathtt{n}}^{\mathtt{p}} \to \mathtt{T}^{\mathtt{p}}$, whose definition domain is $\mathtt{S}^{\mathtt{p}}$ and which is defined by: $F(\mathtt{p})(\mathtt{s}) := \mathtt{c}^{\mathtt{p}}(\mathtt{s})$, for each $\mathtt{s} \in \mathtt{S}^{\mathtt{p}}$.

Given a set $\mathcal{M}$, a *representation of* $\mathcal{M}$ is a tuple $\mathcal{R} = (\mathcal{D}_{\mathcal{R}}, \mathcal{S}_{\mathcal{R}}, =_{\mathcal{R}}, \alpha_{\mathcal{R}})$ where: $\mathcal{D}_{\mathcal{R}}$ is a concrete type called *representation domain*; $\mathcal{S}_{\mathcal{R}}$ is a subset of $\mathcal{D}_{\mathcal{R}}$ called *representation support*; $=_{\mathcal{R}}$ is an equivalence relation on $\mathcal{S}_{\mathcal{R}}$ (greater than the equivalence relation on $\mathcal{S}_{\mathcal{R}}$ inherited from the domain $\mathcal{D}_{\mathcal{R}}$) called *representation equality*; and, finally, $\alpha_{\mathcal{R}}{:}\,\mathcal{D}_{\mathcal{R}} \to \mathcal{M}$ is a (partial) function whose definition domain is $\mathcal{S}_{\mathcal{R}}$, called *abstraction function*, compatible with $=_{\mathcal{R}}$ (that is, if $\mathtt{t}_1 =_{\mathcal{R}} \mathtt{t}_2$ then $\alpha_{\mathcal{R}}(\mathtt{t}_1) = \alpha_{\mathcal{R}}(\mathtt{t}_2)$, for all $\mathtt{t}_1, \mathtt{t}_2 \in \mathcal{S}_{\mathcal{R}}$).

Now, let $\mathtt{p} = (\mathtt{c}^{\mathtt{p}}, \mathtt{S}^{\mathtt{p}}, \mathtt{T}_1^{\mathtt{p}}, \ldots, \mathtt{T}_{\mathtt{n}}^{\mathtt{p}}, \mathtt{T}^{\mathtt{p}})$ be a program and let $\underline{\mathcal{R}} = (\mathcal{R}_1, \ldots, \mathcal{R}_n, \mathcal{R})$ be $n + 1$ representations of $\mathcal{M}_1, \ldots, \mathcal{M}_n, \mathcal{M}$ respectively, such that the representation domains $\mathcal{D}_1, \ldots, \mathcal{D}_n, \mathcal{D}$ are respectively $\mathtt{T}_1^{\mathtt{p}}, \ldots, \mathtt{T}_{\mathtt{n}}^{\mathtt{p}}, \mathtt{T}^{\mathtt{p}}$. The program $\mathtt{p}$ is said to be *coherent with respect to* $\underline{\mathcal{R}}$ if the

following conditions hold: *i)* $\mathtt{S^p} \subseteq \mathcal{S}_{\mathcal{R}_1} \times \cdots \times \mathcal{S}_{\mathcal{R}_n}$; *ii)* $F(\mathtt{p})(\mathtt{S^p}) \subseteq \mathcal{S}_{\mathcal{R}}$; *iii)* $\mathtt{p}$ is compatible with the representation equalities (that is, if $(\mathtt{d_1}, \ldots, \mathtt{d_n})$, $(\mathtt{d'_1}, \ldots, \mathtt{d'_n}) \in \mathtt{S^p}$ are such that $\mathtt{d_i} =_{\mathcal{R}_i} \mathtt{d'_i}$ for all $i = 1, \ldots, n$, then $\mathtt{F(p)}(\mathtt{d_1}, \ldots, \mathtt{d_n}) =_{\mathcal{R}} \mathtt{F(p)}(\mathtt{d'_1}, \ldots, \mathtt{d'_n})$); *iv)* the program $\mathtt{p}$ is compatible with the abstraction equalities (that is, if $(\mathtt{d_1}, \ldots, \mathtt{d_n})$, $(\mathtt{d'_1}, \ldots, \mathtt{d'_n}) \in \mathtt{S^p}$ are such that $\alpha_{\mathcal{R}_i}(\mathtt{d_i}) = \alpha_{\mathcal{R}_i}(\mathtt{d'_i})$ for all $i = 1, \ldots, n$, then $\alpha_{\mathcal{R}}(\mathtt{F(p)}(\mathtt{d_1}, \ldots, \mathtt{d_n})) = \alpha_{\mathcal{R}}(\mathtt{F(p)}(\mathtt{d'_1}, \ldots, \mathtt{d'_n}))$).

The basic idea is that if $\mathtt{p}$ is coherent with respect to $\underline{\mathcal{R}}$ (being $\mathcal{R}_i$ a representation of $\mathcal{M}_{\mathcal{R}_i}$ for all $i = 1, \ldots, n$ and being $\mathcal{R}$ a representation of $\mathcal{M}_{\mathcal{R}}$), then the partial function $F(\mathtt{p})_{\underline{\mathcal{R}}} : \mathcal{M}_{\mathcal{R}_1} \times \cdots \times \mathcal{M}_{\mathcal{R}_n} \to \mathcal{M}_{\mathcal{R}}$, whose definition domain is $Def(F(\mathtt{p})_{\underline{\mathcal{R}}}) = \{(m_1, \ldots, m_n) \in \mathcal{M}_{\mathcal{R}_1} \times \cdots \times \mathcal{M}_{\mathcal{R}_n} : \exists(\mathtt{d_1}, \ldots, \mathtt{d_n}) \in \mathtt{S^p}$ such that $\alpha_{\mathcal{R}_i}(\mathtt{d_i}) = m_i \quad \forall i = 1, \ldots, n\}$ and which is given by $F(\mathtt{p})_{\underline{\mathcal{R}}}(\alpha_{\mathcal{R}_1}(\mathtt{d_1}), \ldots, \alpha_{\mathcal{R}_n}(\mathtt{d_n})) := \alpha_{\mathcal{R}}(\mathtt{F(p)}(\mathtt{d_1}, \ldots, \mathtt{d_n}))$, is well-defined. This function will be called *function defined by* $\mathtt{p}$ *through* $\underline{\mathcal{R}}$.

Let $f : A_1 \times \cdots \times A_n \to A$ be a function and let $\underline{\mathcal{R}} = (\mathcal{R}_1, \ldots, \mathcal{R}_n, \mathcal{R})$ be $n+1$ representations such that $\mathcal{R}_i$ is a representation of A_i, for all $i = 1, \ldots, n$, and $\mathcal{R}$ is a representation of A. We will say that $\mathtt{p}$ *implements* f *through* $\underline{\mathcal{R}}$ (or that $\mathtt{p}$ is an *implementation of* f *through* $\underline{\mathcal{R}}$) if the following conditions hold: *i)* $\mathtt{p}$ is coherent with $\underline{\mathcal{R}}$; *ii)* $Def(F(\mathtt{p})_{\underline{\mathcal{R}}}) \subseteq Def(f)$ and $F(\mathtt{p})_{\underline{\mathcal{R}}} = f|_{Def(F(\mathtt{p})_{\underline{\mathcal{R}}})}$; *iii)* for each $(a_1, \ldots, a_n) \in Def(f)$ such that there is $(\mathtt{d_1}, \ldots, \mathtt{d_n}) \in \mathcal{S}_{\mathcal{R}_1} \times \cdots \times \mathcal{S}_{\mathcal{R}_n}$ with $\alpha_{\mathcal{R}_i}(\mathtt{d_i}) = a_i$ for all $i = 1, \ldots, n$ and $f(a_1, \ldots, a_n) \in Im(\alpha_{\mathcal{R}})$, then there is $(\mathtt{d'_1}, \ldots, \mathtt{d'_n}) \in \mathtt{S^p}$ with $\alpha_{\mathcal{R}_i}(\mathtt{d'_i}) = a_i$, for all $i = 1, \ldots, n$.

In particular, if $\mathtt{T}$ is a concrete type, the identity map defines a representation of $\mathtt{T}$ which will be called the *literal representation of* $\mathtt{T}$. Note that any program is coherent with respect to the literal representations of its types and then, a program $\mathtt{p}$ implements the function $F(\mathtt{p})$.

Now, we are going to introduce a notion of implementation of a Σ-algebra. Given a signature $\Sigma = (S, \Omega)$ and a Σ-algebra $\mathcal{A}$, an *implementation of* $\mathcal{A}$ is a pair $(\underline{\mathcal{R}}_{\mathcal{A}}, (\mathtt{p}_{\omega_{\mathcal{A}}})_{\omega \in \Omega})$, where $\underline{\mathcal{R}}_{\mathcal{A}}$ is a representation of $\mathcal{A}$ (that is, $\underline{\mathcal{R}}_{\mathcal{A}}$ is an S-family $\underline{\mathcal{R}}_{\mathcal{A}} = (\mathcal{R}_{\mathcal{A}_s})_{s \in S}$ of representations, being $\mathcal{R}_{\mathcal{A}_s}$ a representation of the carrier set $\mathcal{A}_s$, for each $s \in S$) and $(\mathtt{p}_{\omega_{\mathcal{A}}})_{\omega \in \Omega}$ is a family of programs which implement the operations $(\omega_{\mathcal{A}})_{\omega \in \Omega}$ through $\underline{\mathcal{R}}_{\mathcal{A}}$.

Moreover, the notion of transformation between implementations, or i–transformation, is defined as follows. Let $\Sigma = (S, \Omega)$ be a signature, let $\mathcal{A}$ and $\mathcal{B}$ be two Σ-algebras and let $\mathcal{I}_{\mathcal{A}} = (\underline{\mathcal{R}}_{\mathcal{A}}, (\mathtt{p}_{\omega_{\mathcal{A}}})_{\omega \in \Omega})$ and $\mathcal{I}_{\mathcal{B}} = (\underline{\mathcal{R}}_{\mathcal{B}}, (\mathtt{p}_{\omega_{\mathcal{B}}})_{\omega \in \Omega})$ be implementations of $\mathcal{A}$ and $\mathcal{B}$, respectively. An *i–transformation from* $\mathcal{I}_{\mathcal{A}}$ *to* $\mathcal{I}_{\mathcal{B}}$ is a pair $(t, f) = (t_s, f_s)_{s \in S}$ such that, for each $s \in S$, $t_s : \mathcal{D}_{\mathcal{R}_{\mathcal{A}_s}} \to \mathcal{D}_{\mathcal{R}_{\mathcal{B}_s}}$ is a partial function whose definition domain is $\mathcal{S}_{\mathcal{R}_{\mathcal{A}_s}}$ and $f_s : \mathcal{M}_{\mathcal{R}_{\mathcal{A}_s}} \to \mathcal{M}_{\mathcal{R}_{\mathcal{B}_s}}$ is a *total* function, in such a way that the following conditions hold: *i)* (t, f) is a transformation between representations (in other words, for any $s \in S$, $t_s(\mathcal{S}_{\mathcal{R}_{\mathcal{A}_s}}) \subseteq \mathcal{S}_{\mathcal{R}_{\mathcal{B}_s}}$, $f_s \circ \alpha_{\mathcal{R}_{\mathcal{A}_s}} = \alpha_{\mathcal{R}_{\mathcal{B}_s}} \circ t_s$, and besides, the representation equalities are preserved); *ii)* $f : \mathcal{A} \to \mathcal{B}$ is a total Σ-homomorphism; *iii)* and, for each operation $\omega : s_1 \ldots s_n \to s$ in Ω, it holds that $(t_{s_1}, \ldots, t_{s_n})(Def(F(\mathtt{p}_{\omega_{\mathcal{A}}}))) \subseteq$

$Def(F(\mathtt{p}_{\omega_{\mathcal{B}}}))$ and $t_s(F(\mathtt{p}_{\omega_{\mathcal{A}}})(\mathtt{d}_1,\ldots,\mathtt{d}_n)) = F(\mathtt{p}_{\omega_{\mathcal{B}}})(t_{s_1}(\mathtt{d}_1),\ldots,t_{s_n}(\mathtt{d}_n))$, for each $(\mathtt{d}_1,\ldots,\mathtt{d}_n) \in Def(F(\mathtt{p}_{\omega_{\mathcal{A}}}))$.

It is not difficult to prove that implementations and i–transformations define a category. Using the same terminology as [29], an *abstract data type* (ADT, in short) is a pair $\mathcal{T} = (\Sigma, \mathcal{C})$, where Σ is a signature and $\mathcal{C}$ is a subcategory of $Alg(\Sigma)$ (closed by isomorphism). Given an ADT $\mathcal{T} = (\Sigma, \mathcal{C})$, an *implementation of* $\mathcal{T}$ is an implementation of any Σ-algebra in $\mathcal{C}$; we will denote by $Imp(\mathcal{T})$ the category of implementations of $\mathcal{T}$.

From the programming perspective, we are interested in the type of the families of implementations of $\mathcal{T}$. In this line, from an ADT $\mathcal{T} = (\Sigma, \mathcal{C})$, we define a new ADT that we denote by $\mathcal{T}_{Imp} = (\Sigma_{Imp}, \mathcal{C}_{Imp})$, whose underlying signature is Σ_{Imp}. The intuitive idea is that an algebra belonging to $\mathcal{C}_{Imp}$ corresponds to a family of implementations of algebras in $\mathcal{C}$ (note that the domains of an implementation of $\mathcal{T}$, together with the functions defined by its programs, determine a Σ-algebra; a Σ_{Imp}-algebra belongs to $\mathcal{C}_{Imp}$ if the Σ-algebra defined for each index of the carrier set of sort imp_s comes from an implementation of an algebra of $\mathcal{C}$). The type $\mathcal{T}_{Imp}$ is called the *ADT of the families of the implementations of* $\mathcal{T}$. In order to handle a class of implementations of $\mathcal{T}$ we need only one implementation of $\mathcal{T}_{Imp}$. Following the same ideas which have led us in the algebraic level, we have to fix a data domain. At the implementation level, this consists in fixing an S-set $\underline{\mathtt{T}}$ of (Common Lisp) concrete types, that is, an S-type. In this way, only the implementations whose underlying sets of elements are (represented by) $\underline{\mathtt{T}}$ are going to be considered. This accurately fits the way in which EAT was programmed.

For a given S-type $\underline{\mathtt{T}}$, there is a distinguished Σ_{Imp}-algebra: each element of the carrier set of sort imp_s is a tuple of functions being the operations of an algebra in $\mathcal{C}$ with S-type $\underline{\mathtt{T}}$ (for any sort $s \in S$, the carrier set is the type $\mathtt{T}_s$). However, now we have to point out an important difference between handling algebras and implementations of algebras: the operations (in an algebra) are functions and the natural representation of a function in computer memory is a functional code. But, from a given functional code $\mathtt{c}$ (and a suitable family of types $\underline{\mathtt{T}}$ for $\mathtt{c}$), we cannot determine a function, since we do not know its definition domain. So, a way of determining the definition domains of the operations is needed. In [27], we solved this problem by fixing an Ω-set $\underline{\mathtt{S}}$ which describes the definition domain of (the implementations of) the operations in Ω. (Other alternatives are also possible; some of them were explored in [36] and [28].)

Therefore, fixed an S-type $\underline{\mathtt{T}}$ and an Ω-set $\underline{\mathtt{S}}$, a distinguished Σ_{Imp}-algebra can be precisely defined: it is the algebra whose carrier set for the sort imp_s consists of all the tuples of functions coming from implementations of $\mathcal{T}$ with S-type $\underline{\mathtt{T}}$ and definition domains $\underline{\mathtt{S}}$ (note that different implementations can give rise to the same tuple of functions); the operations are defined by applying the appropriate function from the first argument, an argument which is always a tuple of functions, over the other arguments. This algebra plays the same role

as the object $\mathbb{1}^D$ given in Section 4. It has the same coproduct and final object properties which were proven in [36]. In particular, the coproduct property implies that this object gathers all the implementations of $\mathcal{T}$, that is to say, each implementation of $\mathcal{T}$ is a datum of this canonical algebra. Then, an implementation of this algebra can be considered as an "implementation squared" of the ADT $\mathcal{T}$. These properties of the distinguished Σ_{Imp}-algebras allow us to only consider their implementations in order to gather all the implementations of $\mathcal{T}$.

The rest of this section is devoted to introduce an implementation of the distinguished Σ_{Imp}-algebra. This implementation is an idealization of the implementation actually used in EAT. For each S-type $\underline{\mathtt{T}}$ and each Ω-set $\underline{\mathtt{S}}$, a canonical implementation $\mathcal{I}^{EAT,\underline{\mathtt{T}},\underline{\mathtt{S}}}$ of the corresponding distinguished Σ_{Imp}-algebra was constructed in [27] (there, it was denoted by $\mathcal{I}^{EAT}$) as follows. For each sort s of Σ, the literal representation of $\mathtt{T}_s$ is chosen. Let $(\omega_1, \ldots, \omega_m)$ be an enumeration of the operation symbols of Σ. For the distinguished sort imp_s, we choose as type for imp_s, $\mathcal{D}_{imp\_s}^{EAT,\underline{\mathtt{T}},\underline{\mathtt{S}}}$, the instances of the structure

```
(defstruct IMP-s  imp-ω1  ... imp-ωm)
```

such that their slots are Common Lisp functional codes. (For further details on the use of the Common Lisp structures, the reader can refer to [49].) An instance of $\mathcal{D}_{imp\_s}^{EAT,\underline{\mathtt{T}},\underline{\mathtt{S}}}$ will be in the support $\mathcal{S}_{imp\_s}^{EAT,\underline{\mathtt{T}},\underline{\mathtt{S}}}$ whether there exists at least one implementation of the initial ADT $\mathcal{T}$ such that the codes of their programs are the components of the instance and the definition domains of the programs implementing the operations are $\underline{\mathtt{S}}$. An example of this construction in EAT was briefly evoked in Subsection 1.2, in the case of chain complexes.

The abstraction function carries each struct to the tuple of functions defined by these functional codes on $\underline{\mathtt{S}}$. We consider as representation equality the abstraction equality, which is, in essence, the behavioural equality in each component. (Two functional codes $\mathtt{c}_1$, $\mathtt{c}_2$ are *behaviourally equal* with respect to $n+1$ concrete types $\mathtt{T}_1, \ldots, \mathtt{T}_n, \mathtt{T}$ and $\mathtt{S} \subseteq \mathtt{T}_1 \times \cdots \times \mathtt{T}_n$ if $F(\mathtt{p}_1) = F(\mathtt{p}_2)$, where $\mathtt{p}_i = (\mathtt{c}_i, \mathtt{S}, \mathtt{T}_1, \ldots, \mathtt{T}_n, \mathtt{T})$, $i = 1, 2$ must be programs.)

Finally, the program chosen to implement imp_ω_i has $\mathcal{S}_{imp\_s}^{EAT,\underline{\mathtt{T}},\underline{\mathtt{S}}} \times \mathtt{S}^{\omega_i}$ as definition domain and its functional code is:

```
#'(lambda(instance-of-IMP-s d1 ... dn)
    (funcall(IMP-s-imp-ωi instance-of-IMP-s) d1 ... dn))
```

As we have said, the object $\mathcal{I}^{EAT,\underline{\mathtt{T}},\underline{\mathtt{S}}}$ is aimed at capturing the nature of the implementations actually used in the EAT system. Although each actual mathematical structure in EAT (chain complexes or simplicial sets, for instance) has certain particularities, we claim that the objects $\mathcal{I}^{EAT,\underline{\mathtt{T}},\underline{\mathtt{S}}}$ accurately express the very nature of the EAT implementations.

We still have to introduce the category where the EAT implementation pattern was characterized, which will be denoted by $Imp^{\underline{\mathtt{T}},\underline{\mathtt{S}}}(\mathcal{T}_{Imp})$. This category

is intended to collect information on some implementations of the corresponding distinguished Σ_{Imp}-algebra; roughly speaking, those which can be defined on $\underline{\mathtt{T}}$ and $\underline{\mathtt{S}}$ (some more technical constraints have to be imposed; the interested reader can find them in [27]). Moreover, the morphisms in $Imp^{\underline{\mathtt{T}},\underline{\mathtt{S}}}(\mathcal{T}_{Imp})$ are transformations between implementations such that they are identities on the concrete types $\underline{\mathtt{T}}$ (that is, all the implementations we are going to consider share the same way of representation for the elements of the mathematical structures which are being implemented).

We will denote by $Imp^{\underline{\mathtt{T}},\underline{\mathtt{S}}}(\mathcal{T})$ the category of implementations of $\mathcal{T}$ on $\underline{\mathtt{T}}$ and $\underline{\mathtt{S}}$ (under constraints similar to those of the implementations in $Imp^{\underline{\mathtt{T}},\underline{\mathtt{S}}}(\mathcal{T}_{Imp})$; see [27]). Finally, let us consider the canonical embedding of $Imp^{\underline{\mathtt{T}},\underline{\mathtt{S}}}(\mathcal{T})$ in $Imp^{\underline{\mathtt{T}},\underline{\mathtt{S}}}(\mathcal{T}_{Imp})$ which is defined by considering an implementation I of $\mathcal{T}$ as an implementation of $\mathcal{T}_{Imp}$ with only one element in the support of the sort imp_s (this element is the tuple of functional objects which realize the operations in I). Bearing in mind these notations, using the machinery introduced in [27] and following the guidelines provided in Section 5 for the algebraic case, the following result is proven.

Theorem 6. *The object $\mathcal{I}^{EAT,\underline{\mathtt{T}},\underline{\mathtt{S}}}$ is the coproduct in $Imp^{\underline{\mathtt{T}},\underline{\mathtt{S}}}(\mathcal{T}_{Imp})$ of the implementations in the image of the canonical embedding functor of $Imp^{\underline{\mathtt{T}},\underline{\mathtt{S}}}(\mathcal{T})$ in $Imp^{\underline{\mathtt{T}},\underline{\mathtt{S}}}(\mathcal{T}_{Imp})$. In addition, there is at least one morphism from any object of $Imp^{\underline{\mathtt{T}},\underline{\mathtt{S}}}(\mathcal{T}_{Imp})$ to $\mathcal{I}^{EAT,\underline{\mathtt{T}},\underline{\mathtt{S}}}$.*

Note that the previous theorem shows that the implementation $\mathcal{I}^{EAT,\underline{\mathtt{T}},\underline{\mathtt{S}}}$ gathers all the individual implementations of $\mathcal{T}$ and, moreover, it can be considered as a *maximal* implementation: there is a natural path from any implementation of $\mathcal{T}_{Imp}$ (on $\underline{\mathtt{T}}$ and $\underline{\mathtt{S}}$) to $\mathcal{I}^{EAT,\underline{\mathtt{T}},\underline{\mathtt{S}}}$. Then, the main result in [27] appears as a direct consequence of this theorem (the reader has to take into account the changes in the notation when comparing both results).

Corollary 2. *The object $\mathcal{I}^{EAT,\underline{\mathtt{T}},\underline{\mathtt{S}}}$ is the final object of the category $Imp^{\underline{\mathtt{T}},\underline{\mathtt{S}}}(\mathcal{T}_{Imp})$.*

Finally, if we denote by $Imp(\mathcal{T}_{Imp})$ the category obtained by grouping the categories $Imp^{\underline{\mathtt{T}},\underline{\mathtt{S}}}(\mathcal{T}_{Imp})$ when $\underline{\mathtt{S}}$ ranges over the set of subsets of $\underline{\mathtt{T}}$ and then $\underline{\mathtt{T}}$ ranges over the collection of S-types, the following result can be proven, explaining how the different final objects are gathered.

Theorem 7. *The family $\{\mathcal{I}^{EAT,\underline{\mathtt{T}},\underline{\mathtt{S}}}\}$ is final in the category $Imp(\mathcal{T}_{Imp})$.*

7 Conclusions and Further Work

In the last section we have stressed how the ideas suggested by certain formalisms for the study of object-oriented programming can be applied to analyze a

concrete symbolic computation system, namely the EAT system. In particular, object-orientation appears in EAT when modelling its data structures as *families of implementations* of abstract data types. As a consequence, some results extending the previous work on the subject [27] have been introduced, in such a way that the canonical EAT implementations can be characterized in a larger category.

Another contribution of our work can be interpreted as an application *from* symbolic computation *to* the theoretical study of object-oriented programming. First, at a conceptual level, because in our very concrete problem (the formal analysis of a symbolic computation system for Algebraic Topology) two main trends in the formalization of object-oriented programming appear: that of "calculus for objects" (see, for instance, [1]) and that of hidden specifications (see, for instance, [20]).

Second, at a more technical level, this relationship allows us to offer a simpler description of the final object which appears in the hidden context (the "magical formula" in [20]) and moreover it allows to study this object in categories larger than the hidden category. Even if our results only work in a particular case (immutable objects or, equivalently, hidden signatures without constructors from data or updating operations), this opens lines for future research. We conjecture that the general case tackled in [20] (hidden signatures without constructors from data) admits a description in our functional context. We also conjecture that this description will establish a link with Cardelli's *recursive records* [8]. Still in order to bring the hidden algebraic specifications closer to the object-oriented programming language type theory, the study of the relations between the *existential types* [31] and our final objects should be undertaken (since our Σ_{Imp}-algebras seem to be a convenient domain over which existentially quantified variables could range).

With respect to the applications to Symbolic Computation, a next step will be to study how our modelling of the system can be used in order to verify (i.e. to prove the correctness of) certain fragments of the EAT program (see some preliminary results in [3]). But, before undertaking this task as a whole, the modelling of EAT should be completed (since in [27], [28] and in this paper we have only dealt with *isolated* data structures; see our comments on constructors in Section 5, and some particular results in [13]).

The following step is to adapt our constructions and results to the Kenzo system, the last of Sergeraert's systems to compute homology and homotopy groups (see some preliminary results in [14]). Kenzo, which is much better than EAT in terms of performance [15], was developed using the powerful object-oriented tools provided in CLOS (Common Lisp Object System). However, the implicit object-oriented features of the EAT system (which have been analyzed in this paper) are also present in Kenzo, and thus, the question of how they can be integrated (into our algebraic specifications framework) with a more explicit CLOS (class-based) object-orientation is unsolved. The problem is even more challenging because Kenzo relies on the *multiple inheritance* abilities of CLOS,

and it is well-known that the formal study of multiple inheritance is much more difficult than that of simple inheritance (see, for instance, [1], [7]).

Finally, we have planned to transfer our results to general symbolic computation environments, such as Axiom [26]. We have also foreseen to translate our approach to typed functional programming languages, such as ML [37], where our results could be properly expressed.

Acknowledgement. We would specially like to thank the referees for their comments and suggestions for improvements to the first version of this paper.

References

1. Abadi, M., Cardelli, L.: A theory of objects, Springer, 1996
2. Alagar, V.S., Periyasamy, K.: Specification of Software Systems, Springer, 1998
3. Aransay, J., Ballarin, C., Rubio, J.: Mechanising proofs in Homological Algebra. Calculemus Autumn School: Poster Abstracts, SEKI Report SR- 02–06, 2002, pp. 13–18
4. Barr, M.: Terminal coalgebras in well-founded set theory. Theor. Comput. Sci. **114**, 299–315 (1993)
5. Börger, E. (ed.) Specification and Validation Methods. Clarendon Press, 1995
6. Burstall, R., Diaconescu, R.: Hiding and behaviour: an institutional approach. Essays in Honour of C.A.R. Hoare, Prentice Hall, 1994, pp. 75–92
7. Cardelli, L.: A semantics of multiple inheritance. Inf. Comput. **76**, 138–164 (1985)
8. Cardelli, L., Mitchell, J.C.: Operations on records. Math. Struct. Comput. Sci. **1**, 3–48 (1991)
9. Cîrstea, C.: Coalgebra semantics for hidden algebra: parameterised objects and inheritance. Lecture Notes in Comput. Sci. **1376**, 174–189 (1997)
10. Cîrstea, C.: Semantic constructions for the specification of objects. Lecture Notes in Comp. Sci. **1589**, 63–78 (1999)
11. Cook, W.R.: Object-oriented programming versus abstract data types. Lecture Notes in Comput. Sci. **489**, 151–178 (1991)
12. Diers, Y.: Familles universelles de morphismes. Annales de la Société Scientifique de Bruxelles **93**, 175–195 (1979)
13. Domínguez, C., Lambán, L., Pascual, V., Rubio, J.: Hidden Specification of a Functional System. Lecture Notes in Comput. Sci. **2178**, 555–569 (2001)
14. Domínguez, C., Rubio, J.: Modeling inheritance as coercion in a Symbolic Computation System. In Proceedings ISSAC'2001, ACM Press, 2001, pp. 107–115
15. Dousson, X., Sergeraert, F., Siret, Y.: The Kenzo program. http://www-fourier.ujf-grenoble.fr/∼Sergerat/Kenzo/. Institut Fourier, Grenoble, 1999
16. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specifications 1, Springer, 1985
17. Giarratana, V., Gimona, F., Montanari, U.: Observability concepts in abstract data type specifications. Lecture Notes in Comput. Sci. **45**, 576–587 (1976)
18. Goguen, J.: Types as theories. In Topology and category theory in computer science, Oxford University Press, 1991, pp. 357–390
19. Goguen, J., Diaconescu, R.: Towards an algebraic semantics for the object paradigm. Lecture Notes in Comput. Sci. **785**, 1–29 (1994)
20. Goguen, J., Malcolm, G.: A hidden agenda. Theor. Comput. Sci. **245**(1), 55–101 (2000)
21. Goguen, J., Meseguer, J.: Universal realization, persistent interconnection and implementation of abstract modules. Lecture Notes in Comput. Sci. **140**, 265–281 (1982)
22. Graham, P.: ANSI Common Lisp. Prentice Hall, 1996
23. Hennicker, R.: Observational Implementations. Lecture Notes in Comput. Sci. **349**, 59–71 (1989)

24. Hoare, C.A.R.: Proofs of correctness of data representations. Acta Inf. **1**, 271–281 (1972)
25. Jacobs, B.: Mongruences and cofree coalgebras. Lecture Notes in Comput. Sci. **936**, 245–260 (1995)
26. Jenks, R.D., Sutor, R.S.: AXIOM: the scientific computation system. Springer, 1992
27. Lambán, L., Pascual, V., Rubio, J.: Specifying implementations. In Proceedings ISSAC'99, ACM Press, 1999, pp. 245–251
28. Lambán, L., Pascual, V., Rubio, J.: Locally effective objects and abstract data types. Preprint, 2002
29. Loeckx, J., Ehrich, H.D., Wolf, M.: Specification of Abstract Data Types. Wiley–Teubner, 1996
30. Malcolm, G.: Behavioural equivalence, bisimulation and minimal realisation. Lecture Notes in Comput. Sci. **1130**, 359–378 (1996)
31. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential types. ACM Transactions on Programming Languages and Systems **10**, 470–502 (1988)
32. Nissanke, N.: Formal Specification. Springer, 1999
33. Nivela, P., Orejas, F.: Initial behavioural semantics for algebraic specifications. Lecture Notes in Comput. Sci. **332**, 184–207 (1988)
34. Orejas, F., Navarro, M., Sánchez, A.: Implementation and Behavioural Equivalence: a Survey. Lecture Notes in Comput. Sci. **655**, 93–125 (1996)
35. Orejas, F., Nivela, P., Ehrig, H.: Semantical constructions for categories of behavioural specifications. Lecture Notes in Comput. Sci. **393**, 220–241 (1989)
36. Pascual, V.: Objetos Localmente Efectivos y Tipos Abstractos de Datos, Ph.D. Thesis, Universidad de La Rioja, 2002
37. Paulson, L.C.: ML for the working programmer. Cambridge University Press, 2000
38. Reichel, H.: Behavioural equivalence – a unifying concept for initial and final specification methods. Arato M., Varga L., (eds.) Third Hungarian Computer Science Conference, Budapest, Akademiai Kiado, 1981, pp. 27–39
39. Reichel, H.: An approach to object semantics based on terminal coalgebras. Math. Struct. Comput. Sci. **5**, 129–152 (1995)
40. Rubio, J.: Homologie Effective des espaces de lacets itérés: un logiciel. Thèse, Institut Fourier, Grenoble, 1991
41. Rubio, J., Sergeraert, F.: Locally effective objects and algebraic topology. In Computational Algebraic Geometry, Birkhaüser, 1993, pp. 235–251
42. Rubio, J., Sergeraert, F., Siret, Y.: EAT: Symbolic Software for Effective Homology Computation. ftp://fourier.ujf-grenoble.fr/pub/EAT. Institut Fourier, Grenoble, 1997
43. Rubio, J., Sergeraert, F., Siret, Y.: Overview of EAT, a System for Effective Homology Computation, The SAC Newsletter. **3**, 69–79 (1998)
44. Rutten, J.J.M.M.: Universal coalgebra: a theory of systems. Theor. Comput. Sci. **249**(1), 3–80 (2000)
45. Sannella, D., Tarlecki, A.: On observational equivalence and algebraic specification. J. Comput. Syst. Sci. **34**, 150–178 (1987)
46. Sergeraert, F.: Functional coding and effective homology. Astérisque **192**, 57–67 (1990)
47. Sergeraert, F.: The computability problem in algebraic topology. Advances in Math. **104**, 1–29 (1994)
48. Smith, D.R.: Designware: Software development by refinement. In Proceedings of the Eighth International Conference on Category Theory and Computer Science, 1999
49. Steele, G.: Common Lisp. The language. Second Edition. Digital Press, 1990
50. Tarlecki, A., Burstall, R.M., Goguen, J.A.: Some fundamental algebraic tools for the semantics of computation: Part 3. Indexed categories. Theor. Comput. Sci. **91**, 239–264 (1991)
51. Wand, M.: Final algebra semantics and data type extensions. J. Comput. Syst. Sci. **19**, 27–44 (1979)
52. Wegner, P.: Concepts and paradigms of object-oriented programming. OOPS Messenger **1**, 7–87 (1990)