Some hints for polynomials in the FOC project

S. Boulmé, T. Hardin, 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.
Sylvain.Boulme, Therese.Hardin, Renaud.Rioboo@lip6.fr

Abstract. 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.

1 Introduction

The FOC project started in 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 is a long-term project as its aims may seem rather ambitious. Indeed to ensure the correctness of implementation of mathematical algorithms one needs to formalize the underlying mathematical theories, to formalize the semantics of the different programming constructions and to create tools for proofs. We do not want to embrace all mathematics at a time and we focus first on computations on polynomials, choosing the sub-resultant algorithm as our running example. In this paper, we report on our implementation on polynomials, trying to explain how our objectives of certification have influenced our choices.

Computer Algebra Systems (CAS in short) perform exact (or symbolic) computations on mathematical entities which are represented by terms of a formal language. The correctness (and the mathematical meaning) of the algorithms underlying these computations are ensured by the proofs given by mathematicians. Despite this care, bugs are not so rare [18]: algorithmic errors (hasty simplifications, required assumptions which are not actually discharged, etc.), implementation errors (incorrect typing, bad management of inheritance, bad deallocation, etc.). Indeed, data manipulated by computer algebra programs are quite huge (polynomial coefficients with several thousands of digits), computations may be long (several hours of CPU time is common). Once the proofs of

1 F for formel i.e. symbolic in French, O for Ocaml, C for Coq[6]
algorithms are written down, there remains a lot of work to choose the appropriate data structures and coding of algorithms. For example, even if explicit manipulation of pointers is recognized to be error-prone (it is banished for strongly critical software), it is often used inside CAS to improve computation times or data representation management.

In software engineering, it is now admitted that formalizing specifications and proving required properties directly on these specifications is an efficient way to increase safety. Then a careful coding in a semantically sound (part of a) programming language is required and correctness proofs on code reinforce confidence. Parts of the code may also be extracted automatically from this formalization. To follow the same trail, besides the formalization of algebraic structures, we have to reduce the gap between the mathematical abstract description of an algorithm and its effective implementation.

It is rather easy to describe formally a given mathematical structure (i.e. a set endowed with operations and properties). But, deciding on what are the primitive notions and what are the derived ones depends of mathematical habits. This may have a very practical influence on further proofs (see for example the formalization of partiality done in Coq[9]).

Often, this mathematical structure depends on previously defined ones, leading to the need for inheritance mechanisms, which have to be semantically described, up to the question of late binding, allowing the user to replace an inherited code by a new one.

As well-known by people working on formalization of mathematics, even if there is no implementation, the specification of fragments of mathematics requires complex representation choices: how to express dependencies above the underlying sets? are functions first-class citizen? which equality on functions? etc. Dealing with true coding of algorithms adds several specific problems. For example, one wants to have a general notion of polynomials, allowing sharing of properties and of some algorithms but one wants also to firmly distinguish between two different implementations (sparse and dense for example) of polynomials because correctness proofs rely heavily on the data representation.

Our first requirement is the following. The library of algebraic structures has to provide not only the implementation of the classical tools to manipulate algebraic structures, but also their semantics, given by explicit verified statements. To code a given algorithm, the user of Foc may freely use 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. This was reflected at the very beginning of the Foc project by the choices we were led to. We could either ease the design of the proof part or of the programming part. For example, we could have chosen to code all the algorithms as Coq functions, code being extracted from the proofs. Then, mathematical description of algebraic structures would have been (rather) simple and encoding of algorithms (rather) close to their specification. But efficiency in Computer Algebra remains a bottleneck. Thus, we chose to focus on the programming part, trying to express as much as possible mathematical properties inside the pro-
gramming language. This led us to some requirements upon this programming language and upon the ways of writing code. At the end, we have chosen the programming language Ocaml as it fits well with our requirements and it is also the development language of the proof language Coq. Some other languages like CASL, Haskell, etc. may fit our requirements as well and it will be interesting to try an implementation with such programming or specification languages in the future, following the same trails.

In this paper, we present our choices for the specification and the implementation of the FOC library, focusing on the programming part but pointing out all the informations which have to be reflected on the proving side. Section 2 presents the encoding of mathematical structures. The example of polynomials is almost completely given in section 3, starting from an abstract view of univariate polynomials, giving then a sparse representation and ending with the recursive representation of multivariate polynomials.

2 Specifying mathematical structures

In this section we discuss the encoding of a mathematical specification of algebraic structures in a programming language. We first recall some basic algebraic definitions needed to describe our running example of polynomials (we apologize to our mathematician readers). Then, we set up our requirements for the encoding and we end with some illustrating code, written in Ocaml. The code is commented so we hope that it remains understandable for the reader, even if not acquainted with this language.

2.1 Algebraic background

A ring[13] \( A \) is a set with a binary addition and a binary multiplication. Usually addition will be denoted +, and, if ambiguous, by writing \(+_A\) when necessary. Multiplication will be denoted by \( * \) or by \( *_A \) if ambiguity. We thus provide the set \( A \) with two binary operations \( +_A : A \times A \to A \) and \( *_A : A \times A \to A \). These operations have some properties:

- \((A, +)\) is an additive abelian group that is
  - \(+\) is associative: \( \forall x, y, z \in A, x + (y + z) = (x + y) + z \). To express this property, an equality is needed. Thus the set \( A \) should have an equality \( = \) (or \( =_A \)).
  - \(+\) has a neutral element \( 0 \) (often disambiguate by \( 0_A \)) in \( A : \forall x \in A, x + 0 = 0 + x = x \). These two properties give to \((A, +)\) the structure of a monoid.
  - Every element of \( A \) has an opposite in \( A : \forall x \in A, \exists y \in A, x + y = y + x = 0 \). With this property, \((A, +)\) is a group. Thus a group has all operations and properties of a monoid. This is a form of inheritance.
  - \(+\) is commutative: \( \forall x, y \in A, x + y = y + x \) this property states that the group is abelian\(^2\).

\(^2\) we reserve the word commutative for a multiplicative monoid
(A, *) is a multiplicative monoid whose neutral element\(^3\) is denoted by 1 or \(1_A\).

multiplication is left and right distributive with respect to addition, that is \(\forall x, y, z \in A, x \ast (y + z) = x \ast y + x \ast z\) for left distributivity and \(\forall x, y, z \in A, (x + y) \ast z = x \ast z + y \ast z\) for right distributivity.

the ring is said to be commutative if its multiplication is commutative.

Now, let \((M, +_M)\) be an additive abelian group and let \((A, +_A, \ast_A)\) be a ring, we say that \(M\) is a left \(A\)-module if we have an external multiplication \(\ast_M\) between elements of \(A\) and elements of \(M\). This multiplication should be compatible with operations on \(A\), meaning that

- \(\forall a, b \in A, \forall m \in M, (a +_A b) \ast_M m = (a \ast_M m) +_M (b \ast_M m)\)
- \(\forall a, b \in A, \forall m \in M, (a \ast_A b) \ast_M m = a \ast_M (b \ast_M m)\)
- \(\forall a \in A, \forall m, n \in M, a \ast_M (m +_M n) = (a \ast_M m) +_M (a \ast_M n)\)
- \(\forall m \in M, 0_A \ast_M m = 0_M\)
- \(\forall m \in M, 1_A \ast_M m = m\)

Similar operations and properties hold for a right \(A\)-module with external multiplication denoted by \(\ast_M\). An \(A\)-module is a left and right \(A\)-module. We thus see that under the common name + we have two distinct operations +\(_A\) and +\(_M\) and that under the common name \(\ast\) we have three distinct operations \(\ast_A\), \(\ast_M\), and \(\ast\). Some choices have to be done for the management of this overloading in mathematical notation.

We say that \(E\) is an \(A\)-algebra if it is an \(A\)-module which has a \(A\)-bilinear mapping denoted by \(\ast_E\). If \(A\) and \(B\) are two rings and if \(f\) is a ring morphism from \(A\) to \(B\), \(B\) may be viewed as an \(A\)-algebra if the sub-ring \(f(A)\) of \(B\) commutes with \(B\). We define the \(A\)-module external multiplications by \(a \ast_B b = f(a) \ast_B b\) and \(b \ast_B a = b \ast_B f(a)\). As usual in commutative algebra we restrict our view to this particular type of algebras. We will often consider rings to be commutative by default and we require the ring morphism \(f\) to be injective allowing to view \(A\) as a sub-ring of \(B\).

### 2.2 Requirements for encoding mathematics

A CAS manipulates entities such as integers, polynomials, which are elements of some set. Thus we need a representation of these entities. What kind of representation? In mathematics sets are often considered as containers for elements. A set may have algebraic properties, such as being a ring. Denotations of the elements are needed to formulate these properties. But, no concrete information on these elements is required, apart from the existence of the membership relation and an equality.

To decrease distance between mathematics and code, tools to encode abstract views of representations are required. There are several ways to do that.

\(^3\) Some authors do not require the existence of a neutral element.
A first one is given by a (naive) object paradigm, described in most of textbooks on classes. Then object-oriented features like inheritance allow to say for instance that a ring is built upon an additive abelian group.

Entities are considered as objects of some class \( C_A \) encoding the set \( A \). A distinguished entity, as the unit of a group, can hardly be encoded as an object because objects are created only at run-time. So it is considered as a nullary operation, which is mathematically correct, but may demand the establishment of a conversion morphism when proving algorithms.

Operations are encoded by methods. For instance to add two elements \( a \) and \( b \) of a group \( G \), one would write \( a +_G b \) in mathematics whereas one should send the method \( +_G \) to the object \( a \) of the class \( G \) with the argument \( b \). Then arity is lost, we encounter the binary method problem, well-identified in programming languages. This is hardly acceptable on the proof side: code has to remain close to specifications and the arity of functions must be kept, at least for practical reasons.

We can adopt a more declarative view of mathematical structures. They are a bunch of operations acting over some sets and having some properties. The representation of entities is simply one part of the definitions. This is typically the view proposed by languages of the abstract data types framework, where types can be parameters or more defined expressions. Following this view, we may encode a set by a type, here after called the carrier of the set. Operations on a set can be encoded as functions on the carrier. For instance if a set \( A \) has a binary additive law this can be encoded by a function named \(+\) having type \( \tau_A \to \tau_A \to \tau_A \) if the carrier for \( A \) is \( \tau_A \). A constant of \( A \) can be some named data of type \( \tau_A \), for instance a neutral element for a binary additive law can be a constant \( 0 \) of type \( \tau_A \).

The carrier and the operations defining a given structure have to be gathered into a programming structure like a package, a module, a class, etc. Then, some powerful (multiple) inheritance mechanisms are required on these programming structures, to ease the programming task. The semantics of multiple inheritance should be clearly stated and if possible, formally studied. Indeed, on the proof side, a lot of lemmas rest upon mathematical inheritance between algebraic structures. This mathematical inheritance is reflected by programming inheritance. Even if we do not want to prove the internal mechanisms of inheritance of the host language, we must be confident of their correctness.

The way of packaging being chosen, we have to deal with overloading. Often, names are qualified by the name of the package, offering a first step of resolution of overloading. For example, the distinction between the operations \(+_A \) and \(+_M \) is achieved by qualification of names, which are \( A\#\text{plus} \) et \( M\#\text{plus} \). Thus, overloading has to be considered only inside a given algebraic structure, where we forbid it. This is the case for external \((*_{R_L} \) and \( *_{R_H} \) ) and internal \((*_{R}) \) multiplications of the \( A\)-algebra \( B \) which serve together to define \( B \). Thus we do not follow the mathematical habit and we require different names. Indeed, it is rarely the case that such an overloading is needed. For the proof side, this
is a sound choice as the properties of these operations differ and as explicit conversion must be done, with the help of the ring morphism $f$.

2.3 Coding mathematical specifications

We pursue our example, with a language of classes, which correspond to our implementation. How to assert that a particular set $A$ is a group with carrier $\tau_A$? As an example, we describe a stand-alone class (or a package or a module) `additive_group`. It is parameterized by the type $\tau_A$, which exports some methods. We use an Ocaml like syntax. Calling a method $m$ of an object $\text{obj}$ is written $\text{obj}.m$. The carrier of the group $A$ appears only as a type parameter of the class.

```ocaml
class virtual [\tau_A] additive_group =
  object (A)
    method = : \tau_A \to \tau_A \to \text{bool}
    method + : \tau_A \to \tau_A \to \tau_A
    method 0 : \tau_A
    method opposite : \tau_A \to \tau_A
    method $a - b = a + (\text{opposite } b)$
    ...
  end
```

The operation $+_A$ is encoded by the operation $A#\text{+}$. The $A$ appearing in $\text{object}(A)$ denotes the current object, like the self of several object languages. Operations are described by a curried type. This is only a matter of taste.

To describe rings, we do not explicitly rewrite all the methods but we use inheritance\(^4\). Note that the type parameter denoting the carrier of the ring serves as an actual parameter for the inheritance declarations. This is a sound encoding of the mathematical specification. Indeed, as they are built upon the same set, the additive group and the multiplicative monoid share equality of this set. This sharing might be extended to the monoid structure, if addition and multiplication could be seen as two instances of a same parameter. This is possible in a proof language like Coq but this is difficult to handle in programming languages, due either to decidability of typing or to the impossibility of changing method names. Thus, we consider separately addition and multiplication, defining two separate notions of monoid.

```ocaml
class virtual [\tau_A] ring =
  object (A)
    inherit [\tau_A] additive_group
    inherit [\tau_A] multiplicative_monoid
    ...
  end
```

$A$-module is parameterized by the ring $A$. To express such parameterized structures, we use value parameters and constraints on them. For example, the

\(^4\) Indeed, `additive_group` inherits from `additive_monoid`. 
constraint on \( A \) expresses that \( A \) should be a ring. This feature, which is offered only by a few programming languages, has to be semantically well-understood. It permits to express some parts of the mathematical specification, directly in the programming language. In the following code, the parameter \( A \) represents an Ocaml object denoting the underlying ring, \( \tau_A \) represents its carrier. As an Ocaml object, \( A \) has a type denoted by \( \sigma_A \), which is constrained to be compatible with the class \([\tau_A]\) ring. This property is checked by the Ocaml type-checker. \( \tau_M \) represents the carrier of the module itself.

```ocaml
class virtual \([\sigma_A, \tau_A, \tau_M]\) left_module \((A : \sigma_A) =
  object(M)
    constraint \( \sigma_A = (\tau_A)#\text{ring} \)
    inherit \([\tau_M]\) additive_group
    method \(*_l : \tau_A \to \tau_M \to \tau_M\)
    ...
  end
```

We can now proceed with the definition of algebras over commutative rings by specifying the ring morphism \( f \), here denoted by `coerce`. As before, the parameter \( B \) (in `object(B)`) denotes the structure being specified, its carrier being denoted by the type parameter \( \tau_B \). The ring \( A \) is denoted by the value parameter \( A \), its carrier by \( \tau_A \) and its properties by \( \sigma_A \). These properties are given by the constraint `\( \sigma_A = (\tau_A)#\text{commutative}\text{ring} \)'.

```ocaml
class virtual \([\sigma_A, \tau_A, \tau_B]\) algebra \((A : \sigma_A) =
  object(B)
    constraint \( \sigma_A = (\tau_A)#\text{commutative}\text{ring} \)
    inherit \([\sigma_A, \tau_A, \tau_B]\) module \((A)\)
    method \text{coerce} : \tau_A \to \tau_B
    method \( a * b = (B#\text{coerce} a) *_B b \)
    ...
  end
```

Note that the external multiplication is defined exactly as it is in the mathematical specification of an \( A \)-algebra.

We end this section by a few words on equality. Some algorithms need to check equality between two elements or check elements to 0. Thus, equality, or at least the considered implementation, should be effective, that is decidable. In the current state of our project, we focus on polynomial arithmetic which heavily use zero checks and we have made this rather strong hypothesis on equality.

### 2.4 Species of Foc

The previous presentation uses several views of mathematical structures, which are considered sometimes as algebraic structures as in any algebra book, sometimes as mathematical specifications, sometimes as (rather abstract) implementations, sometimes as arguments for proofs, etc. We call species our own notion of algebraic structures, seen as mathematical data submitted to effective coding and proving. For example, there is a species (called informally here the rooting
species of monoids) which corresponds only to the mathematical definition of a monoid. Then, this species serves to build a new species, which defines natural numbers as a monoid, leaving free the precise choice of the implementation of the carrier. This last species is used to build the species of, say, big natural numbers, whose representation is given by the library GMP and whose operations are described by functions submitted to invariant properties. Thus, species have a mathematical counterpart, but also a programming and a proving ones. We have to embrace all these aspects at a time, any decision about one of these parts having possibly heavy consequences on the other ones.

The encoding of species into Ocaml is presented in this paper. The specification of species has been heavily studied in S. Boullé’s thesis[2,3], by coding it in Coq and giving a categorical model à la Cartmell[19]. The introduction of a new species has been decomposed into atomic steps, corresponding to atomic stages of proof correctness. A concrete syntax for species is currently under design[15]. This syntax is compiled into Ocaml code, which is very close to the code already written by hand. This syntax will serve also to build statements and proofs, which are to be verified by the Coq verifier (at the moment, proofs are done directly in Coq).

A species is defined by an ordered bunch of components, described rather informally in a polymorphic typed framework. The first component is the carrier of the species, the simplest one being a type variable τ. For example, the parameter τ_A represents the carrier of additive_group. As we shall see below, τ may progressively be instantiated by a type expression still containing other type variables or by an explicit data type. Thus, a first way of creating a new species is a carrier instantiation.

The primitive components 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). This is the case of the + method and of the opposite method of additive_group.

A given species 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 τ). The method (\_ - \_ ) of additive_group is defined using + and opposite.

A second way to create new species is to extend a given species by adding primitive or derived components. For instance ring is an extension of the species additive_group. 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 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. This will be illustrated below.

A component, say c, of a given species S_1 may be redefined, leading to a new species 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. For example, the method $n_1$ of algebra redefines that of left module.

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 gives a complete implementation of a mathematical set, ready for users. It is created by applying an encapsulation mechanism to the given species, to forbid direct use of data representation so for modularity purposes. Here, we use this notion only for the following definition.

Species can receive parameters as long as those are collections or entities. Thus, a parameterized species is a kind of “function” taking collections or entities and returning a species. For instance, algebra is parameterized by the ring $A$. As we will see below, previous operations on species also apply to parameterized species.

A species $\mathcal{S}_1$ can be converted into a species $\mathcal{S}_2$ by establishing 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. This conversion requires a rather complex handling of dependencies, which is not described here.

3 Polynomials

In this section we will carry out the example of the implementation of polynomials. We will first describe univariate polynomials and then proceed toward multivariate polynomials.

3.1 The species of univariate polynomials

Depending of authors, there are several mathematical definitions of polynomials, as the solution of a universal problem, as an almost null sequence of coefficients, etc. We choose to have two different primitive notions of polynomials, according to their arrangement of variables, which correspond to two different species rooting the different implementations of polynomials.

Let $A$ be a ring, $A[X]$ is the ring of univariate polynomials with coefficients on $A$. It is a commutative $A$-algebra, together with some basic primitives such as the degree, the leading coefficient. Usually the degree of a polynomial $P$ of $A[X]$ is a non-negative integer. In fact, the set $D$ of degrees is simply required to be a regular additive monoid (a monoid with simplification: $\forall x, y, z \in D, x + z = y + z \implies x = y$). This monoid should have a total well-founded ordering $\leq$ compatible with addition: $\forall x, y, z \in D, x \leq y \implies x+z \leq y+z$ with 0 as minimal element. This guarantees correct use of the common rules that $X^d \cdot X^{d'} = X^{d+d'}$ and $X^0 = 1_{A[X]}$. 

9
We give first the coding of the species describing monomial orderings, still using Ocaml syntax.

class virtual [\tau_D] monomial_ordering =
  object(D)
    inherit [\tau_D] regular_additive_monoid
    inherit [\tau_D] total_ordering
  end

Notice that the compatibility of \(<\) upon \(+\) is not explicit in the previous code nor the fact that \(0\) is minimal for \(<\). Indeed, such invariant properties cannot be directly reflected in the running code. But, in our user language (currently under design), such properties need to be stated and proved.

Then, the coding of the (rooting) species of univariate polynomials is written:

class [\sigma_A, \tau_A, \sigma_D, \tau_A, \tau_F]  
abstract_univariate_polynomials (A : \sigma_A , D : \sigma_D) =
  object (A[X])
    constraint \sigma_A = (\tau_A)\#commutative_ring
    constraint \sigma_D = (\tau_D)\#monomial_ordering
    inherit [(\tau_F)] commutative_ring
    inherit [\sigma_A, \tau_A, \tau_F] algebra (A)
    method |·| : \tau_F -> \tau_D
  ...
end

3.2 Implementing polynomials: the sparse representation

Usually in computer algebra univariate polynomials are encoded using either a dense or a sparse representation.

Dense polynomials are encoded using vectors of elements of \(A\), indexed by elements of \(N\). We will not give a formalization of these dense polynomials since to obtain an efficient implementation one is lead to have contiguous memory cells for the elements, destructive operations over the vectors and very often require user explicit pointer manipulation. We do not want to tackle now the formalization of such operations. However, this can be done using monads[21], for example and some of these features have been encoded in Coq by Filliatre[10].

We focus on sparse polynomials where a polynomial is encoded using a list of “monomials” which are pairs of non null elements of \(A\) and elements of \(N\). One can give an inductive definition of the sparse representation as follows:

\[
\begin{align*}
P_0 &= 0p \cup \{(a,0_N) \mid a \neq 0_A\} \\
P_d &= \{d \leq \sigma < d \in P_d\} \cup \{(d,a), p) \mid d \in N, a \in A, p \in P_d, \sigma < d, a \neq 0_A\}
\end{align*}
\]

Here \(P_d\) is the set of sparse polynomials of degree at most \(d\). \(A[X]\), the set of sparse polynomials is the inductive limit of all possible \(P_d\). This limit exists because the ordering on \(N\) is well founded. Note that monomials are ordered by decreasing order of degree. If this can be reflected in the implementation, this enables to access both the degree and the leading coefficient of a polynomial in constant time.

10
We now need to implement this abstract recursive data structure, which can be thought as a list of pairs. There are several ways of implementing lists: arrays, pointers, classes, or recursive types. As already said, we reject arrays and pointers. We reject also any too abstract view of lists, either given by abstract data types (or modules) or given by classes, which offer only access to the head or the queue of the list. Indeed, the representation of the entities is intended to model the effective data and operations on this data must use as far as possible the internal definition of the data. Thus, we choose to encode the representation of sparse polynomials by a recursive concrete type:

\[
\text{type } (\tau_A, \tau_N)\tau_A[X] = \\
| Z \\
| N \text{ of } (\tau_N \star \tau_A) \star (\tau_A, \tau_N)\tau_A[X]
\]

Then, implementations of functions, like \(| P |\), can be defined by pattern-matching, which is usually an efficient way to access parts (like \(d\) in \(| P |\)) of data. Moreover, pattern-matching eases correctness proofs. Without it, \(d\) would be obtained by a more complicated code looking like “if list is not empty then (first (head list))”.

\[
\text{class } [\tau_A, \sigma_A, \tau_D, \sigma_D] \\
\text{sparse\_univariate\_polynomials } (A:\sigma_A, D:\sigma_D) = \\
\text{object } (\text{A}[X]) \\
\text{constraint } \sigma_A = (\tau_A)\text{#commutative\_ring} \\
\text{constraint } \sigma_D = (\tau_D)\text{#monomial\_ordering} \\
\text{inherit } [\sigma_A, \tau_A, \sigma_D, \tau_D, (\tau_A, \tau_D)\tau_A[X]] \\
\text{abstract\_univariate\_polynomials } (A, D) \\
\text{method } |P| = \text{match } P \text{ with} \\
| Z \to 0_D \\
| N \to ((d, c), r) \to d \\
\]

Our implementation of the degree function assumes that the first monomial of a non null polynomial is the one with highest degree. This cannot be reflected directly in the Ocaml type and has to be maintained as an invariant of the implementation.

Here, we have defined a new species, \text{sparse\_univariate\_polynomials}, from the one called above \text{abstract\_univariate\_polynomials}. We use the inheritance mechanism. We have made a carrier instantiation of \(\tau_P\) with the more defined expression \((\tau_A, \tau_D)\tau_A[X]\), still containing type variables. We have implemented the function \(| P |\), doing a step of refinement. Thus, we have built this new species by a composition of primitive operations on species.

This new species is parameterized by the ring of coefficients and the set of degrees, which are given as actual parameters to \text{abstract\_univariate\_polynomials}. Therefore, the constraints on them are automatically checked by Ocaml and they could remain implicit. We choose to make them explicit as they are parts of the mathematical specification of the sparse polynomials.
3.3 Multivariate Polynomials

We do not describe here the (rooting) species of multivariate polynomials. We explain only our choice for the representation of these polynomials.

To represent polynomials in several variables, say $X$ and $Y$, we can use either a distributed representation, either a recursive representation.

Let $N_1$ be the set of degrees in $X$ and $N_2$ be the set of degrees in $Y$. The distributed representation is parameterized by a well founded ordering on $N_1 \times N_2$. This ordering can be the lexicographical ordering, the total degree ordering, etc. We use the latter sparse representation with the set $D = N_1 \times N_2$ as the set of degrees of the polynomials. We thus see that distributed representations are a generalization of sparse univariate encoding.

For the recursive representation of polynomials in $X$ and $Y$ we use a precedence between $X$ and $Y$, say $Y$ greater than $X$. Thus, we view polynomials in $X$ and $Y$ as univariate polynomials in $Y$ whose coefficients are polynomials in $X$.

We distinguish the case of constant polynomials, which are considered as constant polynomials in any variable. The recursive representation of polynomials in $X$ and $Y$ is defined as follows.

$$
\begin{align*}
P_0 &= A \\
P^X_0 &= P_0 \cup \{(1, p) | p \in P_0, p \neq 0_A\} \\
P^X_i &= P^X_{i-1} \cup \{(i + 1, p), q) | q \in P^X_{i-1}, p \neq 0_A\} \\
P^X &= \bigcup_i P^X_i \\
P^X_Y &= P^X \cup \{(i, p) | p \in P^X, \deg\text{r} > 0_N\} \\
P^X_j &= P^X_{j-1} \cup \{(j + 1, p), q) | q \in P^X_{j-1}, p \in P^X \neq 0_A\} \\
P^X_{j, Y} &= \bigcup_j P^X_{j, Y}
\end{align*}
$$

The set $P^X_{j, Y}$ is the set of all polynomials in $X$ and $Y$. It is well-defined as the inductive limit of the $P^X_{i, Y}$. Note that precedence is used to construct the fix-point step by step.

To obtain the set $P$ of all polynomials, we need to choose a precedence, which is a total well-founded ordering on variables. Let $(X_k)$ be the ordered set of variables. Then, $P = \bigcup_k P^{X_1, \ldots, X_k}$ and the definition of $P^{X_1, \ldots, X_k}$ is the straightforward generalization of the one of $P^{X, Y}$. Here is the corresponding type for the carrier, in a Ocaml like syntax :

```
type \((\tau_A, \tau_N, \tau_V)\tau_A[X_1, \ldots, X_n] =
  \mid \text{G of } \tau_A
  \mid \text{P of } \tau_V\ast(\tau_A, \tau_N, \tau_V)\tau_A[X_1, \ldots, X_n], \tau_N)\tau_A[X]
```

Here $(\tau_A[X_1, \ldots, X_n], \tau_N)\tau_A[X]$ is the type for univariate polynomials with coefficients over $\tau_A[X_1, \ldots, X_n]$ using $\tau_N$ to represent the degrees and $\tau_V$ for indexing the variables. For example, the polynomial $(5X^7 + 3X)Y^8 + 2$ is coded as follows:

```
P(Y, \mu((8, P(X, \mu((7, G 5), \mu((1, G 3), Z)))), \mu((0, G 2), Z)))
```

Again, invariants given by the orderings are lost in this definition. We cannot deduce from the type, that if a polynomial is non constant, it is a univariate polynomial in its main variable.
To end this example, we will define $P$ as a commutative $A$-algebra with carrier $(\tau_A, \tau_N, \tau_V)\tau_A[x_1 \ldots x_n]$. This last type is abbreviated into $\tau_P$, using the keyword as.

To implement the operations of $A[x_1 \ldots X_n]$ we need to view it as the set of univariate polynomials $A[x_1 \ldots X_{n-1}][X_n]$. Note that the representation of $A[x_1 \ldots X_{n-1}]$ has its type also described by $P_r$. Therefore, we are led to call operations of $P_r[X]$. But we have already an implementation of sparse univariate polynomials, given by the \texttt{sparse_univariate_polynomials} class. Therefore, we reuse it.

The following code is very close to the running one. It expresses a lot of mathematical but also implementation dependencies. We have omitted some details to ease the comprehension. But, as the reader can see, this code is not so simple.

```ocaml
class [\sigma_A, \tau_A, \sigma_D, \tau_D, \sigma_V, \tau_V]
  recursive sparse_multivariate_polynomials
  (A : \sigma_A, D : \sigma_D, V : \sigma_D)
object(P_r : \sigma_P)
  constraint \sigma_A = (\tau_A)\texttt{#commutative_ring}
  constraint \sigma_D = (\tau_D)\texttt{#monomial_ordering}
  constraint \sigma_V = (\tau_V)\texttt{#ordered_set}
inherit [(\tau_A, \tau_N, \tau_V)\tau_A[x_1 \ldots x_n] as \tau_P] \texttt{commutative_ring}
inherit [\sigma_A, \tau_A, \tau_P] \texttt{algebra(A)}
method P_r[X] = \texttt{new sparse_univariate_polynomials}(P_r, D)
...
end
```

The method $P_r[X]$ enables to call operations from the ring $P_r[X]$ which are implemented in the class \texttt{sparse_univariate_polynomials}. This method is correctly typed. Indeed, $P_r$ is an Ocaml object with type $\sigma_P$, this type is compatible with \texttt{commutative_ring} since the class of $P_r$ inherits from it.

\section{Conclusion}

In this paper, we have tried to give some insights on the building of the Foc library, following the implementation of polynomials as a running example. No algorithm on polynomials is presented here, due to the lack of place. But we can say that the sub-resultant algorithm was encoded using the recursive representation, without any difficulty. Running it on classical benchmarks shows that it meets our requirements on time and memory efficiency. These benchmarks are available at \url{www-calfor.1lp6.fr/~foc}.

Designing the Foc library, we were always concerned by its emerging proof counterpart. This led us to introduce the notion of species, which gathers together different views of effective mathematics. The specification of the species and the properties of operations manipulating them are formally studied in \cite{2}. This paper may be viewed as a presentation of the implementation of species, the motivations of our choices and their justification.
The FoC library implements general computer algebra notions without the help of a dedicated computer algebra language. Using a general purpose language has some advantages, among them to have libraries for data structures (lists, trees, graphs, etc.), for input-output in different formats, for interoperability with specialized libraries such as GMP. The major advantage is that it avoids writing a full compiler, a task which needs the skills of specialists. Moreover, a general purpose language is tested every day by its users. This gives a guaranty on its maintainability (correction of bugs and extensions).

However, using such a general language has several drawbacks. The first one is that we have to be confident on its semantic foundations. These ones have to be exposed and formally studied as long as possible. This is the case for Ocaml, which is based upon research[14,16,17] on types, modules, classes, etc. Thus, we can claim that this drawback is only a minor one. Moreover, any semantic flaw is eventually emerging, due to intensive uses of this language.

A more serious drawback is that a general language offers a lot of possibilities to encode algorithms, which are not guaranteed to fit our requirements. The first species were implemented by a very limited team, with permanent code revue. But, we hope to have some helps to describe species for other mathematical domains and we need to elaborate a programming discipline. It makes a restricted use of object oriented features but uses the full power of the class sub-language. We tried to give a flavor on it with the examples (choice for the representation of the carrier, binary operations keeping their two arguments, constraints expressing mathematical facts, redefinition and late binding, etc.). This discipline is natural enough to be followed by some undergraduate students adding species (fractions, matrix) without true difficulties. But we think that we cannot rely only on willingness. Therefore, a concrete syntax has been defined for species. It disguises Ocaml syntax for declarations of types, classes and offers only restrictive constructions to write functions (no references for example), allowing however all the ones used in code written by hand like late binding. Therefore, this is not surprising that the parser producing Ocaml sources generates code very close to the code written by hand. The concrete syntax serves also to build statements, submitted to proofs. Thus, the implementation of species can be certified at the source level. We are aware that this does not give full guarantees but this is not realistic to attempt certifying executable object-code.

References

5. S. Boumê, T. Hardin, R. Rioboo, Polymorphic data types, objects, modules and functions: is it too much?, LIP6 report 2000-014.
13. S. Lang Algebra Addison-Wesley Publishing Co., 1965