Polymorphic Data Types, Objects, Modules and Functors, : is it too much?

S. Boulmé T. Hardin R. Rioboo

March 21, 2000

Abstract

Abstraction is a powerful tool for developers and it is offered by numerous features such as polymorphism, classes, modules and functors, ... A working programmer may be confused with this abundance. We develop a computer algebra library which is being certified. Reporting this experience made with a language (Ocaml [7]) offering all these features, we argue that they are all needed together. We compare several ways of using classes to represent algebraic concepts, trying to follow as close as possible mathematical specification. Then we show how to combine classes and modules to produce code having very strong typing properties. Currently, this library is made of one hundred units of functional code and behaves faster than analogous ones such as Axiom.

1 Introduction

Any software engineer is aware of the importance of *abstraction* in the process development. This concept corresponds in fact to several different methods. The first abstraction method is binding : binding an identifier in a type expression i.e. defining parametric polymorphism, binding an identifier in a value expression i.e. defining a function, binding a signature i.e. building a functor. The second method is to give a naming mechanism for collections of entities i.e. to offer objects, classes or/and modules, etc. The third one is a kind of hiding : hiding definitions of types (abstract data types, abstract/manifest types), hiding definitions of functions (interfaces/signatures), hiding names (private fields), etc. These three aspects of abstraction are provided by most modern languages. Some ones, as Ocaml, go a step further by offering polymorphic data types with records and unions, classes with multiple inheritance, modules and functors. Is it not too much? Faced with such a wealth of abstraction methods, which do not seem so far from each other, the programmer may be puzzled, when designing the implementation of a somewhat intricate specification.

In this paper, we relate our experience in the Foc project and we would like to explain why all these different handlings of abstraction are all needed together to satisfy the Foc requirements. In the following, we present briefly the motivations of Foc. Then we give a short acquaintance to its requirements, in order to justify the choices made in its conception.

The Foc[1] project, started at the fall 1997, is aimed to build a development environment for certified algebra, that is to say, a framework for programming algorithms, proving their mathematical properties and the correctness of their implementations. This aim may be a little surprising as, by definition, Computer Algebra Systems (in short CAS) work on mathematical entities represented by terms of a formal language, whose rules describe exact computations and algorithms (system or user-implemented) rely upon mathematical proofs. So little place seems to remain for bugs as, usually, implementations are carefully done. Despite of this care, bugs are not rare[10] :algorithmic errors (hasty simplifications, no verification of required assumptions, etc.), bugs during coding (incorrect typing, bad management of inheritance, bad deallocation, etc.). As CAS tend to be more and more used in critical systems (robotics, cryptography, physics, etc.), safety properties are required on their outputs. But, usual methods to guaranty a high level of safety are here rather difficult to use for the two following reasons. First, computer algebra programs tend to be large and complicated, and hence difficult to maintain. Then, testing symbolic manipulations may be difficult, due to the size of the data (for instance, polynomial coefficients with several thousands of digits) or the time needed for verification (several hours of CPU time computations is common). Furthermore, the ouput may be non-constructive, for instance, the result that a given polynomial has no root.

The Foc environment is based on a library of algebraic structures, which is providing not only the implementation of the classical tools to manipulate algebraic structures, but also their semantics, given by explicit verified statements. The user of Foc should have the possibility to specify a given algorithm by using together elements of this library, prove properties of this algorithm, define an implementation and prove its correctness. This needs a strong interaction between programming and proving, through user interface, which has been considered from the beginning of the project.

To increase safety, the gap between mathematical description of an algorithm and its encoding in the programming language has to be reduced. This requires a syntax powerful enough to reflect mathematical properties, as well as a firm semantics associated with this syntax. That was already pointed out several years ago by Davenport[5] and, as no programming language was meeting these requirements, the Computer Algebra community was led to develop its specific programming languages, giving birth to powerful systems, e.g. Axiom[6], which is perharps the most achieved. But this effort is not yet sufficient to get rid of bugs or ambiguities (for example, on solving multiple inheritance conflicts). Indeed, the syntax of Axiom encourages the user to follow a certain programming discipline but there is no effective semantic control. We tried[1] to prove some properties of Axiom programs by interfacing it with the proof assistant Coq. The conclusion was that such a task needs a programming language whose semantics is fully understood (and, possibly, formalized).

To decrease the distance between mathematics and code, to help carrying proofs, we made emerging the following requirements:

R1 The overall organization of the library should reflect its mathematical counterpart, e.g. groups should be defined upon monoids.

[1](F for Formel i.e. symbolic in French + O for Ocaml + C for Coq [3])

R2 Several levels of abstraction must be available for a given notion : the type of the operation of a group can be accessed before any implementation of it.

R3 Some notions may be defined *by default*, 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, `is_different` should be defined by default as the negation of `is_equal` in any structure built upon sets with equality but may be redefined within specific structures.

R4 Implementations of a given algebraic structure may be progressively refined: from an abstract view of $\mathbb{Z}/2\mathbb{Z}$, one may go to an implementation, ZI, representing the inhabitants by integers and to another one, ZB, using booleans. Some constructions have to be shared between ZI and ZB.

R5 Different implementations of an abstract algebraic structure, such that ZI and ZB, must be distinguished by typing, in order to avoid confusions or misuses.

R6 The correspondance between the coding of the structures in the programming language and the prover should be as natural as possible.

R7 To have a true prototype, the library should contain a significant amount of basic notions in Computer Algebra: big integers, modular integers, and several representations of polynomials, at least the distributed and the recursive ones. Indeed, 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.

These requirements are not all specific to Computer Algebra. they correspond to well-known paradigms in programming languages. The language must have a strong expressive power to meet `R1` and also to ease `R7`. `R2` together with `R5` asks for abstract data types on one hand and concrete (or manifest) types on the other hand, leading to modules. `R2`, `R3`, `R4` together call for object-oriented features (classes, inheritance, late binding). A functional programming style, free from assignments, but with exception handling helps for `R6`. `R7` needs also an efficient language, with recursive types and garbage collection.

Considering all these points, the language Ocaml was chosen. It 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.

This choice being done, the development is not yet ready to start. In fact, our requirements are in a certain sense contradictory. Indeed, `R5` asks to differentiate ZI and ZB by typing (module-oriented aspect), and at the same time `R4` asks to share some constructions between these two rings (object-oriented aspect). Thus, a design discipline for the definition of our library has to be elaborated, through the understanding of the balance between the use of module-oriented and object-oriented features. Doing that, we have also to describe the different dependency links between the library units in a rather uniform way, to ease forthcoming proofs.

To have an account of the actual difficulties arising from different design choices and of their possible solutions, we have written several versions of the basic library described by `R7`. These versions have been analyzed according to three criteria: whether they fit `R1-R7` 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.

In this paper, we first describe the conception of Foc in section 2, we then explain briefly in section 3 why a first try based only on modules was rejected and in section 4 we comment several ways of using the object-oriented features, focusing on their drawbacks. The retained solution is given in section 5, in a rather detailed way. The current state of the library is given in section 6 and some efficiency comparisons are done in section 7. Along the paper, we provide examples written in Ocaml, trying to remain understandable to people not acquainted with this language.

2 Analysis of the FOC's conception

In the following sections, we are commenting several ways for implementing the Foc library. These comments are done, according to the specification of this development, which is detailed in this section.

Engineers, scientists, etc. use day-to-day CAS like Mathematica and Maple for solving symbolic problems such as integration or equation solving, in much the same way that they would use a pocket calculator for numerical calculations. However some domains (robotics, cryptography, ...) require very more involved computations with CAS, needing well-engineered and robust libraries. Foc is not intended to be an interactive computer algebra system like Maple but only to offer a library, which can be used by engineers to build their own unit. However, a top-level system may be built upon it in the future.

The job of computer algebra engineers is to implement mathematics, more precisely, to implement tools which compute with mathematical data. Their aim is not to prove theorems with some verification tools but to produce data with algorithms built upon some theorems. The point where the approachs of engineers and mathematicians depart from each other is the notion of *representation* : in mathematics, there exists a unique set of integers defined by some caracteristic properties, in computer algebra, there are several implementations of integers (BigNums, GMP, etc.), sometimes needing explicit conversions between them. There are several ways to link mathematical data and their representations. In the following, we give our approach and we compare it with Axiom 'one.

2.1 Our vue

A CAS manipulates *entities* such as integers, polynomials, etc. These entities have a *representation* which must explicitely be stated as part of their definition. In our opinion it is important to distinguish mathematical operations performed on an entity from those performed on the representation. This give better control over the data being manipulated.

These entities are characterized by properties of their operations, which rely on mathematical properties together with representation properties. We also want to have a clear distinction between mathematical depencies and data representation depencies.

Thus, our choice is to have a neat separation between data manipulation and mathematical properties handling. Data manipulation is a concern of programming languages, which is not at all in the scope of computer algebra. For instance, lists with their tools and their properties are assumed to be available.

2.2 Categories and domains of Axiom

In Axiom, entities belong to some *domain*, which is presented as their type. Domains themselves belong to structures called *categories*. Categories are also presented as types of domains.

The membership of a domain to a category is asserted by a declaration and a domain may belong to different categories. Categories may be combined to build new categories by adding some components and by a join operation. So, categories and domains are akin to classes and objects, with an inheritance-like mechanism. The symbol % is used to denote the domain being described or implemented. It thus appears as the classical `self` of object-oriented languages. But it denotes also the abstract type of its entities. Categories export signatures containing names of available operations, with their prototypes written with %. These operations are not implemented at category level, but inside each of the domains of the category, using the representation for entities chosen in the domain. The representation of entities is always hidden outside their domain definition and can be manipulated only by the signatures of the categories the domain belongs to. Thus, with these signatures, a domain is like an abstract data type.

In Axiomxl (recent versions of Axiom) the two faces (abstract/concrete) of entities is explicited using two special functions `rep : % -> Rep` which gives access to the representation of the entity and `per : Rep -> %` which hides this representation. `Rep` is a conventional name to denote the representation of the entities. `rep` and `per` can be seen as conversions between abstract and manifest types.

2.3 Species and collections of Foc

Mathematical structures are here described by *species*, which are defined by a set of *components*, describing mathematical operations and properties available for an entity of this specie. So, species are roughly Axiom's categories. We detail carefully in the following the atomic steps of the introduction a new specie, as each of these steps corresponds to an atomic stage of proof correctness so needs to be easily identifiable in the source program.

The representation of its entities is the first component of a specie. it is called the *carrier* of the specie. Working in a polymorphic typed framework, the simplest carrier is a type variable T. T may progressively be instanciated by a type expression still containing other type variables or by an explicit data type. This is a first way of creating a new specie, which is called *carrier instanciation.*

The components, called *primitive*, of a specie are *named* and described by their *prototype*, written as a type expression possibly depending on T (or by a logical statement depending on T for components recording properties). A given specie can also have *derived* components which receive, beside a name and a prototype, an implementation build upon the primitive components (and functionalities supposed available over T).

A second way to create new species is to *extend* a given specie by adding primitive or derived components. For instance an additive group is an extension of an additive monoid by a primitive operation finding the opposite of an entity and another primitive one that checks an element to 0. From these operations one can describe a derived binary subtraction and implement equality (which was a primitive operation in the specie of monoids) in terms of subtraction and zero check. Sometimes, an extension adds only new properties: an abelian group has the same operations than a group but has new properties.

Now, a primitive component of a specie can receive an implementation, defining a new specie by a way usually called a *refinement* (so no extension of the specification, only a step to approach a full implementation). The code has only to meet the declared properties of the component. Thus the refinements of a specie share names, prototypes, some properties and some definitions.

A derived component, say c, of a given specie $\mathcal{S}_1$ may be *redefined*, leading to a new specie $\mathcal{S}_2$. As in the previous case, the new code has to meet the declared properties of the component in $\mathcal{S}_1$. Moreover, as redefinitions of a specie share also names, prototypes, and some properties, if some of these properties in $\mathcal{S}_1$ rely upon the code of c, they have to be reproved. Redefinition of primitive components is considered as well.

Whenever every primitive component of a specie has a definition, this specie can only be extended by derived components. We will call *collection* such a specie if we don't want to extend it anymore. A collection thus appears as a terminal element of the species creation process.

Species can receive parameters as long as those are collections or entities. Thus, a *parametrized specie* is a kind of "function" taking collections or entities and returning a specie. For instance, $\mathbb{Z}/n\mathbb{Z}$, the specie of modular integers, is parametrized by the integer n and there exists a specie of univariate polynomials, parametrized by the ring R of coefficients.

All previous operations on species apply to parametrized species. *Refining* a parametrized specie may be also done by instanciating some parameters. For instance, instantiating n by 2 builds the specie $\mathbb{Z}/2\mathbb{Z}$ and R may be instantiated by $\mathbb{Z}/2\mathbb{Z}$.

A specie $\mathcal{S}_1$ can be *converted* into a specie $\mathcal{S}_2$ by etasblihing a correspondance between the primitive components of $\mathcal{S}_2$ and some components of $\mathcal{S}_1$, ensuring the same properties. A specie can always be *restricted* to another specie of which it is an extension. Namely a field can always be provided where a ring is wanted.

Some operations of a specie can be *renamed* to create a new specie. For instance in an additive monoid we should be able to rename the "plus" operation into a "mult" operation and the "zero" constant into a "one" constant.

Summarizing, new species can be defined by representation instantiation, extension, refinement and parameter instanciation, redefinition. Moreover, conversion and explicit renaming are needed. These different links between species define a sort of hierarchy between them.

The introduction has described seven general requirements for Foc. Having an easy implementation for all these operations on species is also a major requirement. As already seen for `R1-R7`, these operations on species correspond to module-oriented or object-oriented features. Thus, we turn now to the description and the comparison of several attempts of designing the discipline coding of Foc.

3 Data encapsulation

The requirement `R5` of the introduction concerns safety : collections must be differentiated by typing, in order to avoid misuses and inconstencies. We focus first on this point.

As well-known, a simple way to increase safety is to export only abstract types for the representations. Indeed, data representation often uses implicit invariants. Abstract types forbid the user to misuse the representation by ignoring some of these invariants. Furthermore, the representation can be changed without disturbing users. On the other hand, developers of the units of the library need to know the exact implementation of the representation. So we fit in the very usual discussion on abstract versus manifest types and encapsulation of data representation. It firmly corresponds to a module-oriented programming style. Our first try was to use modules only and we comment it briefly.

We recall that Ocaml's modules system is a simply typed lambda-calculus language with a subtyping relation and constraint expressions. *Structures* allow to package together definitions sharing a common environment, which can be referred to, outside the structure, using the dot notation. *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 or to export some components with a restricted type. *Functors* are "higher-order functions" from structures to structures. So, abstract data types correspond to signatures and their implementations to structures. Species, such as `ring` below, are coded by module signatures and collections, such that $\mathbb{Z}/2\mathbb{Z}$, by module implementations.

```
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

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 abstraction mechanism is reinforced by the use of functors that allow to obtain the desired level of type abstraction within parameterized collections. For instance, the specie of univariate polynomials and the parameterized collection of sparse 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

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

The following declaration builds the collection of sparse polynomials with coefficients in $\mathbb{Z}/2\mathbb{Z}$.

```
module Pol_Z2z = Sp_Poly (Z2z);;
```

`Pol_Z2z.t` and `(Z2z.t*int) list` are incompatible types: the structures importing the sparse polynomials do not have access to their representation.

```
let id y = ...
val id : Pol_Z2z.t -> Pol_Z2z.t = <fun>
 id [(Z2z.one,1)];;
This expression has type (Z2z.t * int) list
but is here used with type Pol_Z2z.t = Sp_Poly(Z2z).t
```

But, `Pol_Z2z.Base.t` and `Z2z.t` are equal types. This testifies the correctness of the coding with respect to the specification.

So, Ocaml's modules system allows for an exact description of the specification and its powerful typing algorithm helps a lot to avoid inconsistencies. But, as stated by `R2`, some inheritance mechanism is needed. There is no such possibility in the current version of Ocaml. We tried to micmic it by hand but it turns out to be unrealistic when it comes to real-size attempts. Declaring inherited modules as components of heirs leads to a notation with a painful sequence of dots. Putting by "cut and paste" the components of the inherited module inside the heir is definitively too hard to maintain. Such a mechanism may perharps be automatized, but not so simply, as extending it to module structures and functors 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 use of them would not respect their purpose.

4 First runs with classes

Having a true inheritance seems to be a necessity; we are thus led to consider working with Ocaml classes. As stated by requirement R3, redefinition of certain operations is crucial, typically for optimization purposes. For instance the function `mult_extern`, which computes the product of a polynomial with a scalar number, may take advantage of the specific representation of sparse polynomials. Late binding allows to modify only those fields that need to be redefined and these modifications have not to be reported in the methods using these fields. So we firmly want late binding.

4.1 Entities as objects

We first try to use object-oriented features as 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). But this simple design does not meet our requirements; this can be seen on the following example, where the `ring` specie 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
end
```

The *"unity"* entity `one` and the *"opposite"* operator `opp` have the same type. Actually, types of components do not reflect their arities, because they are implicitly applied to the underlying object. For instance, binary operations become within this approach unary methods, which introduces a gap between the syntax and mathematical notation. Moreover `opp` applies to the underlying entity, whereas `one` is given by the underlying collection, these semantic differences are not reflected.

Now, instance variables, such as `my_rep` are private in Ocaml, insuring data encapsulation. However, when coding the binary operations, one needs to know the actual 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 plus x = < my_rep = my_rep + x#rep >
  method rep = my_rep
  method one = < my_rep = 1>
  ...
end
```

We then face a new problem, coming from the fact that `rep` is shared by all the sub-classes having the same carrier. For instance, the entities of $\mathbb{Z}/2\mathbb{Z}$ may be canonically coded by `0` or `1`, by using an implicit invariant.

```
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.

In conclusion, this "object oriented" solution does not fit our requirements. There is no neat correspondance between the implementation of species and collections and their 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 himself the unity, and all the operations of this collection and of the species of rings. The relation object-class does actually not fit to the relation entity-collection. This is exemplified by the loss of arity mentioned above, which makes difficult to express properties like associativity or commutativity of operations. At last, such a use of objects and classes is rather inefficient, due to the continuous use of "`rep`" at running time. Remember that data encapsulation within modules is guaranteed by a typing mechanism without additional cost at run-time.

4.2 Classes as Abstract Data Types

We now want to restore the correspondence between type and arity of operations by rending explicit the implicit argument "self". We give here to the type `'a` of the object the same status as % in Axiom. So, binary operations correspond to binary methods and constants appear as true constants. In other words, the representation of a specie or a collection is an abstract data type, encoded by a class.

```
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_ints = my_rep
  method equal (x,y) = x#rep_ints=y#rep_ints
  method plus (x,y) = < my_rep=x#rep_ints+y#rep_ints>
  ....
end;;
```

This concrete class `integers` may be viewed as a specie if we want still to refine it. It will be considered as a collection if the refinement process is frozen. But, how to interpret the values `my_int` and `zero`?

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

They have the same type, they may be compared. But, `zero` is clearly an entity wheras `my_int` may be intended as a collection. Distinguishing between these two possible uses of objects may be difficult at the proof level.

This model has been developed up to the implementation of distributed polynomials. We however rejected it as furthermore, entities are still encapsulated in objects, still paying the cost of the calls to the `rep_` methods.

5 Encapsulating classes within modules

To differentiate species, collections and entities by static typing, we develop a new model, which also gets rid of the instance variable `my_rep`. The information on the carrier is now given as a type parameter `'a` and has the same status than Axiom's `Rep`. Methods have types depending on the carrier and not on an abstracted view of the representation. For instance, 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 concrete but can still be refined using inheritance. We consider it firmly as the *specie* of the integers. More generally, classes are considered only as implementations of species. Collections are always created only by using the keyword `new`. So, they are Ocaml objects. Entities are simply elements of the carrier of the specie specifying the collection. Applying operations to entities is sending a message to the object (collection).

Whit this choice, we have a one-to-one correspondance between mathematical notions and semantics of typing. But, the problem of carrier abstraction described above remains. It is due to the powerful mechnism of subclassing and cannot be solved within the object-oriented framework. To handle this problem, we propose the following solution.

As defined above, the collection of integers provides access to its operations but also to its carrier. This is fine for, for example, to pass them as actual parameters to parameterized species. But, common uses of the library do not need full access to the carrier. So it is safe to add an encapsulation mechanism, building structures called *E-collections*.

An E-collection is obtained as follows:

```
module type E_collection = sig
  type abstract
  val a_collection: abstract type_class
end
```

```
module my_E_collection : E_collection =
struct
  type abstract= some_type
  let  a_collection  = (new a_class)
  end
```

For instance, we define the E-collection Z2z by :

```
module type Ring = sig
  type abstract
  val a_ring: abstract ring
end
module Integers : Ring =
struct
  type abstract=int
  let a_ring =(new integers)
end
module Z2z: Ring =
struct
  type abstract=int
  let a_ring  =(new z2z)
end
```

Calculations are performed using `a_ring` and the specification is type safe. 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.a_ring ;;
val integers : Integers.abstract ring
# let one = integers#one;;
val one : Integers.abstract
# let z2z = Z2z.a_ring ;;
val z2z : Z2z.abstract ring = <obj>
# let one_z2z = z2z#one;;
val one_z2z : Z2z.abstract = <abstr>
# z2z#plus (one_z2z,one);;
This expression has type Z2z.abstract * Integers.abstract
but is here used with type Z2z.abstract * Z2z.abstract
```

As shown by the previous examples, an E-collection A is represented via a module as a pair `(abstract, some)` where `abstract` is the type of its entities and `some` is the object that "contains" the methods of the collection. The representation of `abstract` should be known only by the species underlying A and the collections extending it, while being hidden to all users of the corresponding E-collection. This mechanism can be easily extended to handle parameterized collections like polynomials.

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. Note however that in this model, the whole functional expressiveness provided by Ocaml is exploited.

6 Description of the library

In its current state the Foc library is made of about 100 Ocaml class for about 3500 lines of Ocaml code, comprising :

- parameterized species by base integers, which encapsulate small and big integers. Currently small integers are used to build degrees of polynomials and small modular arithmetics. Big integers are used as coefficients rings of those polynomials. Support is provided using two different big integer packages : BigNum and GMP[2].

- Base species to provide monomials and ordering over those monomials. Current implementation supports several variables with lexicographical ordering providing "degree arithmetic". The usual case of one variable is then seen as a special (degenerated) case of this.

- Distributed polynomial arithmetics is then provided up to exact division.

This code achieves most of the functionalities of Axiom's polynomials but with increased reusability since in Axiom univariate and distributed polynomials have different (though similar) implementations.

We then provide support for recursive polynomials with strictly higher generality than those of Axiom. In Axiom recursive polynomials are an iteration of the univariate case, viewing a polynomial in X and Y as a polynomial in X which coefficients are polynomials in Y. The carrier is a recursive type, the base case is given by the coefficient ring and the inductive case uses the distributed polynomials specie. Suppose given a a ring collection R with carrier α (or `'a`) and a degree collection D with carrier β (or `'b`), then the carrier for recursive polynomials is :

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

The type of recursive polynomials is denoted by γ_r for short. Here the parameter of type `string` is used to represent the set of variable names of the multivariate polynomial and to define the level ordering.

Building the specie of recursive polynomials, we have to express that the collection R (or `r`) is a ring and that D (or `d`) is a degree collection. Here R and D are Ocaml object values with types ω (or `'r`) and δ (or `'d`) respectively. We achieve this by writing type constraints :

$$\texttt{constraint } \omega = (\alpha)\texttt{\#ring} \text{ and}$$
$$\texttt{constraint } \delta = (\beta)\texttt{\#monomial\_ordering}$$

Now, recursive polynomial operations usually proceed by calling univariate operations. We depart from that by calling distributed operations.

We thus need a specie D_p (or `distr_p`) for distributed polynomials (not detailled here).

Let us name by R_r (or `rec_p`) the object being defined in the class `recursive_pols`. Its type is ω_r (or `'rec_p`). We define it as a collection, obtained from an instantiation of the specie D_p, giving R_r as the actual coefficient parameter collection and D as the actual degree collection. We thus need R_r to be a ring with carrier γ_r :

[2]other packages are being included

$$\texttt{inherit } (\gamma_r)\texttt{\#ring}$$

and we can now hold the collection $D_p(R_r, D)$ by defining a method `the_dp` :

$$\texttt{method the\_dp = new } D_p(R_r, D).$$

We thus have :

```
class ['r,'a,'d,'b] recursive_pols ((r,d):('r,'d)) =
object(rec_p:'rec_p)
  constraint 'r = ('a)#ring
  constraint 'd = ('b)#monomial_ordering

  inherit [('a,'b)rec_struct]ring
  method the_dp = let dp = new distr_p(rec_p,d) in dp
  ...
end
```

The method `the_dp` can now be used inside other method bodies to encapsulate distributed opertions. For instance the code for recursive polynomial multiplication uses the construction :

```
let ( * ) p q = (rec_p#the_dp)#mult(p,q) in ...
```

Note that late binding and open recursion is essential to this process. Current Foc implementation uses further abstraction by manipulating a function $F_p = R_r \leadsto D_p(R_r, D)$ and uses an effective collection constructor and further type parameters which abstract the effective D_p implementation of ditributed polynomials.

7 Benchmarking

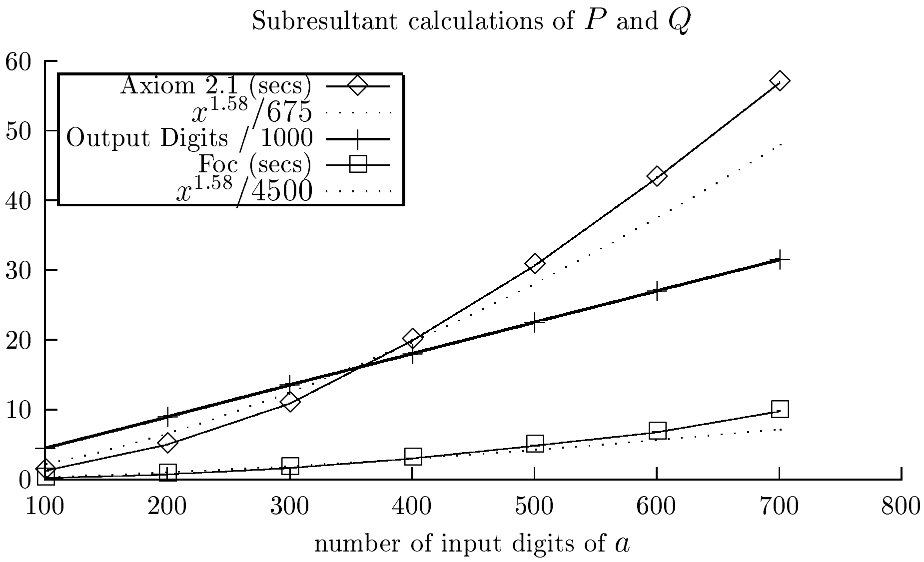
Coding with the Foc library uses functional style programming. Most of Foc's code does not overwrite derivated components and many operations use their default implementation which induces an extra cost. Representations used in Foc are close to those of Axiom though they are strictly more general. It thus make sense to compare Foc with Axiom, wheras it would not with other computer algebra systems.

The benchmark consists in resultant computations (they are determinant of matrices that are computed using polynomial arithmetics). Operations involved in the coefficient ring are addition, multiplication and exact division. The same algorithm has been coded using Foc and Objective caml 2.04 and Axiom 2.1. The two univariate polynomials involved are $P = x^{30} + ax^{20} + 2ax^{10} + 3a$ and $Q = x^{25} + 4bx^{15} + 5bx^5$ with a varying and $b = a + 1$. The results are obtained on a pentium 450Mhz machine running redhat linux 5.2. Timings are computed by the Ocaml `Sys.time` function. In Axiom 2.1 time statistics are unreliable in the presence of garbage collections and we designed our own timing function from the basic timer of the underlying Lisp system. Both timimgs take GC activity into account.

The first bench[3] in figure 1 mesures big integers capabilities and a is an integer in the range 10^{100} to 10^{700}. The result is also an integer with 4500 to 31500 digits in base 10.

[3]dashed lines represent theoretical complexity for Karatsuba multiplication which appears not to be implemented in either big integer packages

Figure 1: subresultant calculation of P and Q



Objective Caml has the ability to produce byte or native code, in the previous computation both timings cannot be distinguished since most of the time is spent inside big integer calculations. We thus can see that Axiom's big integers are less efficient than Ocaml.

In figure 2 we change the coefficient ring and a will be a polynomial of the form $\sum_{i=0}^{i=k} A^i$. The result is a polynomial in A of degrees varying from 45 to 315. We measure this degree, the maximum size of the result's coefficients (which are integers) together with times. Here the size of coefficients varies from 1 to 72 digits in base 10.

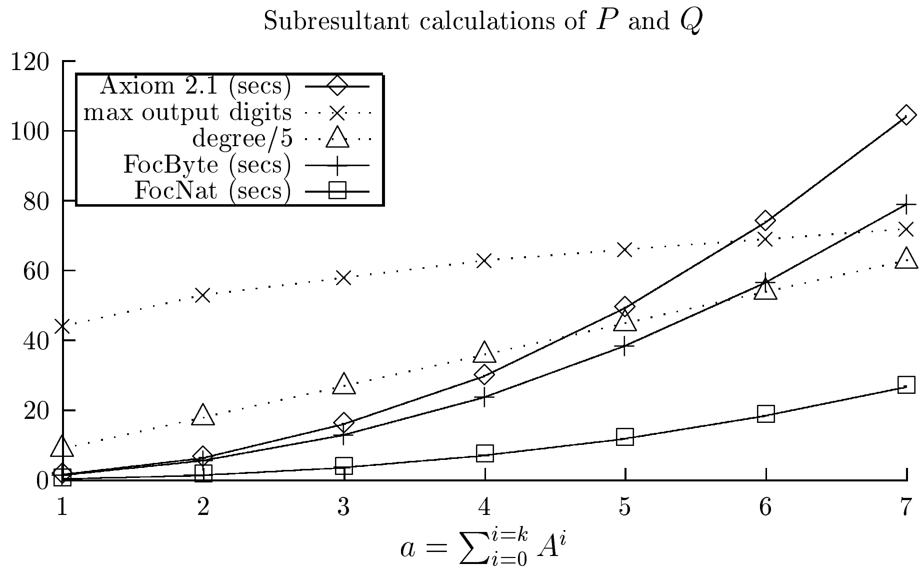
Here numbers are relatively small and time spent inside big integer arithmetic is negligible with respect to the time spend inside polynomial arithmetics (polynomials in A). We can see that timings compare for the byte code version of Foc, and are much better for the native code version of Foc.

8 Conclusion

The title of this paper expresses well our questioning at the start of the Foc project. Carrying out several sizeable prototypes, we have been able to elaborate an answer, which shows that all the abstract methods offered by our programming idiom are needed. Some other languages offer a mechanism inheritance within modules. For instance, mixins[2] are modules in which some components are deferred i.e. their definition has to be provided by another module. They can be mutually dependent and their composition supports redefinition of components. But, this is not enough as late binding, not only overidding, is also crucial.

What we have done in fact is to design a framework well-adapted to the specification of a given trade : the one of the computer algebra engineer. And we think that this experience can be redone with another trades (chemistry, physics, etc.), leadind perharps to very different

Figure 2: subresultant calculation of P and Q



uses of abstract methods. So, as a first conclusion, this is indeed important to dispose of a programming language with rich features, but, only if they are semantically well-understood so if they can serve to express without ambiguity the specifications of a given area.

As a second conclusion, we may say that, 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 have been completely irrealistic to try to create our own programming language. As it is a semantically well founded language, Ocamlis a good compromise. Using only a functional style certainly will help the proofs to be done. Also, the richness of the syntax allows to code algorithms very closely to their mathematical formulation. This will help also proving stages.

The third conclusion may be on efficiency. We have noted that the encapsulation of data inside objects is really costly. But, there is no need to use it. On the opposite, me may claim that functional style is efficient, more efficient in this case than traditional implementations making fine tuning of pointers.

As shown by the number of classes and the benches, the library has now reached the state of a full development. Our design conception has been tested by students which have added some units, following it without difficulties. On the side proof, the major difficult point is to define the representation of species, collections and of the different operations on them. A solution, based on dependent labelled records coded in Coq is under submission. The next step of the project is to define the user interface, that is, a syntax for programs and statements, well-adapted to computer algebra engineers and to extract Ocaml code and Coq code from it.

Acknowledgments

We would like to acknowledge the Coq and Cristal teams at INRIA for the languages Ocaml and Coq. This work has benefited of numerous discussions with V. Ménissier-Morain, D. Hirschkoff, D. Doligez, M. Moreno, V. Viguié, the members of SPI and CALFOR teams at LIP6, the members of the CFC action at INRIA.

References

[1] G. Alexandre. *D'Axiom à Zermelo.* Thèse de l'université Paris 6, February 1998.

[2] D. Ancona and E. Zucca *An algebraic approach to mixins and modularity.*. In M. Hanus and M. Rodrigez Artalejo, editors, *ALP'96*, LNCS 1139, Springer Verlag.

[3] Coq project, *The Coq Proof Assistant Reference Manual*, version 6.2.4, 1999.

[4] Judicaël Courant, *MC : un calcul de modules pour les systèmes de types purs.* Thèse de doctorat, École normale supérieure de Lyon, February 1998.

[5] J. Davenport and Y. Siret and E. Tournier and D. Lazard *Computer Algebra*, Masson, 1993.

[6] Richard D. Jenks and Robert S. Stutor. *AXIOM, The Scientific Computation System.* Springer-Verlag, 1992.

[7] Xavier Leroy. *The Objective Caml system, release 2.0.* Software and documentation available: `http://caml.inria.fr/ocaml/`, 1998.

[8] Didier Rémy. *Des enregistrements aux objets.* Mémoire d'habilitation à diriger des recherches en informatique. Université Paris 7, september 1998.

[9] Didier Rémy et Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 1998, to appear.

[10] David R. Stoutemyer. Crimes and Misdemeanors in the Computer Algebra Trade. *Notices of the AMS*, pp. 778-785, September 1991, Vol. 38, Number 7.

[11] Jérôme Vouillon. Using modules as classes. In *Informal proceedings of the FOOL'5 workshop*, 1998. available at `http://pauillac.inria.fr/~remy/fool`

[12] S. Watt, P. Broadbery, S. Dooley, P. Iglio, S. Morrison, J. Steinbach, et R. Sutor. *AXIOM Library Compiler User Guide.* NAG Ltd, March 1995.