Formalization of Mathematics : proof of the Cayley-Hamilton Theorem

Sidi Ould Biha [1]

1: Inria Sophia-Antipolis,
2004, route des Lucioles - B.P. 93 06902 Sophia Antipolis Cedex, France
Sidi.Ould_biha@sophia.inria.fr

Abstract

The Cayley-Hamilton theorem is a major theorem of linear algebra. In this article, we present a first formalization of this theorem in a proof assistant. This formalization was developed in Coq using its extension SSReflect developed by G. Gonthier. This work is based on developments on matrices, polynomials and indexed operations. It is a part of the work of formalization of the Feit-Thompson theorem on the groups of odd order.

1. Introduction

Formal proofs Systems can be very useful in the verification and validation of mathematical proofs, especially when the proofs are complex and lengthy. Recent work, as formal proof of the theorem of 4 colors [6] or the theorem of prime numbers [2], show that these systems have reached a level of maturity to deal with non-trivial mathematical problems. The work of formalization of mathematical theories involving a wide variety of mathematical objects requires the adoption of a similar approach to software engineering. The formalization of such theories can be seen as a development involving different components: definitions and mathematical proofs.

A list of the 100 greatest mathematical theorems [1] was formed by Paul and Jack Abad. This list takes into account the place of theorem in the mathematical literature, the quality of its proof and the importance of the result he introduced. F. Wiedijk maintains a list [13] which enumerates formalizations of those theorems in some formal proofs systems. The Cayley-Hamilton theorem is in this list. This paper presents a formalization of this theorem, which is to our knowledge the first. The fact that he had never been formalized can be explained by the fact that it involves a lot of mathematical objects and properties of different kind (linear algebra, multi-linear, combinatorics, etc...). These objects are not used independently; but on the contrary they fit with each other. This work of formalization of the theorem of Cayley-Hamilton is a part of the work of formalization of the theorem of Feit-Thompson on the groups of odd order. The objectives is not only to formalize Cayley-Hamilton; but to organize the proof in reusable libraries.

The article is organized as follows. In section 2, we present the statement and the proof of the Cayley-Hamilton theorem. In section 3, we briefly present SSReflect, the extension of Coq and platform of our development. Finally, in section 4, we present the development which was necessary to arrive at this formalization of the Cayley-Hamilton theorem.

2. The Cayley-Hamilton theorem

The Cayley-Hamilton theorem can be stated [4] in the following way:

Any square matrix on a commutative ring satisfies its own characteristic equation.

More formally, if R is a commutative ring and A a square matrix on R, then the characteristic polynomial of A, defined by: $p_A(x) = \det(xI_n - A)$, vanishes in A.
The theorem can be stated differently by considering the endomorphisms of vector space. In this case it is not any more question of commutative ring but of field.

The Cayley-Hamilton theorem is used to make computations on square matrices or endomorphisms : computation of the inverse matrix and the eigenvalues. A corollary of this theorem is the result according to which, the minimal polynomial of a given matrix is a divisor of its characteristic polynomial.

The proof of the Cayley-Hamilton theorem presented in [4] use the Cramer formula. By noting $Adj(B)$ the adjugate matrix of B (the transpose of the cofactor matrix of B), the Cramer rule states :

$$B * Adj(B) = Adj(B) * B = \det(B) * I_n \tag{1}$$

By applying the formula (1) to the matrix $(xI_n - A) \in M_n(R[x])$, we obtain:

$$Adj(xI_n - A) * (xI_n - A) = \det(xI_n - A) * I_n = p_A(x) * I_n \tag{2}$$

The ring $M_n(R[x])$ of the matrices of polynomials is also the ring of the polynomials with matrix coefficients $(M_n(R))[x]$. In $(M_n(R))[x]$, the equality (2) is written :

$$Adj(xI_n - A) * (x - A) = p_A(x) \tag{3}$$

This shows that $(x - A)$ is factor of $p_A(x)$ in $(M_n(R))[x]$, so $p_A(A) = O_n$.

To formalize a mathematical proof in a proof assistant, we need to develop this proof to be comprehensible to a computer. To reach this objective, two difficulties are to overcome. In first place, it is necessary to make explicit the parts of the proof which are implicit or " trivial' " for a mathematician. Paradoxically, the implementation on computer of these parts, which do not appear in the proof, is the most complex work in the formalization. In the second place, it is necessary to have statements comprehensible for a mathematician and human reader. The interest is not simply to make proof on computers but it's necessary that the statements of this proofs are nearest possible to those used in the mathematics literature. This will facilitate the re-use of this proofs in other developments.

In the case of the Cayley-Hamilton theorem and by considering the proof above, several problems arise at the time of its formalization. To say that $M_n(R[x])$ is identical to $(M_n(R))[x]$ is algebraically equivalent to say that there is an isomorphism of ring between them. Indeed, any matrix of polynomials can be written, in a single way, as the sum of powers in x multiplied by matrices, i.e. a polynomial with matrix coefficients. For example :

$$\begin{pmatrix} x^2+1 & x-2 \\ -x+3 & 2x-4 \end{pmatrix} = x^2 \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + x \begin{pmatrix} 0 & 1 \\ -1 & 2 \end{pmatrix} + \begin{pmatrix} 1 & -2 \\ 3 & -4 \end{pmatrix} \tag{4}$$

The formalization of this isomorphism corresponds to the writing of the function of transformation described in the example above. The properties of this morphism, which we will note ϕ, are used

implicitly in the proof. Indeed, in (2) the members of the equality are matrices of polynomials. The application of ϕ to the left and right side of the equality (2) gives us:

$$\phi(Adj(xI_n - A) * (xI_n - A)) = \phi(p_A(x) * I_n)$$

The properties of morphism of ϕ are used to obtain:

$$\phi(Adj(xI_n - A)) * \phi(xI_n - A) = \phi(p_A(x) * I_n)$$

The formula (3) corresponds explicitly to the equality above.

3. SSReflect

SSReflect [7, 8] (for *Small Scale Reflection*) is an extension of Coq [3] which introduces new tactics and Coq libraries adapted to work on types with a decidable equality and equivalent to the structural equality of Coq, called the Leibniz equality. It was initially developed by G. Gonthier in his proof of the 4 colors theorem. A development [8] on the theory of finite groups was made with SSReflect. This development includes, among others, a formalization of the Sylow theorem and the Cauchy-Frobenuis lemma.
We will present firstly the method of SSReflect to reflect between Boolean and Propositional predicates of Coq logic. We introduce thereafter the language of tactic of SSReflect. Finally, we present some SSReflect Coq libraries that we used in our development. More detailed information on SSReflect and precisely its language of tactic can be obtained in [7].

Reflection

In the Coq proof system, the default logic is intuitionistic. In this logic, the logical proposition and the Boolean values are distinct. SSReflect makes it possible to combine the best of the two visions and to pass from the propositional version of a decidable predicate towards there Boolean version. The propositional version makes it possible to have structured proof whereas the Boolean makes it possible to make computation. To do this, the Boolean type is injected into the proposition type by a coercion:

```
Coercion is_true (b: bool) := b = true.
```

Therefore, and transparently to the user, when Coq waits an object of type `Prop` and receives a value `b` of type `bool`, it will automatically translate it into the proposition `(is_true b)`, which corresponds to the proposition `b = true`.
The inductive predicate `reflect` gives a practical and comfortable equivalence between the decidable propositions and the Booleans:

```
Inductive reflect (P: Prop): bool -> Type :=
  | Reflect_true: P => reflect P true
  | Reflect_false: ~P => reflect P false.
```

The proposition `(reflect P b)` indicates that `P` is equivalent to `(is_true b)`. For example, equivalence between the Boolean conjunction `&&` and the propositional one `/\` is given by the following lemma:

```
Lemma andP: forall a b:bool, reflect (a /\ b) (a && b).
```

Lemma of the same nature as `(andP)` are defined when we want to have the equivalence between computational representation of a function (which can be calculated) and its logical representation. More details on the use of `reflect` are available in the SSReflect documentation [7].

Tactics language

The proof scripts written with SSReflect differ from those written using standard Coq tactics. The tactic language of SSReflect facilitates the operations of interpretation and development of proof scripts. In practice, the proof scripts written with SSReflect are more concise than those written using standard Coq tactics.
All the frequent operations that consists in moving, splitting, generalizing formulas form (or to) the context are regrouped in the tactic `move`. For example, the tactics "`move: (H1 a)`" put, in the current goal, a instance of the hypothesis `H1` for the variable `a`. Another example is the tactics "`move=> x y H2`" which corresponds to the introduction of the variables `x` and `y`, and a new hypothesis `H2` in the current context. The tactic "`move: (H1 a) => H2 x y`" corresponds to the combination of the two examples above in a single tactic.
The tactic `rewrite` combines in single command all the operations of conditional rewriting, unfolding of definition, simplifying and rewriting selecting occurrences or patterns. Those operations can be used together or separately. For example, the tactic `rewrite /def H1 ?H2 !H3 {2}[_ * _]H4 //` unfolds the definition `def`, rewrites the hypothesis `H1`, rewrites zero or several times with the hypothesis `H2`, rewrites at least once time with the hypothesis `H3`, rewrites in the second occurrence of the pattern `[_ * _]` with the hypothesis `H4` and simplifies the current goal.

The mechanism of reflection between the decidable proposition and the boolean, described above, is integrated in the SSReflect tactic language. For example, given a context with a hypothesis `H` of type `a && b`, the tactic `move/andP : H => H` apply the lemma `andP` to `H` and introduces in the current context a new hypothesis `H` of type `a /\ b`. On the other hand, when the goal is like `a && b`, the tactics `apply/andP` replaces it by `a /\ b`. Finally, when the goal is `(a && b) -> G`, the tactics `case/andP => H1 H2` changes it to `G` and introduces two new hypothesis `H1 : a` and `H2 : b` in the context.

Libraries

Some Basics libraries are defined in SSReflect. They consist in a hierarchy of structures to work with decidable theories and especially finite types. The structure `eqType` defines the types equipped with a decidable equality and equivalent to the Leibniz one.

```
Structure eqType : Type := EqType {
  sort :> Type;
  _ == _ :  sort -> sort -> bool;
  eqP :  forall x y, reflect (x = y) (x == y)
}.
```

The `:>` symbol declares `sort` as a coercion from an `eqType` to its carrier type. It is the standard technique to get subtyping. The `eqType` structure not only assume the existence of a decidable equality `==` but also `eqP` injects this equality into the Leibniz one. Then, We can benefit from the power of rewriting of Coq.
A major property of `eqType` structures is that they give the property of *proof-irrelevance* for the equality proofs of their elements. Therefore we have a one proof of equality for each pair of equal objects.

```
Lemma eq_irrelevance: forall (d: eqType) (x y: d) (E: x = y) (E': x = y), E = E'.
```

A set on a structure of `eqType` is represented by its characteristic function:

```
Definition set (d: eqType) := d -> bool.
```

A Boolean property over an `eqType` corresponds to the set of all elements which satisfy it. With this definition the set operations like the intersection or the complementary are defined by the corresponding boolean functions.

```
Definition setI (a b : set d) : set d := fun x => a x && b x.
Definition setC (a : set d) : set d := fun x => ~~ a x.
```

It is useful to have a type of lists on a `eqType d` to be able to define more naturally the operations which is based on a boolean test like the search of an element in a list. The type of lists on an `eqType d` is defined in an inductive way by:

```
Inductive seq : Type := Seq0 | Adds (x : d) (s : seq).
```

`Adds` and `Seq0` correspond respectively to the constructor `cons` and `nil` of the Coq standard type `list`. The type `seq d` defines a list over an `eqType d`. On this new type of list, functions are defined to handle and reason on its elements. The function `foldr` corresponds in SSReflect to the operation *fold* used in functional programming. The extraction operation of an element of a list is given by the function `sub`. For example, `sub x0 s i` return the element of index `i` in the list `s`, if `i` is strictly lower than the length of the list, and `x0` otherwise. The function `mkseq` builds a list of given length from a function on the naturals. For example, `mkseq f n` corresponds to the list `[(f 0), (f 1), ..., (f n-1)]`.

A finite type can be seen as a finite set. It can then be represented by the list of all its elements. The definition of finite types is based on that of `seq` type. The structure `finType` consist of a list on a `eqType sort` and a proof that no element of this list appears more than one time.

```
Structure finType : Type := FinType {
  sort :> eqType;
  enum :  seq  sort;
  enumP :  forall x, count (set1 x) enum = 1
}.
```

In this definition `(set1 x)` is the set that contains only x and `(count f l)` computes the number of elements `y` of the list `l` for which `(f y)` is true. The parameter `enum` corresponds to the list of the elements of the finished type. In the case of a `finType d`, (`enum d`) return the list of elements of `d`.

To represent the finished types of the element of the interval $0..n-1$, the library SSReflect provides a family of types named `ordinal` whose elements are pairs of a natural number p and a proof that p is lower than n. As this proof is based on a boolean test, the property of irrelevance applies and the elements of this type are only characterized by the component p. The notation `I_(n)` represents the type `ordinal n`.

4. Formalization

In this work of formalization of the Cayley-Hamilton theorem, we use libraries on indexed operations and determinants. Those libraries were developed by Y. Bertot and G. Gonthier in the project "Mathematical Components". The library on the polynomials provides formalization of the algebraic properties of the polynomials, the evaluation morphism and the factor theorem. The definition of isomorphism between the ring of the matrices of polynomials and that of the polynomials of matrices is the ultimate step in the formalization of Cayley-Hamilton theorem.

4.1. Indexed operations

In the definition of the operations on the matrices, for example the multiplication or the determinant, the indexed operations (sum and product) is frequent. Factorize the proof of general properties on the indexed sums and products reduces considerably the length and the complexity of the proof, and gives a more readable statements. A library for indexed operations is not only useful in the development on the matrices theory, it could be useful in developments more general than linear algebra.

An indexed operation is the generalization of the definition of a binary operation to the elements of a finite sequence. In the particular case of the addition, it is the sum of all the elements of a given sequence.
The definition of an indexed operation is given by:

```
Definition reducebig R I op nil (r : seq I) P F : R :=
  foldr (fun i x => if P i then op (F i : R) x else x) nil r.
```

The parameters of the function `reducebig` are : a type (not necessary an `eqType`) `R`, an `eqType I`, a binary operation `op` on `R`, a element `nil` of `R` corresponding to the empty set, a list `r` of elements of `I`, a characteristic property `P` of `I` (a function form `I` to `bool`: a set of elements of `I`) and a function `F` form `I` to `R`. The result of `reducebig` corresponds schematically to:

$$F\ p_1\ op\ F\ p_2\ op\ \cdots\ op\ F\ p_n\ op\ nil,$$

The p_i are the elements of the list r for which the property P is true : the elements of the set P. The use of `reducebig` is more natural when the operation is associative and commutative and when `nil` is the neutral element of this operation. In other words, when `(R, op, nil)` is a monoid.
The notation `\big[*%M/1]_(i| P i) F i` corresponds to the application of `reducebig` to a monoid operation `*` which has a neutral element `1`. The others parameters of `reducebig` are inferred implicitly by Coq. For example, `\big[*%M/1]_(i| i < n) i` represents the mathematical standard notation $\sum_{i<n} i$.

A interesting lemma on `reducebig` is the one who gives the usual re-indexation property. In the case of an indexed sum, this lemma corresponds to the equality between the both sums $\sum_{i=0}^{n}(i+m)$ and $\sum_{j=m}^{n+m} j$. To formalize this property, we consider that i and j are respectively of types $[0..n]$ and $[m..n+m]$ (different types), and there is a bijection between these two types. This bijection is the function $f : x \to x+m$.
The predicate `ibjective P h` says that the function `h` is bijective on the set `P` of his domain. It is necessary for proving the re-indexation lemma.

```
Lemma reindex : forall (I J : finType) (h : J -> I) P F,
  ibijective P h ->
  \big[*%M/1]_(i | P i) F i = \big[*%M/1]_(j | P (h j)) F (h j).
```

Another interesting property of indexed operations is that of the decomposition following a partition of the index set. For example and in the case of the indexed sum, this property is written $\sum_{i=0}^{n+m} i = \sum_{i=0}^{n} i + \sum_{i=n+1}^{n+m} i$. The generalization of the property above is written in Coq :

```
Lemma bigID : forall (I : finType) (a : set I) (P : I -> bool) F,
  \big[*%M/1]_(i | P i) F i
    = \big[*%M/1]_(i | P i && a i) F i * \big[*%M/1]_(i | P i && ~~ a i) F i.
```

In this lemma, being given a set `a`, a partition of a set `P` is the two sets $\mathtt{P} \cap \mathtt{a}$ and $\mathtt{P} \cap \bar{\mathtt{a}}$. The sum of the elements indexed by `P` can be decomposed into two sums of the elements indexed by these two sets.

4.2. Canonicals Structures

In the proof assistant Coq, *a Canonical Structure is an instance of a record/structure type that can be used to solve equations involving implicit arguments* [12]. For example, to define a structure of `eqType` on the type `nat`, we have to define a decidable equality on the Coq naturals and proof that this equality is equivalent to the Leibniz one.

```
Fixpoint eqn (m n : nat) {struct m} : bool :=
  match m, n with
  | 0, 0 => true
  | S m', S n' => eqn m' n'
  | _, _ => false
  end.
Lemma eqnP : reflect_eq eqn.
Proof.
...
Qed.
Canonical Structure nat_eqType := EqType (@eqnP).
```

The following lemma shows a simple example of use of the `Canonical Structure`.

```
Lemma eqn_add0 : forall m n:nat, (m + n == 0) = (m == 0) && (n == 0).
```

We mention above that `==` represents the equality in a `eqType`. In the statement above Coq expect that `m` and `n` are of type a `eqType`. But they are of type `nat`. Coq try to find a definition of a `eqType` whose parameter `sort` is `nat`. Thanks to the definition of `Canonical Structure nat_eqType`, Coq can infer automatically the type `nat_eqType` to the arguments `m` and `n`.

The mechanism of `Canonical Structure` is powerful and very useful in work with algebraic structure like groups or rings. SSReflect contains a library `ssralg` which, by using this mechanism, provides a hierarchy of algebraic structures including monoid, group, ring and field. In the libraries on matrices and polynomials, we define the types of polynomials and matrices over a structures of rings. Rings structures are then defined on these types. Thereby, the same operations of rings (addition, multiplication and opposite) are used for the matrices and the polynomials. With the definition of the `Canonical Structure` on these structures, Coq will be able to infer automatically the structure of ring on polynomials and matrices. This enables us to have statements close to those used in standard mathematics and more readable from the point of view of the user.

4.3. Matrices and determinant

A matrix on a ring R is a double indexed list of coefficients. It can be seen as a function which associates a position (i, j) to a value in the ring R. Being given m and n two naturals and R a ring, a matrix on R (an object of the type $M_{m,n}(R)$) can be represented by the following function: $[0..n[\to [0..m[\to R$. To define a `eqType` on the matrices, which are functions, we need the extensionnality. The function `fgraphType` builds the graph of a function whose domain is a `finType` and the Co-domain a `eqType`. With the definition of the graphs of functions, the functions are now equipped with a Leibniz equality. For two functions `f` and `g`, the notations `f =1 g` and `f =2 g` correspond respectively to $\forall$`x`, `f x` $=$ `g x` and $\forall$`x y`, `f x y` $=$ `g x y`. If the two functions have domain of type `finType` and co-domain `eqType`, the previous notations are equivalent to `f = g`. Using those definitions, the type of matrices of size (m, n) is defined by :

```
Definition matrix (m n :nat) :=
```

```
  fgraphType (I_(m) * I_(n)) R.
```

The functions `matrix_of_fun` and `fun_of_matrix` take respectively a function or a matrix and return a matrix or a function. The latter function is not other than a coercion from the type `matrix` to the type of functions. It enables us to say that two matrices `A` and `B` are equal if and only if we have `A =2 B`. That means that the functions associated to these matrices are equal: `fun_of_matrix A =2 fun_of_matrix B`.
In the following, the notations `M_(n)` and `\Z x` correspond respectively to the type of the square matrices and the scalar matrix of `x`. The notation `\matrix_(i,j) E`, where `E` is an expression of `i` and `j`, corresponds to the application of `matrix_of_fun` to the function `f i j => E i j`. For example the scalar matrix corresponding to an element `x` is given by the following formula:

```
Definition scalar_mx n x : M_(n) :=
  \matrix_(i, j) (if i == j then x else 0).
```

The library on the determinants uses the formula of Leibniz to define the determinant. This choice is justified by the fact that this formula is well adapted and we have the necessary components to its formalization. In fact, a formalization of the permutations (group, signature...) was already developed in the development on the finites groups [8]. Being given a square matrix A of size n, the determinant is given by:

$$\det(A) = \sum_{\sigma \in S_n} \epsilon(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)} \tag{5}$$

In this formula, it is a question of indexed sums and products, but the mathematical notations hide several other elements. In the library on the determinant, to be able to formalize the Leibniz formula (5):

- the indexation of lines and columns of the matrix by naturals are replaced by an indexation by the elements of type `I_(n)`,
- the permutations on this finite set is described as a finite set which could be enumerated and then used as index.

With these choices, thanks to the developments on the computation of the parity of permutations and with that on the groups of permutations, the Leibniz formula (5) is written in Coq:

```
Definition determinant n (A : M_(n)) :=
  \sum_(s : S_(n)) (-1) ^ s * \prod_(i) A i (s i).
```

The notations `\sum` and `\prod` respectively represent the indexed sum and the indexed product. They are instances of `reducebig` for the internal operations (addition and multiplication) of the matrix coefficients ring. The notation `S_(n)` represents the group of the permutations on a set with n elements. In the following, the notation `\det` represents the function `determinant`.

To formalize the Cramer rule, the Co-matrix of a matrix is defined using the functions `row'` and `col'`. The first takes in entry a number i lower than m (an element of type `I_(m)`) and a matrix of size (m, n); it return the matrix of size $(m-1, n)$ where the line i was removed. The function `col'` makes the same work that `row'` but for columns. With the same arguments, the function `row` return the matrix of size $(1, n)$ (a line matrix) which contains the line i. The transpose of the Co-matrix is represented by the function `adjugate`. The Cramer rule is formally represented by the lemma `mulmx_adjr`.

```
Definition cofactor n (A : M_(n)) (i j : I_(n)) :=
   (-1) ^+(i + j) * \det (row' i (col' j A)).
Definition adjugate n (A : M_(n)) := \matrix_(i, j) (cofactor A j i).

Lemma mulmx_adjr : forall n (A : M_(n)), A * adjugate A = \Z (\det A).
```

The proof uses the Laplace formula ($\det(a) = \sum_{i=1}^{n} a_{i,J}\mathrm{Co} - \mathrm{matrice}_{i,j}$). This formula is formally given by :

```
Lemma expand_determinant_row : forall n (A : M_(n)) i0,
  \det A = \sum_(j) A i0 j * cofactor A i0 j.
```

The lemma according to which the determinant is an alternate form (the determinant of a matrix, where at least two lines are identical, is null) is formally stated as follows :

```
Lemma alternate_determinant : forall n (A : M_(n)) i1 i2,
  i1 != i2 -> A i1 =1 A i2 -> \det A = 0.
```

Let's remind that the matrices are functions with two arguments. The term `A i1` is a function with an argument and it corresponds to the line `i1` of the matrix `A`.

4.4. Polynomials

A polynomial is formally defined by the list of its coefficients a_i which are elements of a ring R :

$$a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$

To represent a polynomial we need to know there coefficient. They can be represented by a list. But this representation is not unique. For example, the both polynomials $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ and $0x^{n+1} + a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ have different lists of coefficients but they are equal. To have a Leibniz equality on this representation, it is necessary to consider only the normalized polynomials, i.e. those whose coefficient of greater degree is not null, and to have a Leibniz equality on the coefficients. We represented the polynomial by the following structure :

```
Structure polynomial (R : ring) : Type := Poly {
  p :>  seq R;
  normal : last 1 p != 0
}.
```

The proposition `normal` says that the last element of the list of the coefficients is not null. With this definition, we can define a structure of `eqType` on the polynomials. After that, we defined the function of coefficients of a polynomial by:

```
Definition coef (p : polynomial) i := sub 0 p i.
```

The function `coef p` is of type `nat -> R`. The following lemma gives the equivalence between the equality between the polynomials and those between the functions of coefficient:

```
Lemma coef_eqP : forall p1 p2, coef p1 =1 coef p2 <-> p1 = p2.
```

With this lemma, we can switch between the structural representation of the polynomials and that which considers only the function of coefficients. The advantage of the second representation is to make

the proof of the algebraic properties of the polynomials more intuitive. For example, the multiplication of two polynomials is defined by :

$$\left(\sum_{i=0}^{n} a_i x^i\right)\left(\sum_{j=0}^{m} b_j x^j\right) = \sum_{k=0}^{m+n}\left(\sum_{i+j=k} a_i b_j\right) x^k. \tag{6}$$

The proof of the associativity of this multiplication can be obtained with reasoning on indexed sums, without needing to make recurrences on the polynomials.
In the following, the notations `\X` and `\C c` correspond respectively to the monomial x and the constant polynomial c.
The operation of multiplication of a polynomial by `x` (shift on the right) and addition of a constant is one of the basic operations on the polynomials. In the library it is given by the following function :

```
Definition horner c p : polynomial :=
  if p is Poly (Adds _ _ as s) ns then Poly (ns : normal (Adds c s)) else \C c.
```

From this definition, the function of construction of a polynomial starting from a list of coefficients is simply defined by :

```
Definition mkPoly := foldr horner \C0.
Notation "\poly_ ( i < n ) E" := (mkPoly (mkseq (fun i : nat => E) n)).
```

The notation `\poly` corresponds to the function `mkPoly`. For example, the polynomial corresponding to `\poly_ ( i < n ) i` is $n - 1x^{n-1} + \cdots + 1x + 0$.

The basic operations on the polynomials are defined by recurrence on the list of the coefficients. The list result is then normalized by the function `mkPoly`. For example, the multiplication of two polynomials is defined as follows:

```
Fixpoint mult_poly_seq (s1 s2 : seq R) {struct s1} : seq R :=
  if s1 is Adds c1 s1' then
    add_poly_seq (maps (fun c2 => c1 * c2) s2)
                 (Adds 0 (mult_poly_seq s1' s2))
  else seq0.

Definition mult_poly (p1 p2 : polynomial) := mkPoly (mult_poly_seq p1 p2).
```

In the second definition, with the coercion from `polynomial` to `seq` we can write `mult_poly_seq p1 p2` although `p1` and `p2` are of type `polynomial`. The lemma `coef_mul_poly`

```
Lemma coef_mul_poly : forall p1 p2 i,
  coef (mult_poly p1 p2) i = \sum_(j <= i) coef p1 j * coef p2 (i - j).
```

give a relation between the coefficients of two polynomials and those of the result of their multiplication. It corresponds to the relation of the formula (6).

Another important operation on the polynomials is the evaluation function. It consists in replacing the variable by a given value in an R-algebra. This function can be described with the diagram of Horner for a polynomial p and a value x by:

$$p(x) = ((...((a_n x + a_{n-1})x + a_{n-2})x + ...) + a_1)x + a_0 \tag{7}$$

The diagram (7) corresponds to a recurrence on the list of the coefficients. According to this diagram the evaluation depends only on the list of the coefficients and the value where we evaluate. The evaluation function can be defined by recurrence on an arbitrary list as follows :

```
Fixpoint eval_poly_seq (s : seq R) (x : R) {struct s} : R :=
  if s is (Adds a s') then eval_poly_seq s' x * x + a else 0.
```

Let us recall that in the definition of the structure `polynomial` we have a coercion between it and the type of the list of the coefficients. This enables us to define the evaluation of a polynomial in the following way:

```
Definition eval_poly (p : polynomial R) : R-> R := eval_poly_seq p.
```

In our formalization of the function of evaluation, the value where the polynomial is evaluates must be of the same type as its coefficients. In our case this choice does not pose a problem considering which one can canonically inject a polynomial on a ring R towards a polynomial on the ring of the matrices on this same ring R. The notation `p.[c]` corresponds to the application of the function `eval_poly` to a polynomial `p` and a value `c`.
The properties of morphism of the evaluation function are used implicitly in the proof of the theorem of Cayley-Hamilton. These properties are given by the following lemmas:

```
Lemma eval_polyC : forall c x, (\C c).[x] = c.
Lemma eval_poly_plus : forall p q x, (p + q).[x] = p.[x] + q.[x].
Lemma eval_poly_mult : forall p q x, x * q.[x] = q.[x] * x ->
                                     (p * q).[x] = p.[x] * q.[x].
```

In the lemma `eval_poly_mult` on the evaluation of a product of polynomials, it is necessary to have that the value `x` where the polynomial `p * q` is evaluates commutates with the evaluation of `q` in this same value. This assumption is necessary because we evaluate polynomials on a no-commutative ring: the ring of the matrices.

After these developments, the factor theorem can be stated as follows:

```
Theorem factor_theorem : forall p c,
  reflect (exists q, p = q * (\X - \C c)) (p.[c] == 0).
```

In the proof of this theorem, to be able to say that `p.[c]` is equal to `q.[c ] * (\X - \C c).[c]`, we should prove that the coefficients of the polynomial `(\X - \C c)` commutate with `c`. What is proved easily because `1` and `c` commutate with `c`.

4.5. Proof of the Cayley-Hamilton theorem

The morphism between the ring of the matrix of polynomials and that of the polynomials of matrices is the central part of the proof of the Cayley-Hamilton theorem. The other components of the proof, the Cramer rule and the factor theorem, are properties which are attached respectively to the matrices and the polynomials.
This morphism that we will call `phi` takes in argument a matrix of polynomials and return a polynomial of matrices. The length of the list of the coefficients of the result (a polynomial) is the maximum size of the polynomials of the input matrix. The size of a polynomial corresponds to the length of the list of these coefficients, in other term its degree plus one. For a matrix of polynomials `A`, `phi A` is the polynomials of matrices whose coefficient of index `k` is the matrix whose coefficient in `i` and `j` is `coef (A i j) k`.

In the following, the notations `R[X]`, `M(R)`, `M(R[X])` and `M(R)[X]` respectively represent the ring of the polynomials, that of the matrices, that of the matrices of polynomials and that of the polynomials of matrices.

```
Definition phi (A : M(R[X])) : M(R)[X] :=
  \poly_(k < \max_(i) \max_(j) size (A i j)) \matrix_(i, j) coef (A i j) k.
```

The lemma `coef_phi` gives a relation between a matrix of polynomials and its image by `phi`.

```
Lemma coef_phi : forall A i j k, coef (phi A) k i j = coef (A i j) k.
```

To be able to define the evaluation of the characteristic polynomial of a matrix in it, we defined the canonical injection of the ring of the polynomials into that of the polynomials of matrices. This injection follows from that of the ring of the coefficients into that of the matrices on this same ring, in other words the multiplication by the identity matrix.

```
Definition Zpoly (p : R[X]) : M(R)[X] := \poly_(i < size p) \Z (coef p i).
```

The characteristic polynomial of a matrix is defined by applying the determinant function to the matrix $xI_n - A$ which is a matrix of polynomials.

```
Definition matrixC (A : M(R)) : M(R[X]) := \matrix_(i, j) \C (A i j).
Definition char_poly (A : M(R)) : R[X] := \det (\Z \X - matrixC A).
```

The function `matrixC` is the canonical injection of the ring of matrices into that of matrices of polynomials.

After these definitions, the Cayley-Hamilton theorem is proved formally in the following way:

```
Theorem Cayley_Hamilton : forall A, (Zpoly (char_poly A)).[A] = 0.
Proof.
move=> A; apply/eqP; apply/factor_theorem.
rewrite -phi_Zpoly -mulmx_adjl phi_mul; move: (phi _) => q; exists q.
by rewrite phi_add phi_opp phi_Zpoly phi_polyC ZpolyX.
Qed.
```

The proof proceeds exactly as described in the second section. After applying the factor theorem, the factor polynomial is given while rewriting with the Cramer rule (`mulmx_adjl`). The result of the Cayley-Hamilton theorem is then proved by rewritings and simplifications in the term obtained. The use of SSReflect, of the mechanisms of `Canonical Structure` and notations of Coq, as well as the definition of a hierarchical algebraic structures allowed us to have a such concise proof: 3 lines of codes.

5. Conclusion

In this paper,we presented a formalization of the Cayley-Hamilton theorem which adopts a modular approach. The proof that we presented in section 4.4 can appear very simple; but the design was being rather long specially in the choose of the architecture of the proof and the types for representing the data structure we manipulate (polynomials and matrices). The choices were justified by preoccupations of readability and reutilisability. The use of the Canonical Structure of Coq enabled us to have statements close to those used in usual mathematics. The cutting of the various components

of the proof in modular form (indexed operations, matrices and polynomials) promotes the re-use of these libraries in independent developments. The libraries on the indexed operations and the matrices will be re-used in our next work on the characters theories, one of the components of the proof of the Feit-Thompson theorem.
This work that we presented here is the first formalization of the Cayley-Hamilton theorem. It is not the first formalization of the matrices or the polynomials. Formalizations of these structures are respectively presented in [9, 11] and [5, 10]. But it is the first development which gathers a formalization of the matrices and polynomials and where these objects are assembled to form new objects: matrices of polynomials and polynomials of matrices.

In the formalization of the Cayley-Hamilton theorem, presented in this article, we chose to build our structures of data on types with a decidable equality: the `eqType`. In addition to the fact that in traditional mathematics all the types are decidable, our proof on the types where the equality is decidable can be generalized to any types. This is done by pointing out that any ring is a $\mathbf{z}$-algebra and by considering the morphism of evaluation of polynomials with n^2 variables and coefficients in $\mathbf{z}$ which is a type where the equality is decidable. We are going to have to work with the `Setoid`.

References

[1] Paul et Jack Abad, *The Hundred Greatest Theorems*, Available at http://personal.stevens.edu/ nkahl/Top100Theorems.html.

[2] Jeremy Avigad, Kevin Donnelly, David Gray, et Paul Raff, *A Formally Verified Proof of the Prime Number Theorem*, ACM Transactions on Computational Logic, A paratre.

[3] Yves Bertot, Pierre Castran, *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*, Springer Verlag, 2004.

[4] Nathan Jacobson, *Lectures in Abstract Algebra: II. Linear Algebra*, Springer Verlag, 1975.

[5] Herman Geuvers, Freek Wiedijk et Jan Zwanenburg, *A Constructive Proof of the Fundamental Theorem of Algebra without Using the Rationals*, Types for Proofs and Programs, TYPES 2000 International Workshop, Selected Papers, volume 2277 of LNCS, pages 96-111, 2002.

[6] Georges Gonthier, *A computer-checked proof of the four-colour theorem*, Available at http://research.microsoft.com/ gonthier/4colproof.pdf.

[7] Georges Gonthier, Assia Mahboubi, *A small scale reflection extension for the Coq system*, Available at http://www.msr-inria.inria.fr/ assia/rech-eng.html.

[8] Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi et Laurent Thry, *A Modular Formalisation of Finite Group Theory*, Rapport de Recherche 6156, INRIA, 2007.

[9] Nicolas Magaud, *Ring properties for square matrices* contribution to Coq, http://coq.inria.fr/contribs-eng.html.

[10] Piotr Rudnicki, *Little Bezout Theorem (Factor Theorem)*, Journal of Formalized Mathematics volume 15, 2003, Available at http://mizar.org/JFM/Vol15/uproots.html.

[11] Jasper Stein, *Linear Algebra* contribution to Coq, http://coq.inria.fr/contribs-eng.html.

[12] Coq Team, *The Coq reference manual V 8.1*, http://coq.inria.fr/V8.1/refman/index.html.

[13] Freek Wiedijk, *Formalizing 100 Theorems*, http://www.cs.ru.nl/freek/100/.