On the way to certify Computer Algebra Systems

S. Boulmé, T. Hardin, D. Hirschkoff, V. Ménissier-Morain, and R. Rioboo

Laboratoire d'Informatique de Paris 6 (LIP6),
Université Pierre et Marie Curie (Paris 6),
8, rue du Capitaine Scott, 75015 Paris, France.
Therese.Hardin@lip6.fr

++

CALCULEMUS-2001 --- SUBMISSION FORM

++

TITLE: "Some hints for polynomials in the FOC project"

AUTHORS' NAMES: "Boulme, Hardin, Rioboo"

PAPER CATEGORY full paper / system description: "full paper"

KEYWORDS: 2-3 from the keywords listed below, or others: "CAS FM Systems"
[ATP (Automated Theorem Proving)]
[IPD (Interactive Proof Development]
[TRS (Term Revriting Systems)]
[CAV (Computer-Aided Verification)]
[SE (Software Engineering)]
[CAS (Computer Algebra Systems)]
[SC (Symbolic Computation)]
[NC (Numeric Computation)]
[FM (Formal Methods)]
[Theory]
[Systems]
[Applications]

ABSTRACT (max 200 words):
"The Foc project aims at supporting, within a coherent software system,
the entire process of mathematical computation, starting with proved
theories, ending with certified implementations of algorithms. In this
paper, we explain our design requirements for the implementation, using
polynomials as a running example. Indeed, proving correctness of
implementations depends heavily on the way this design allows
mathematical properties to be truly handled at the programming level. "

CORRESPONDING AUTHOR
- NAME: "Hardin Therese"
- EMAIL: "Therese.Hardin@lip6.fr"
- FULL ADDRESS: "LIP6, Universite Paris 6,
  8, Rue du Capitaine Scott, 75015 Paris, France"
- PHONE: "33 + 1 44 27 73 69"
- FAX: " 33 + 1 44 27 88 78"

Abstract. The FOC project, started at the fall 1997, is aimed to build a programming environment for the development of certified symbolic computation. The working languages are Coq and Ocaml. In this paper, we present first the motivations of the project. We then explain why and how our concern for proving properties of programs has led us to certain implementation choices in Ocaml. This way, the sources express exactly the mathematical dependencies between the different structures. This may ease the achievement of proofs.

1 Introduction

By definition, Computer Algebra Systems perform exact computations, mathematical entities are represented by terms belonging to a formal language, whose rules describe the computations, and there is no notion of numerical approximation, at least in their kernel. The algorithms, either the system's ones or user-implemented ones, rely upon mathematical properties which have been completely proved. So there seems to be little place for bugs if the implementations are carefully managed as is usually the case.

Despite of this care, bugs occur: hasty simplifications, no verification of required assumptions such that subexpressions being nonnegative, etc. (see for example [9]), i.e. what one could consider algorithmic errors. But there are also bugs introduced by the coding itself. The first problem is that computer algebra programs tend to be large and complicated, and hence difficult to maintain. The second problem comes from the specific difficulty in the task of testing symbolic manipulations, due to the size of the data or the time needed for verification. Typically, it is the case that polynomial coefficients with several hundreds of digits are encountered in practice. More generally, several hours, even several days, of CPU time are commonly needed to achieve a computation. Therefore, doing extensive testing may be too costly, even if verifying the correctness of an answer can sometimes be considerably easier that its computation (e.g. checking that a number is a root of a given polynomial). In some cases, the answer may also be non-constructive, when there is no solution. This happens for example when a polynomial is irreducible; in such a situation, no verification can be done, it is only possible to try to compute again the result with another algorithm or another Computer Algebra System. In the present state of the art, however, there may simply in some cases be no other way to compute this answer again.

A classical way to tackle this problem is to reduce the gap between the mathematical statement of an algorithm and its encoding in the programming language. This requires a syntax which is powerful enough to reflect mathematical properties, as well as a semantics associated to this syntax. As in the past years (see for example Davenport[4]), there were no programming language that fully met the requirements for being a Computer Algebra System programming language, the Computer Algebra community has been led to develop its specific programming languages, giving birth to powerful systems, e.g. Axiom[5]. In this system, the user manipulates *categories* and *domains*, which are akin to classes of object-oriented languages but with very appropriate syntax, type system, etc., in order to allow the source program to remain close to the mathematical algorithm.

But this effort is not yet sufficient to get rid of bugs or ambiguities: for instance, if the way the multiple inheritance conflicts are solved is not completely described, misleading interpretation may be a source of bugs in user programs. Indeed, although Axiom introduces a certain programming discipline by allowing the user to make statements over the entities being manipulated, there is no effective semantic control over these statements. For such a verification to actually take place, one could imagine a scenario where the user has to provide a proof of the statements he wishes to "decorate" his programs with. Typically, whenever he declares some new object as having a group structure, he would have to effectively demonstrate that the definitional properties of a group

hold. This goes back to the fundamental issue of whether existing computer algebra systems and deduction systems can be integrated.

To define a code-certification mechanism in order to fulfill these tasks, several solutions can be taken into account.Within a previous project, we have tried to interface Axiom and the Coq proof assistant, with the aim of proving some properties of Axiom programs[1]. A prototype has been done, consisting firstly in interfaces between libraries of Axiom (basic, sets, ordered sets, ..., groups) with the corresponding theories in Coq, and secondly, in a compiler from Axiom functions to their specifications in Coq. The emphasis has been put on the compilation of the hierarchy of categories and domains; as far of the actual code is concerned, only the imperative kernel was considered. The main outcome of this previous work was that such a task should rather be attacked using a programming language whose semantics is fully understood (and, possibly, formalized). Indeed, for example, some difficulties arose in modeling multiple inheritance within Axiom programs, mostly because we could not coin a precise semantics of this (quite intricate) programming construct. Overall, it seems that in some sense the Axiom programmer has too much freedom for his implementation task, and can hence design programs that are quite difficult, if not impossible, to certify. In particular, the management of definitions *by default*, as well as definition overriding, seems difficult to handle when it comes to guarantee a reasonable form of coherence within the "proof" part of a development.

The FOC project (F for Formel i.e. symbolic in French + O for Ocaml + C for Coq), which started at fall 1997, is based on an attempt in interfacing Axiom and Coq. The ambition is to build an environment for the development of certified programs for symbolic computation, with three components, namely a programming language, a proof development tool, and an environment to federate the two latter parts as two different views of the same objects.

Tools for symbolic computation programming are essentially based on libraries for algebraic structures; FOC should contain such a library. In order to develop certified programs, this library should provide not only the implementation of the classical tools to manipulate algebraic structures, but also their semantics, given by explicit statements and proved properties. A user should have the possibility to specify a given algorithm by putting together elements of this library, prove properties of this algorithm, define an implementation and prove its correctness. Consequently, the interaction between the programming language and the proof assistant should be very strong in the FOC system.

To obtain a full certification, the compiler of the programming language should itself be certified. No such compiler exists for the time being, even if some kernels of functional languages have been formally studied. Nevertheless, it would be completely irrealistic to try to create our own programming language. We have therefore chosen a programming language which is semantically well founded, namely Ocaml[6]. This language belongs to the ML family, and as such, goes back to ML, the meta-language of the LCF prover. The language Ocaml offers a very strong system of modules, object-oriented features with well-understood semantics and many user libraries. Furthermore, Ocaml is the implementation language of Coq, the proof development tool retained for this project. Finally, Ocaml programs have very reasonable run times to satisfy the permanent concern for time and space efficiency of symbolic computation programmers.

As already said, the proof tool we have adopted is Coq[2], a proof assistant based on the Calculus of Inductive Constructions (CIC), a lambda-calculus equipped with a very expressive language of types. Within Coq, we would like to fully describe the different inheritance and dependence relations (rings from additive groups and multiplicative monoids, the building of recursive polynomials upon sparse polynomials, etc.). We would also like to prove mathematical properties not only of specifications, but also of programs. Once implemented, these definitions and proofs have to be reusable,

as Computer Algebra research constantly introduces new algorithms. Our previous experience has shown that, even while still being under development, Coq can fit these requirements.

The goals of the FOC project seem rather effort-demanding and very ambitious. Due to the complexity of the data structures manipulated by symbolic computations, it does not seem reasonable to hope to complete such a work, and cover the whole area of Computer Algebra Systems. Nevertheless, one can hope to certify some complex algorithms such as the computation of greatest common divisors (gcds for short) of polynomials with the subresultant method, as well as (parts of) the programs that encode them, like inheritance links, for example. But it is out of our scope to certify e.g. the firing of exceptions within the compiler, even if it is possible to model the use of exceptions in specifications.

In this paper, we present the first step of the FOC project, which consisted in fixing the development methodology. We first focus on the conception of the programming part, in order to test whether Ocaml reaches our efficiency requirements. In doing that, we kept in mind the will of facilitating the proof facet of the work. Several programming prototypes have been developed, the selection criterion being the point of view of the proof. Within this effort, the running example for testing our prototypes is the computation of the gcd of polynomials over a factorial ring, by the subresultant method.

Section 2 details the requirements that are made for the first part of the FOC development. Section 3 states the reason why we have rejected the solution based only on modules. In section 4, we present several ways of using the object-oriented features, commenting on their drawbacks. The retained solution is given in section 5, in a rather detailed way. Along the paper, we provide examples written in Ocaml, trying to remain understandable to people not acquainted with this language.

2 The project requirements

The first step in our effort is the definition of what lies at the heart of a Computer Algebra System, namely a library of algebraic structures, together with the relations between them. It is important at this point to understand precisely what are the requirements to be formulated about this core part, in particular with respect to the aim of being able to perform proofs about the various constructs it contains. Together with the definition of those features that seem to be crucial to reach this goal, we felt the need to introduce a specific terminology to reason about Computer Algebra Systems from both a programming and a code verification perspective: this will be presented at the end of the section.

2.1 Software requirements

The requirements we formulate for the development of the core part of a Computer Algebra System, and which were respected along the experiments we made, are the following:

1. The overall organization of the library should reflect its mathematical counterpart, e.g. groups should be defined upon monoids.
2. It should be possible to manipulate notions at several levels of abstraction; for example one should be able to exploit the properties of a given group (say the type of an operation) without dealing with its actual implementation.
3. One should have the possibility of providing a definition *by default* of certain notions, so that they can be shared by a whole family of structures, and still possibly be locally redefined for a specific inhabitant of the family. For example, defining by default `is_different` as the

negation of `is_equal` should be a standard construction for structures that are built upon sets (an equality relation being a constituent of a set), but one may sometimes want to redefine it within some of them.

4. Along these lines, one should progressively refine the implementations of an algebraic structure, e.g. to go from a structure giving an abstract view of $\mathbb{Z}/2\mathbb{Z}$ to an implementation using integers and to another one using booleans for representation of the inhabitants. In doing this, however, we want to be able to share some constructions between both implementations. At the same time, we also want to distinguish between these structures via typing, in order to avoid confusions or misuses.
5. Of course, the library that is built should contain a significant amount of basic notions that are common in Computer Algebra: it should in particular contain big integers, modular integers, and several representations of polynomials, at least the distributed and the recursive ones. It seems indeed reasonable to think that the problems arising at the level of certification can be visible only after a certain amount of complexity, both in the organization of the algebraic structures and in implementation issues, has been reached.

In conforming to these five requirements, we shall try to keep the correspondence between the Ocaml description of the structures we handle and their Coq counterpart as natural as possible, in order to ease the development of the proofs. Moreover, when going through our implementations, we shall put the emphasis on the typing discipline, still respecting the points stated above.

2.2 Meeting these requirements

The requirements formulated above are not specific to Computer Algebra; they correspond to well-known paradigms in programming languages, such as strong typing, polymorphism, genericity, definitions by default, inheritance and late binding, sharing and reuse. This variety of features cannot be easily described with the basic type constructs of programming languages (that is union and record types). The requirement 2. asks for abstract data types on one hand and concrete (or manifest) types on the other hand, so for a performing system of modules. The requirement 3. can be satisfied by an inheritance mechanism coupled with a late-binding possibility. Thus, these requirements meet the ones which led to the introduction of modules and objects in programming languages.

The language Ocaml has a very strong discipline of types, with parametric polymorphism and type inference, it provides both modules and objects, which are powerful enough to define our library. Moreover, the interaction between classing and subclassing mechanism and the typing algorithm is fully described and semantically understood. These are the reasons of our choice. This choice being done, we are not yet ready to start the development. In fact, our requirements are in a certain sense contradictory. Indeed, one wishes to define $\mathbb{Z}/2\mathbb{Z}$ and $\mathbb{Z}$ as two different types (module-oriented aspect), and at the same time to share some constructions between these two rings (object-oriented aspect). We thus have to elaborate a design discipline for the definition of our library, through the understanding of the balance we need between the module-oriented and the object-oriented aspects of our programming idiom. Doing that, we have also to keep in mind that our programs are to be proved, so, the code must reflect as far as possible its mathematical counterpart and the different dependency links between the libraries must be described in a rather uniform way.

In order to have an account of the actual difficulties that can arise due to the programming style we chose, and of the solutions that can be proposed to handle these, we have been experimenting on several versions of a basic library. As said above, this library consists in programming tools which

are used to compute the gcd of polynomials over factorial rings. The resulting implementations have been analyzed according to three criteria: whether they fit to the mathematical specification of their constituents, whether they are easy to handle both from the developer's and from the user's point of view, and whether they give rise to efficient algorithms. This last point is important because there is no use to pay for the proof of a program if it will be rejected, due to inefficiency reasons.

2.3 Vocabulary of the project

Along this study, we have been led to define a specific terminology to distinguish between the two points of view on the objects being manipulated, the abstract, or mathematical one, and the concrete, or computational one. We give a short account of this terminology, as we shall use it in the following.

In Computer Algebra, one handles (abstract) *individuals*, like for example $X+2$, or (concrete) *entities*, like [(1,1);(2,0)], that can be a representation of the latter polynomial. *Societies* are used to express a relation between individuals, via *functionalities*, e.g. the relation given by an internal composition law verifying certain properties. Entities in turn belong to *collections*, put together via *operations* (that define a relation between manipulations on entities).

Prototypes are used to describe functionalities to regroup societies in a *gender*, which corresponds to a mathematical structure. Accordingly, *signatures* are used to describe operations to regroup collections in *species*. A collection has semantical properties, which depend upon its species and its representation. One should for example provide several collections of polynomials (e.g. corresponding to a distributed representation, or to a recursive one).

3 Using modules only

We first present an attempt of building libraries, using only modules. This attempt is reasonable as the Ocaml modules offer a smooth way to go from n abstract data type such as `ring` in the following to several implementations such as `Z2z` also given in the following. Moreover, the typing system is a very powerful help for the developer as it retrieves a lot of inconsistencies.

Ocaml's modules system is a simply typed lambda-calculus language with a subtyping relation and constraint expressions. This language is independent from the core language, which allows separate compilation. *Structures* are the basic notion; they allow to package together definitions sharing a common environment. In a "record" programming manner, these definitions can be referred to, outside the structure, using the "dot notation", that is, identifiers qualified by a structure name.

Signatures are interfaces for structures. A signature specifies the name and the type of the components of a structure, which are available from the outside. It can be used to hide some components of a structure (e.g. local function definitions) or to export some components with a restricted type.

Functors are "higher-order functions" from structures to structures. A structure A parameterized by a structure B is simply a functor F with a formal parameter B (along with the expected signature for B) which returns the actual structure A itself. The functor F can then be applied to one or several implementations B_1 ... B_n of B, yielding the corresponding structures A_1 ... A_n.

Within the framework of modules, Ocaml's module signatures should be used to represent *species*, and accordingly, module implementations should encode *collections*. This way, once *societies* (i.e. mathematical structures) are known, the corresponding collections can be statically implemented, thus avoiding run-time cost.

3.1 Abstraction and type inference

The abstract versus manifest type mechanism naturally fits to the encapsulation of data representation, corresponding to a module-oriented programming style. This mechanism provides both safety and modularity to the code. Indeed, data representation often uses implicit invariants. Abstract types forbids the user to misuse the representation by ignoring some of these invariants. Furthermore, the representation can be changed without disturbing users.

For example, the species of rings is described by the following signature:

```
module type Ring =
sig
  type t
  val equal: (t*t) -> bool
  val plus: (t*t) -> t
  val mult: (t*t) -> t
  val opp: t -> t
  val one: t
end
```

Then the $\mathbb{Z}/2\mathbb{Z}$ ring is defined by:

```
module Z2z: Ring =
struct
  type t=bool
  let equal (x,y) = (x=y)
  let plus (x,y) = (x || y) && not (x && y)
  let mult (x,y) = x && y
  let opp x = x
  let one=true
end
```

The type of the entities of `Z2z` (`should_be_zero` for example) is abstract, which means that the fact that these entities are indeed booleans is hidden, as shown by the following attempt:

```
# Z2z.plus (Z2z.one,true);;
This expression has type Z2z.t * bool but is here used with type Z2z.t * Z2z.t
```

This type abstraction mechanism is reinforced by the use of functors that allow one to obtain the desired level of type abstraction within parameterized collections. Let us illustrate this on by example.

The species of univariate polynomials can be described as follows:

```
module type FormalPoly =
sig
  module Base: Ring
  type t
  val equal: (t*t) -> bool
  val mult_extern: (Base.t*t)->t
end
```

The parameterized collection of sparse polynomials is described as follows:

```
module SparsePoly (A:Ring)
       : (FormalPoly with module Base = A ) =
struct
  module Base = A
  type t= (A.t * int) list
  let equal = ...
  ...
end
```

The collection of sparse polynomials with coefficients in $\mathbb{Z}/2\mathbb{Z}$ is then given by `module PolZ2z = SparsePoly (Z2z)`. In the following example, `PolZ2z.t` and `(Z2z.t*int) list` are incompatible types: the structures importing the sparse polynomials do not have access to their representation, that can be modified in a transparent way. At the same time, we indeed have that `PolZ2z.Base.t` and `Z2z.t` are equal types. This evidenciates the correctness of the coding with respect to the specification.

```
# let id y = PolZ2z.mult_extern (Z2z.one,y);;
val id : PolZ2z.t -> PolZ2z.t = <fun>
# id [(Z2z.one,1)];;
This expression has type (Z2z.t * int) list but is here used with type
PolZ2z.t = SparsePoly(Z2z).t
```

It turns out that Ocaml's modules system offers a typing algorithm powerful enough to allow for an exact description of the specification and to control the correctness of this description.

3.2 Inheritance

We want to be able to define algorithms that can be applied to a whole family of structures, and, if needed, to refine these operations for specific cases. In other words, we need an inheritance mechanism. To model inheritance using modules turns out to be unrealistic when it comes to real-size attempts. We illustrate this statement on a case study, the implementation of integral domains, trying different strategies for embedding inheritance.

A straightforward solution would be:

```
module IntegralDomain =
  sig
    module A: Ring
    val exquo: A.t*A.t->A.t
  end
```

With this choice, since a ring `A` is built on a group `G`, that in turn inherits from a monoid `M`, in order to access in `A` a field belonging to `M`, one would have to concatenate prefixes, leading to a construction like "`A.G.M...`". This means that at any point, the full knowledge of the hierarchy is explicitly required, which seems to be too demanding for the user (typically, in a minimal setting, we estimate that about 50 modules, distributed over a dozen of layers, are necessary).

To avoid this, we can use the subtyping expressed by the inclusion of signatures to full effect. Accordingly, an integral domain can be seen as a subtype of a ring, by cloning the "ring" components within the signature of the integral domain, as follows:

```
module IntegralDomain =
  sig
    type t
    val equal: t*t -> bool          val plus: t*t -> t
    val mult: t*t -> t              val exquo: t*t -> t
    ....
  end
```

Following this approach, we have developed a small prototype containing 25 modules and 8 layers. Here again, this methodology is not tractable in practice, basically for robustness considerations pertaining to software maintenance. Each time we modify a module, every module importing it has to be updated accordingly. Such a mechanism of "cut and paste" should be automatized, but this seems rather difficult for module structures and functors, basically because such a mechanism would require a semantical analysis of code. Furthermore, modules are designed to minimize the propagation of modifications during code generation, allowing separate compilation: such a way of using them would thus not respect the purpose of modules.

Along the mechanism of inheritance, it seems crucial to have the possibility to redefine certain operations (typically for optimization purposes). Late binding allows one to modify only those fields that need to be redefined and these modifications have not to be reported in the methods using these fields.

This can be useful for example when designing a version of `mult_extern` (the function to compute the product of a polynomial with a scalar number) for sparse polynomials, to take advantage of the specific representation we have for this kind of polynomials.

These considerations are expressed by a few design assumptions, that we state in the framework of our terminology as follows. A species is defined by some *primitive* operations and may have some operations defined *by default*, using the primitive ones. When species are refined into collections, every primitive operation should be implemented. For example, gcd can only be a primitive operation for factorial rings. In Euclidean rings, however, it is defined by default using Euclidean division. At

the level of collections, we want sometimes the implementation of an Euclidean ring to be seen as an implementation of a factorial one, without handling the underlying conversions explicitly. We will not enter the tedious technical work involved in the management of such an interaction in the module framework. Let us just say that we retrieve in some way the phenomenon seen above, where some parts of a signature have to be explicitly cloned at various places in the library, which seems to be a serious drawback inherent to the "only modules" approach.

4 Programming with objects

For our purposes, having a true inheritance and late binding mechanism seems to be a necessity; we are thus led to consider working with Ocaml classes.

The object-oriented part of the Ocaml language is a conservative extension of Caml. It is a strongly typed class-based language; the corresponding type system is based on ML polymorphism and type inference, and builds on extensible record types with polymorphic access. It provides most features of object-oriented languages including multiple inheritance, subtyping, methods returning self and parametric classes.

4.1 Programming by data-encapsulation

We first try to use object-oriented feature as it is usually done in textbooks on object-oriented languages: species are described by virtual classes, collections by concrete classes, and entities by objects (indeed by instance variables of objects). This simple representation does not meet our requirements; this can be seen on the following example, where `ring` species is described by:

```
class virtual ring =
object (self:'a)
  method virtual equal:'a -> bool    method virtual plus:'a -> 'a
  method virtual mult: 'a ->'a       method virtual opp: 'a
  method virtual one: 'a             method virtual print: string
end
```

Note that `one` and `opp` have the same types whereas their semantics are very different. Indeed, `opp` is the *"opposite"* operator and depends on the underlying entity, whereas `one` is the *"unity"* entity and does only depend on the underlying collection. Actually, types of functionalities do not reflect their arities, because functionalities are partially applied to the underlying entity, often denoted by `self` in object-oriented languages. In order to make `one` appear in the specification of `ring`, we have to make this constant depend on an implicit argument; such a trick is source of difficulties when proving properties. Similarly, binary operations become within this approach unary methods, which introduces a gap between the syntax and mathematical notation.

We come to another problem, illustrated by the following example. In Ocaml, instance variables, such as `my_rep`, are private, insuring data encapsulation. However, in coding the binary operations, one needs the actual internal representation, given here by `my_rep`, of the explicit argument. Therefore, the value of the instance variable has to be made public by a specific method, called below `rep`. This way, the integer collection may be given by:

```
class integers =
object
  inherit ring
  val my_rep = 0
  method rep = my_rep
  method plus x = < my_rep = my_rep + x#rep >
  method one = < my_rep = 1>
  method print = string_of_int my_rep
  ...
end
```

We then face a new problem, coming from the fact that `rep` is shared by all the sub-classes having the same carrier. A representation of entities is indeed often built up respecting an implicit invariant, which has to be preserved by the methods of the class. For instance, the entities of $\mathbb{Z}/2\mathbb{Z}$ may be canonically coded by `0` or `1`:

```
class z2z =
object
  inherit ring
  val my_rep = 0
  method rep = my_rep
  method plus x =
    let tmp=my_rep+x#rep in
      if tmp=2 then < my_rep = 0 > else < my_rep = tmp >
 method print = (string_of_int my_rep)^"[2]"
 ....
end;;
```

It is now possible to mix integers and modular integers, as follows:

```
# let one_z2z = (new z2z)#one ;;
val one_z2z : z2z = <obj>
# let one = (new integers)#one ;;
val one : integers = <obj>
#let three = (one#plus one)#plus one ;;
val three : integers = <obj>
# (one_z2z#plus three)#print;;
- - : string = "4[2]"
```

The implicit invariant has been broken, because the methods of `z2z` and `integers` have the same name and the same type, and thus are considered by Ocaml's typing system to be compatible, which should not happen. The only way to solve this problem and to fulfill the requirements is to modify the "`rep`" method; one can think of two ways of achieving this.

The first solution uses a convention on the names of the "`rep`" methods. For each collection, we choose a new name for "`rep`": `rep_`A is the `rep` of the collection A. Hence, `rep_integers` and `rep_z2z` will make `integers` and `z2z` incompatible types. Nevertheless, this solution will require some intricate typing management when dealing with parametric classes such as polynomials. It is therefore not very practical, and a bit dangerous; we hence reject it.

The second solution uses the type abstraction of the modules system (see 3). The idea is to define classes directly inside modules. This way, the concrete type of say the "`rep`" method can be easily hidden. It is still possible to define, outside a module, some subclasses of the class being encapsulated in this module. But, as the representation of the objects is hidden, there is not enough information about `rep` to define interesting new methods along the inheritance chain; there are several ways to patch this problem but they are just patches. This solution actually enlights a problem which was present since the very beginning of this naive development.

In conclusion, this "object oriented" solution does not fit our requirements for at least two reasons. Firstly, this kind of programming discipline is not very efficient: calling "`rep`" has a cost at running time, while in the "module" approach (in 3), data encapsulation is guaranteed by a typing mechanism without additional cost at run-time. Secondly, syntactical constructions implementing species and collections do not have a very clear mathematical semantics. For instance, the collection of integers is implemented by the class `integers` of this model; an object of this class is the coding of an entity of the collection, but carries within self the unity, and all the functionalities of this collection and of the species of rings. The relation object-class does actually not fit to the individual-society semantics; this is particularly visible in the loss of arity mentioned above, which makes it difficult to express properties like associativity or commutativity of operations. Therefore, this naive solution is incompatible with the will of programming using algebraic notions.

4.2 Towards ADT programming

Using the model we now turn to, we restore the correspondence between type and arity of operations. We proceed as follows:

```
class virtual ring =
object (self :'a)
  method virtual equal:'a*'a->bool    method virtual plus:'a*'a->'a
  method virtual mult:'a*'a->'a       method virtual opp:'a->'a
  method virtual one:'a
  ...
end

class integers =
object
  inherit ring
  val my_rep=0
  method rep_integers = my_rep
  method equal (x,y) = x#rep_integers=y#rep_integers
  method plus (x,y) = < my_rep=x#rep_integers+y#rep_integers>
  ....
end;;
```

This model has been developed up to the implementation of distributed polynomials. One of its interests is that it does not depend on the abstraction mechanism of module types. We however rejected it for the following reasons: within such a model, entities are still encapsulated in objects, still paying the cost of the calls to the `rep_`A methods. Moreover, the possibility of distinguishing between the object `integers` and the object `zero` is only given by a programming discipline, which will be very difficult to handle at the proof level. Indeed, one would in general be able to write silly comparisons like the following:

```
# let integers = new integers;;
val integers : integers = <obj>
# let zero = integers#plus (integers#one,integers#opp integers#one);;
val zero : integers = <obj>
# integers#equal (integers,zero);;
- - : bool = true
```

5 Chosen solution

5.1 Presentation

For collections and entities to have different types, we develop a new model, getting rid of the instance variable `my_rep`. The information on the carrier of the structure is now given as a type parameter and the `ring`, `integers` and `z2z` classes of the preceding models become:

```
class virtual ['a] ring =
object
  method virtual equal:'a*'a->bool
  method virtual plus:'a*'a->'a
  ...
end

class integers =
object
  inherit [int] ring
  method equal (x,y) = (x = y)
  method plus (x,y) = x + y
  ...
end
```

```
class z2z =
object
  inherit [int] ring
  method equal (x,y) = x=y
  method plus (x,y) =
    let tmp=x+y in if tmp=2 then 0 else tmp
  method mult (x,y) = x*y
  method opp x = x
  method one = 1
  method print x =
    (string_of_int x)^"[2]"
end
```

The `integers` class implements the mathematical integers. This class is a concrete one but can still be refined using inheritance. Thus, we may consider it as an implementation of the *species* of the integers, leaving place to define several different collections based on integers, defined by inheritance of the `integers` class.

The society of integers is described by a collection, which should provide access to its operations, while hiding the type of the carrier; it would be defined by a module `Integers`. Its signature shows that the carrier type is exported in an abstract way, through name `t`. We define the collection `Z2z` in the same way:

```
module type Ring = sig
  type t
  val meth: t ring
end
```

```
module Integers : Ring =
struct
  type t=int
  let meth=(new integers)
end
```

```
module Z2z: Ring =
struct
  type t=int
  let meth=(new z2z)
end
```

The calculations in these collections are performed using the `meth` field and the specification is well embodied in the types. A user may now declare an object, still called `integers`, which allows him to use the integer collection in a simple way:

```
# let integers = Integers.meth;;
val integers : Integers.t ring
# let one = integers#one;;
val one : Integers.t
```

```
# let z2z = Z2z.meth;;
val z2z : Z2z.t ring = <obj>
# let one_z2z = z2z#one;;
val one_z2z : Z2z.t = <abstr>
```

```
# z2z#plus (one_z2z,one);;
This expression has type Z2z.t * Integers.t but is here used with type Z2z.t *
Z2z.t
```

As shown by the previous examples, a collection A is represented via a module as a pair `(t, meth)` where `t` is the type of its entities and `meth` is the object that "contains" the methods of the collection. The representation of `t` should be known only by collection A and the collections extending it, while being hidden to all users of these collections. The classes like `integers` are only used as generators for collections; the access to the actual representation is still possible inside the class.

This mechanism can be easily extended to handle parameterized collections like polynomials. Furthermore, the double use of identifier `integers`, both as a class and as an object, suggests that this encapsulation of structures inside modules may probably be automated.

In this model, unlike the traditional way of programming in object-oriented style, an object does not have an internal state, that is there is no instance variable. The main point here is that the class is completely described by the functionalities of the species or the collection, in the same spirit as algebraic abstract datatypes. Consequently we coin the phrase "ADT model"; note however that in this model, the whole functional expressiveness provided by Ocaml is exploited.

5.2 Other features used

The hierarchy of classes currently consists of about fifty species: sets, ordered sets, lattices, additive monoids, multiplicative monoids, additive groups, different varieties of rings and algebras. The development of this hierarchy strongly builds on the previously detailed features: inheritance, virtual/concrete methods, delayed binding. We mention here other programming features that seem important to us.

Class variables Class variables, defined at the level of the class, are shared between all the objects belonging to it. They are notably useful for parameterized species, to perform operations on the parameter at the moment of its instantiation. For example, the species `algebra` described below uses the variable `under_minus_one`. In this example, the parameter `r` corresponds to the underlying ring and its type is `'r`. The parameters `'a` and `'b` correspond to the carrier type of the entities respectively of `r` and of the instantiations of the species `algebra`.

```
class virtual ['r,'a,'b] algebra (r:'r) =
  let under_minus_one = r#opposite r#one in
    object (the_alg)
      constraint 'r = ('a) #commutative_ring
      inherit ['b] ring
      method virtual mult_extern : ('a*'b) -> 'b
      method opp =
        let ( * ) x y  = (the_alg#mult_extern)(x,y) in
          function x -> under_minus_one * x
end;;
```

This species extend the notion of ring with an external multiplication `mult_extern`, which allows us to implement the `opp` method (declared in the definition of `ring`) using `under_minus_one`.

Recursive objects Let us study how recursive objects may be used for the recursive implementation of multivariate polynomials. We suppose here that we already have at our disposal a distributed implementation for multivariate polynomials, named `polynomial_algebra`, which generalizes the sparse representation for univariate polynomials. We recursively construct multivariate polynomials by iteration of the distributed representation. This way, a polynomial of $A[X,Y,Z,U]$ may be considered as a polynomial of $(((A[X])[Y])[Z])[U]$ or of $(A[X,Y])[Z,U]$. Let us call `recursive_polynomials` the class being defined. In this class, the following data structure represents the entities corresponding to recursive polynomials. The data structures types `'a` and `'b` represent coefficients and degrees respectively; a recursive polynomial is either a constant (`Base`) or a pair (`Composed`) consisting in a variable name (coded by a string) and a polynomial whose coefficients are themselves recursive polynomials.

```
type ('a,'b) rec_struct =
  | Base of 'a
  | Composed of string * ((('a,'b)rec_struct *'b) list);;
```

We exploit this recursion on the data structure to describe the simple methods of `recursive_polynomials`. For example, `mult_extern` computes the product of a polynomial and an element of the ring carrier:

```
method mult_extern =
  let rec up_mult = function
    | (a,Base b) -> Base (under_multiply(a,b))
    | (a,(Composed (v2,a2))) ->
        Composed(v2, (List.map (fun (v,d) -> (up_mult(a,v), d)) a2))
  in function (a,b) -> if under_is_zero(a) then zero else up_mult(a,b)
```

`under_is_zero` is here a class variable corresponding to the test of equality to zero in the ring carrier.

More complicated methods of `recursive_polynomials` will use the operations on distributed polynomials, for example:

```
method dist_coll = new polynomial_algebra (rec_rng,degs)
```

In this declaration, `degs` is the collection of exponents. `rec_rng` is the ring carrier of the distributed implementation, and hence is itself a recursive polynomial ring. Therefore, `rec_rng` is an object of the class being defined. Let us remark that `rec_rng` is indeed a recursive object. This `dist_coll` method can be used for coding operations such as printing:

```
method print = function
      | Base a -> under_print a
      | Composed (v,p) -> ((rec_rng#dist_coll)#output)(p,v)
```

The `output` method is defined in `polynomial_algebra`. It prints a polynomial, by giving a name to the variables and calling the `print` method of `rec_rng`. The `print` function carries out an iteration at every layer of the distribution through this single call to `output`.

Let us remark that we are not compelled, as is usually the case in Computer Algebra System, to consider recursive polynomials of n variables as univariate polynomials over polynomials of $n-1$ variables.

5.3 Suitability of the model

Let us now explain why the model we introduced corresponds to the schedule of conditions given in 2.1. The species are virtual classes: a group is a subclass of a monoid, and is hence actually defined as an enriched monoid. The different abstraction mechanisms we use (inheritance, parameterization, late binding) enable us to write in the definition of the species only what is required by the definition of the gender it implements. Despite of these dependences, the type system remains very strong up to the implementation of the collections. With this model, our goal of implementing algebraic structures can be completed successfully in a general purpose language. In doing this, we avoid building a dedicated language, and we can take advantage of numerous existing libraries.

By now, 50 species define the base of Computer Algebra. As a comparison, the Axiom basic library contains 46 categories. These 50 species need only 700 Ocaml lines whereas 1800 lines are needed in Axiom. Our running example is fully implemented and this was quite easy. The benchmarks are quite encouraging. They will be given in the full paper.

6 Conclusion

Designing the five prototypes we presented has been fruitful. We have an original development architecture, which mixes at best module-oriented and object-oriented features of Ocaml, in order to provide a system which is open, flexible, and powerful enough to express algorithms of Computer Algebra System. For instance, to our knowledge, the recursive representation of polynomials is more powerful than the one provided by other systems. In this paper, we have tried to assess, from our

experiments, the programming features which we need for this development, keeping in mind that certification has to be carried along the code generation. These features are now a schedule for the development of the proof part, as they have to be reflected in the architecture design of this part. Some preliminary results attest that it is indeed possible.

References

1. Guillaume Alexandre. *D'Axiom à Zermelo*. Thèse de l'université Paris 6, February 1998.
2. Coq project, *The Coq Proof Assistant Reference Manual*, version 6.2.4, 1999.
3. Judicaël Courant, *MC : un calcul de modules pour les systèmes de types purs*. Thèse de doctorat, École normale supérieure de Lyon, February 1998.
4. J. Davenport and Y. Siret and E. Tournier and D. Lazard *Computer Algebra*, Masson, 1993.
5. Richard D. Jenks and Robert S. Stutor. *AXIOM, The Scientific Computation System*. Springer-Verlag, 1992.
6. Xavier Leroy. *The Objective Caml system, release 2.0*. Software and documentation available: `http://caml.inria.fr/ocaml/`, 1998.
7. Didier Rémy. *Des enregistrements aux objets*. Mémoire d'habilitation à diriger des recherches en informatique. Université Paris 7, september 1998.
8. Didier Rémy et Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 1998, to appear.
9. David R. Stoutemyer. Crimes and Misdemeanors in the Computer Algebra Trade. *Notices of the AMS*, pp. 778-785, September 1991, Vol. 38, Number 7.
10. Jérôme Vouillon. Using modules as classes. In *Informal proceedings of the FOOL'5 workshop*, 1998. available at `http://pauillac.inria.fr/~remy/fool`
11. S. Watt, P. Broadbery, S. Dooley, P. Iglio, S. Morrison, J. Steinbach, et R. Sutor. *AXIOM Library Compiler User Guide*. NAG Ltd, March 1995.