Mathematical Knowledge Management in Foc

T. Hardin[1]

Laboratoire d'Informatique de Paris 6 (LIP6),
Université Pierre et Marie Curie,
8, rue du Capitaine Scott, 75015 Paris, France.
INRIA, Projet MOSCOVA
Rocquencourt, 78153 Le Chesnay CEDEX, France.
Therese.Hardin@lip6.fr

Abstract. We present a library which enables to implement general computer algebra notions called here entities, collections and species without the help of a dedicated language. We give an operational model for these notions, which has been formalized and verified using the Coq proof assistant. We rely on this model to design the programming discipline of the library. This allows us to produce semantically founded programs in a general purpose language (Ocaml). Thus, a lot of programming features such as polymorphism, type inference, data structure management, garbage collection are obtained for free while keeping good efficiency when running computer algebra algorithms. The discipline corresponds to a restricted use of "object oriented" together with "module" features, which is formalized in the theorem prover Coq. Following the discipline is enforced by the concrete syntax given to users.

The Foc[1] project started at the fall 1997 is currently building a development environment for certified computer algebra. That is a framework for programming algorithms, proving their mathematical properties and the correctness of their implementations. This talk gives a presentation of this project, focusing on the mathematical knowledge management within a general purpose programming language.

Computer Algebra Systems (CAS in short) perform *exact* (or symbolic) computations on mathematical entities which are represented by terms of formal language. Moreover, the correctness (and the mathematical meaning) of the algorithms underlying these computations have been strictly established, by mathematicians. Thus, no place seems to be left for bugs, because implementations are usually carefully done. Despite of this care, bugs are not rare [16]: algorithmic errors (hasty simplifications, required assumptions which are actually not discharged, ...), implementation errors (incorrect typing, bad management of inheritance, bad deallocation, ...).

To increase safety, the gap between mathematical description of an algorithm and its implementation has to be reduced. This requires a syntax powerful enough to reflect mathematical properties, as well as a strong semantic associated with this syntax. That was already pointed out several years ago in [10] and, since no programming language was meeting these requirements, the Computer Algebra community was led to develop its specific programming languages, such as the powerful systems Axiom([12]) or Magma ([4]). 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 firmly stated (and possibly formalized).

In parallel with this effort of the Computer Algebra community, research on semantics of programming languages has led to powerful languages such as Ocaml that can combine reliability and high level programming with efficient programming. Such languages allow many styles of programming (applicative, imperative, object-oriented, abstract data types...). The semantics of programs using together these different features has been carefully studied (reliability of the language), and efficient compilers have been developed. We claim in this talk that it is now possible to use such a "general" language to program computer algebra libraries.

However, to use a language having a well-studied semantics is not enough to build a *certified* CAS. Indeed, the user of Foc should have the possibility to specify a given algorithm, to prove its properties, to implement

[1] F for formel i.e. symbolic in French+O for Ocaml+C for Coq[8]

it, showing the correctness of her/his code. As Ocaml provides static (polymorphic) typing and subclassing, some semantic properties of algorithms and programs can be granted for free. Some of these properties are not so trivial, has it will be illustrated during the talk. But, some required formal specifications and proofs are much involved. Thus, we intend also to describe the contents of the library in the theorem prover Coq. The structure of the library has been already formalized in Coq and modelized in a Category theory based on dependent types[5]. The heart of the Foc environment is a user-interface that extracts from the user source both a Ocaml program and Coq specifications. Then, the proof is to be done at the user-interface level and translated into a proof term, checked by Coq. This last part of the user-interface is currently under design[11]. In the prototype of this interface, proofs are done directly in Coq.

1 Semantic Analysis

The point where the approaches 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 characteristic properties. In computer algebra, there are several implementations of integers (BigNums, GMP, etc.), sometimes needing explicit conversions between them. There are several ways to handle the correspondence between a mathematical object and its representations.

A CAS manipulates *entities* such as integers, polynomials, etc. These entities have a *representation* which must explicitly 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 both on mathematical properties and on representation properties. It is important for us to have a clear distinction between mathematical depencies and data representation depencies.

Thus, our choice is to have a clear separation between data manipulation and mathematical operations. 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. Thus our choice is to write a library whereas Axiom defines a complete computer language. We have to live with the types and values of our implementation languages (Ocaml and Coq), whereas a complete language should define its own types and values, together with a fully stated (and if possible formalized) typing and evaluation semantics.

1.1 Terminology of the project

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 species. So, species are roughly Axiom's categories. We detail carefully in the following the atomic steps of the introduction a new species, 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 species. It is called the *carrier* of the species. Working in a polymorphic typed framework, the simplest carrier is a type variable τ. This may progressively be instantiated by a type expression still containing other type variables or by an explicit data type. This is a first way of creating a new species. We call this *carrier instantiation*.
– The components, called *primitive*, of a species are *named* and described by their *prototype*, written as a type expression possibly depending on τ, or by a logical statement depending on τ for components recording properties. A given species can also have *derived* components which receive, besides a name and a prototype (or a statement), an implementation (or a proof) build upon the primitive components and functionalities supposed available over τ.
– A second way to create new species is to extend a given species 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 species of monoids) in terms of subtraction and zero check. Sometimes, one adds only new properties: an abelian group has the same operations than a group but has new properties. We call the introduction of a single new component an *extension*.

- Now, a primitive component of a species can receive an implementation, defining a new species 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 species share names, prototypes, some properties and some definitions.
- A component, say c, of a given species $\mathcal{S}_1$ may be *redefined*, leading to a new species $\mathcal{S}_2$. 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 species 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.
- Whenever every primitive component of a species has a definition, this species can only be extended by derived components. We will call *collection* such a species 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 species* is a kind of "function" taking collections or entities and returning a species. For instance, Z/nZ, the species of modular integers, is parametrized by the integer n and there exists a species of univariate polynomials, parametrized by the ring R of coefficients.
- All previous operations on species apply to parametrized species. The operation, called *parameter instantiation*, is a way of building a new species. For instance, instantiating n by 2 builds the species $Z/2Z$ and R may be instantiated by $Z/2Z$.
- A species $\mathcal{S}_1$ can be *converted* into a species $\mathcal{S}_2$ by etasblishing a correspondence between the primitive components of $\mathcal{S}_2$ and some components of $\mathcal{S}_1$, ensuring the same properties. A species can always be *restricted* to another species of which it is an extension. Namely a field can always be provided where a ring is wanted.
- Some operations of a species can be *renamed* to create a new species. 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 a composition of atomic steps: carrier instantiation, extension, refinement, parameter instantiation, redefinition, conversion, restriction, renaming. For instance, a mechanism like multiple inheritance can be decomposed into restrictions leading to a common ancestor followed by some extensions.

These different operations induce links between species defining a hierarchy between them. The specification of this hierarchy and of these operations has been studied in [5, 6]. These operations are compatible with standard Ocaml's inheritance (see [14] or [15]).

2 Choosing an object-oriented model

We have considered various models based on object and classes to implement these notions inside the Ocaml programming language. These are summarized in figure 1 and we discuss them on a simplified version of our basic operations. We consider the representation component as distinguished from other components which are just *operations*, their protoype is given by a method name together with a signature.

The class model of figure 1 which encodes entities as objects and collections as classes was rejected because it was too far from "mathematical specifications". For example, arities of operations are not explicit so have to be inferred when proving properties. In the talk, we give a short example to illustrate that point.

The rep(resentation) model of figure 1 restaures operation arities but still encodes entities as objects. This leads to a permanent encapsulation of the representation inside an instance variable, which needs functions `rep_of` and `of_rep`) to manipulate these representations. This introduces a semantic confusion between the collection itself (for example, a group) and its elements. We shall explain this point during the talk.

We are thus led to a strictly more general class based model (the class model of figure 1) which gets rid of `rep_of` and `of_rep`. This model *replaces data encapsulation by data type abstaction*, using type parametric classes. In this model the carrier is explicit, carrier instantiation corresponds to type instantiation. Primitive

Fig. 1. Object based paradigms

	class model	rep model	object model
entity	object	representation	arbitrary
operation name	method	method	method
collection	class	class	object
prototype	method type	method type	method type
species	class type	class type	class

components correspond to virtual methods whereas derived components correspond to implemented methods. We shall give several examples along the talk.

3 Description of the library

We follow the paradigm of the object model where classes are species and where objects carry a "mathematical view" on a data structure which is always explicit. So, we consider that a class Γ views a data structure α as a mathematical structure $\mathcal{A}$. In this model an object is a collection and a class is a species.

3.1 Basic structures

In order to validate our model, we designed first a small, but meaningful, library of units covering common abstract algebra and common computer algebra basic notions. We focused on polynomials arithmetics, trying to achieve a good level of absraction. The current library is made of around 120 parametric species for around 4000 lines of code. It covers most of Axiom's functionalities for polynomials. The library provides concrete classes which enable to do integer arithmetics using either standard `big_int` big integers or the standard big integers Gnu package `GMP` (in versions 2 or 3 through the use of a `C` interface). We also provide support for small integers (for degree arithmetics) through the standard Ocaml integers `int`. In the talk, we will comment on the building of this library, focusing on the following points.

In Ocaml, classes can have two kinds of parameters, value parameters and type parameters. We use both, as shown by the following species corresponding to Axiom's category `Algebra`.

```
class virtual ['r,'a,'b] algebra(R:'r) =
object(the_alg)
  constraint 'r = ('a #commutative_ring)

  inherit ['b]ring
  method virtual mult_extern :
    ('a * 'b) -> 'b
  method lift x =
    the_alg#mult_extern(x , the_alg#one)
  method opposite x =
    the_alg#mult_extern((R#opposite)(R#one) , x)

end;;
```

The carrier of this algebra is encoded by the type parameter `'b`. The ring R of this R-Algebra is given by the value parameter `R` and his type by the type parameter `'r`. The carrier of this ring is encoded by the type parameter `'a`.

The type constraint expresses that `'r` (the type of the value parameter `R`) is a collection which should have the methods of the species of commutative rings with carrier `'a`. This constraint is verified by the type checker. That is the type checker infers the most general type ρ for the value parameter `R` and verifies that the type constraint defines an instantiation of ρ.

Here we see that to describe an R-Algebra we had to use three type parameters `'r`, `'a`, `'b` and one value parameter `R`. Moreover there is a dependencie between the value parameter `R` and the type parameter `'r`. More generally a species with n collection parameters (so value parameters) is described by a class with $2n+1$ type parameters in addition to the n value parameters. The carrier of the species being described is encoded by one type parameter. Moreover, for each value parameter P, we have one type parameter C_p encoding the carrier of P and a second type parameter S_p for the species that describes the operations required for P. This seems rather heavy but it reflects only mathematical coherence. And, it has the advantage that the type checker can verify the mathematical constraints expressed by the user. Roughly speaking, there is no 'type lie'.

Passing entities as argument to species is easy. We show that through the example of modular integers.

```
class modular_integers(the_mod) =
object
  inherit [int]commutative_ring

  method is_zero(x) = (x = 0)
  method one = 1
  method add(m,n) =
    let r = (m + n ) in
    if (r < 0)
    then foc_error "Overflow"
    else (r mod the_mod)
  ...
end;;
```

Here the value parameter `the_mod` is simply `int`, as infered by the system. More generally, suppose that a class is parameterized by an entity `v` of a collection `C`, specified by a species `S`. Then, `v` appars as a value parameter. C is introduced by a second value parameter and two additional type parameters, `'a` for the type of `v` and `'c` for the type of C. Access to the methods of C is given by the constraint :

`constraint 'c = 'a # S`

This constraint enables a type safe use of the mathematical operations of species `S`. It expresses that the type `'a #S` is more specific than the type `'c`.

So, the correctness of the coding of species, collections and entities relies on a "programming discipline", ensuring some specification invariants. This is enforced by the use of a concrete syntax which is translated to Ocaml code under this programming discipline (see [11] for the first attempts).

4 Conclusion

In this talk, we present a library which enables to implement general computer algebra notions called entities, collections and species without the help of a dedicated computer algebra language. An operational model for these notions enables us to produce safe (semantically founded) programs in the Ocaml general purpose language using a programming discipline. Furthermore, this informal model is formalized and verified using the Coq proof assistant. The programming discipline makes a restricted use of object oriented features but uses the full power of the class sub-language.

As a conclusion, we can affirm that it is possible to write generic computer algebra algorithms, in a general programming language, with a functional style, while keeping good efficiency when running them. But, the programming language must be very carefully chosen and must offer powerful, semantically understood, features among them strong typing, semantically studied multiple inheritance, late binding and a mechanism of abstract/manifest types.

The Foc environment is still in infancy but is promising. A concrete syntax, still under design, obliges the user to follow the programming discipline through a translator to Ocaml code and through a translator into formal specifications in Coq. Interface with Open-Math is also currently under design. When mature, it would enable to produce certified implementation of many computer algebra algorithms. The first α-version will be disposable at the end of 2001.

References

1. G. Alexandre. *D'Axiom à Zermelo.* Ph. D. Thesis, University of Paris 6, LIP6, 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. D. Ancona and E. Zucca. An algebra of mixin modules. In *WADT'97*, LNCS 1376. Springer-Verlag, January 1998.
4. W. Bosma, John J. Cannon, C. Playoust. *The Magma algebra system. I. The user language.* Journal of Symbolic Computation, Vol 24, 1997.
5. S. Boulmé. *Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de calcul formel.* Ph. D. Thesis, avaliable in french on `http://www-spi.lip6.fr/~boulme/bin/these.ps.gz.cgi`, Université Paris 6, Décembre 2000.
6. Sylvain Boulmé. Specifying in Coq inheritance used in computer algebra. Research report, available at `http://www.lip6.fr/reports/lip6.2000.013.html` 013, LIP6, May 2000.
7. S. Boulmé, T. Hardin, D. Hirschkoff, V. Ménissier-Morain, and R. Rioboo. On the way to certify computer algebra systems. In *Calculemus 99 Systems for Integrated Computation and Deduction*, volume 23 of *ENTCS*. Elsevier, 1999.
8. Coq project, *The Coq Proof Assistant Reference Manual*, version 6.2.4, 1999.
9. Judicaël Courant, *MC : un calcul de modules pour les systèmes de types purs.* Thèse de doctorat, École normale supérieure de Lyon, February 1998.
10. J. Davenport, Y. Siret, E. Tournier, D. Lazard *Computer Algebra*, Masson, 1993.
11. V. Prevosto. *Vers une interface utilisateur pour Foc.* Internal Report LIP6, October 2000.
12. R. D. Jenks, R. S. Stutor. *AXIOM, The Scientific Computation System.* Springer-Verlag, 1992.
13. X. Leroy. *The Objective Caml system, release 2.0.* Software and documentation available: `http://caml.inria.fr/ocaml/`, 1998.
14. D. Rémy. *Des enregistrements aux objets.* Mémoire d'habilitation à diriger des recherches en informatique. Université Paris 7, september 1998.
15. D. Rémy, J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 1998.
16. D. R. Stoutemyer. Crimes and Misdemeanors in the Computer Algebra Trade. *Notices of the AMS*, pp. 778-785, September 1991, Vol. 38, Number 7.
17. J. Vouillon. Using modules as classes. In *Informal proceedings of the FOOL'5 workshop*, 1998. available at `http://pauillac.inria.fr/~remy/fool` See also Ph.D. thesis.
18. S. Watt, P. Broadbery, S. Dooley, P. Iglio, S. Morrison, J. Steinbach, et R. Sutor. *AXIOM Library Compiler User Guide.* NAG Ltd, March 1995