

# Formalisation des mathématiques : une preuve du théorème de Cayley-Hamilton \*

---

Sidi Ould Biha <sup>1</sup>

*1: Inria de Sophia-Antipolis,  
2004, route des Lucioles - B.P. 93 06902 Sophia Antipolis Cedex, France  
Sidi.Ould\_biha@sophia.inria.fr*

*\*: Ce travail a été possible grâce au financement du laboratoire commun Microsoft-Inria  
<http://www.msr-inria.inria.fr>*

## 1. Introduction

Les systèmes de preuves formelles peuvent être d'une grande utilité dans la vérification et la validation de preuves mathématiques, surtout lorsque ces preuves sont complexes et longues. Les travaux récents, comme la preuve formelle du théorème des 4 couleurs [6] ou celle du théorème des nombres premiers [2], montrent que ces systèmes ont atteint un niveau de maturité leur permettant de s'attaquer à des problèmes mathématiques non triviaux. Le travail de formalisation de preuves mathématiques faisant intervenir une large variété d'objets mathématiques nécessite l'adoption d'une approche semblable au génie logiciel. La formalisation de telles théories peut être vue comme un développement faisant intervenir différentes composantes : définitions et preuves mathématiques.

Une liste des 100 plus grands théorèmes mathématiques [1] a été constitué par Paul et Jack Abad. Cette liste prend en compte la place du théorème dans la littérature mathématique, la qualité de sa preuve et l'importance du résultat qu'il introduit. F. Wiedijk maintient une liste [13] qui recense les formalisations de ces théorèmes dans des systèmes de preuves formelles. Le théorème de Cayley-Hamilton, est l'un des théorèmes présents cette liste. Ce papier présente une formalisation de ce théorème, qui est à notre connaissance la première. Le fait qu'il n'avait pas été jusqu'à ce jour formalisé peut s'expliquer par le fait qu'il fait intervenir de nombreux objets et propriétés mathématiques. Ces objets ne sont pas uniquement utilisées, de façon indépendante; mais ils sont aussi emboîtés les uns sur les autres. Ce travail de formalisation du théorème de Cayley-Hamilton est utilisé dans le cadre des travaux de formalisation du théorème de Feit-Thompson sur les groupes d'ordre impaires. L'objectifs n'est pas seulement de formaliser Cayley-Hamilton; mais d'avoir en plus des bibliothèques réutilisables.

L'article est organisé comme suit. Dans la section 2, nous présentons l'énoncé et la preuve du théorème de Cayley-Hamilton. Dans la section 3, nous présentons SSREFLECT, l'extension de COQ et plate-forme de notre développement. Enfin, dans la section 4, nous présentons le développement qui a été nécessaire pour arriver à la formalisation du théorème de Cayley-Hamilton.

## 2. Le théorème de Cayley-Hamilton

Le théorème de Cayley-Hamilton [4] peut être énoncé de la façon suivante :

*Toute matrice carrée sur un anneau commutatif annule son polynôme caractéristique.*

Plus formellement, soient  $R$  un anneau commutatif et  $A$  une matrice carrée sur  $R$ . Alors, Le polynôme caractéristique de  $A$ , défini par :  $p_A(x) = \det(xI_n - A)$ , s'annule en  $A$ .  
Le théorème peut être énoncé différemment en considérant les endomorphismes d'espace vectorielle. Dans ce cas il n'est plus question d'anneau commutatif mais de corps.

La preuve du théorème de Cayley-Hamilton présentée dans [4] découle de la formule de Cramer. En notant  ${}^t\text{com}B$  la transposée de la co-matrice de  $B$ , la règle de Cramer est :

$$B * {}^t\text{com}B = {}^t\text{com}B * B = \det B * I_n \quad (1)$$

En appliquant la formule (1) à la matrice  $(xI_n - A) \in M_n(R[x])$ , nous obtenons :

$${}^t\text{com}(xI_n - A) * (xI_n - A) = \det(xI_n - A) * I_n = p_A(x) * I_n \quad (2)$$

Comme  $M_n(R[x]) = (M_n(R))[X]$  ( $M_n(R[x])$  et  $(M_n(R))[X]$  représentent respectivement l'anneau des matrices de polynômes et celui des polynômes de matrices sur un anneau  $R$ ), l'égalité (2) s'écrit dans  $(M_n(R))[X]$  :

$${}^t\text{com}(xI_n - A) * (X - A) = p_A(X) \quad (3)$$

Ceci montre que  $(X - A)$  est facteur de  $p_A(X)$  dans  $(M_n(R))[X]$  et donc  $p_A(A) = O_n$ .

Formaliser une preuve mathématique dans un assistant à la preuve consiste à développer cette preuve pour qu'elle soit compréhensible pour un ordinateur. Pour arriver à cet objectifs deux difficultés sont à surmontées. En premiers lieu, il faut expliciter les parties de la preuve qui sont implicites ou triviales pour un mathématicien. Paradoxalement, l'implémentation sur ordinateur de ces parties, qui n'apparaissent pas dans la preuve, est la tâche la plus complexe dans ce travail de formalisation. En second lieu, il faut avoir des formules et des scripts de preuves compréhensible pour un mathématicien. Les énoncés et preuves doivent être le plus proche possible de ceux utilisés de façon standard en mathématiques.

Dans le cas du théorème de Cayley-Hamilton et en considérant la preuve ci-dessus plusieurs problèmes se posent lors de sa formalisation. Dire que  $M_n(R[x])$  est identique à  $(M_n(R))[x]$  équivaut algébriquement à dire qu'il existe un isomorphisme d'anneau entre eux. En effet, toute matrice de polynômes peut s'écrire, de façon unique comme la somme de puissances en  $x$  multipliées par des matrices, c'est-à-dire un polynôme à coefficients matriciels. Par exemple :

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

La formalisation de cette isomorphisme correspond à l'écriture de la fonction de transformation décrite dans l'exemple ci-dessus. En notant  $\phi$  cette isomorphisme, ses propriétés de morphisme sont utilisées implicitement dans la preuve. En effet, dans (2) les membres de l'égalité sont des matrices de polynômes. L'application de  $\phi$  aux parties gauche et droite de (2) nous donne :

$$\phi({}^t\text{com}(xI_n - A) * (xI_n - A)) = \phi(p_A(x) * I_n)$$

Les propriétés de morphisme de  $\phi$  sont alors utilisées pour obtenir :

$$\phi({}^t\text{com}(xI_n - A)) * \phi(xI_n - A) = \phi(p_A(x) * I_n)$$

La formule (3) correspond explicitement à l'égalité ci-dessus.

### 3. SSREFLECT

SSREFLECT [7, 8] (pour *Small Scale Reflection* ou réflexion à petite échelle) est une extension de COQ [3] qui introduit de nouvelles tactiques et des bibliothèque COQ adaptées pour travailler sur des types avec une égalité décidable. Elle a été initialement développée par G. Gonthier dans le cadre de sa preuve du théorème des 4 couleurs. Un développement [8] sur la théorie des groupes finis a été fait au dessus de SSREFLECT. Ce développement comprend, entre autre, une formalisation du théorème de Sylow et du lemme de Cauchy-Frobenius.

Nous allons présenter en premier lieu la méthode de SSREFLECT pour avoir la réflexion entre les prédicats décidables et les booléens. Nous allons introduire par la suite quelques bibliothèques COQ de SSREFLECT que nous avons utilisées dans notre développement. Des informations plus détaillées sur SSREFLECT et précisément son langage de tactique peuvent être obtenues dans [7].

#### Réflexion

Dans le système de preuve COQ la logique par défaut est intuitionniste. Dans cette logique, les propositions logiques et les valeurs booléennes sont distinctes. Lorsque nous travaillons avec des types décidables cette distinction n'a pas lieu d'être. SSREFLECT permet de combiner le meilleur des deux visions et de passer de la version propositionnelle d'un prédicat décidable vers la version booléenne. Pour ce faire, le type booléen est injecté dans celui des propositions par une coercion :

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

Ainsi, et de façon transparente pour l'utilisateur, lorsque COQ attend un objet de type `Prop` et reçoit une valeur `b` de type `bool`, il la traduira automatiquement en la proposition `(is_true b)`, qui correspond à la proposition `b = true`.

Le prédicat inductif `reflect` donne une équivalence pratique et confortable entre les propositions décidables et les booléens :

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

La proposition `(reflect P b)` indique que `P` est équivalent à `(is_true b)`. Par exemple, l'équivalence entre la conjonction booléenne `&&` et celle propositionnelle `/\` est donnée par le lemme suivant :

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

Dans ce lemme `a` et `b` sont des variables booléennes.

#### Bibliothèques

Des bibliothèques de base sont définies dans SSREFLECT. C'est une hiérarchie de structure pour travailler avec les types décidables et en particulier les types finis. La structure `eqType` définit les types munis d'une égalité décidable et de Leibniz.

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

Le symbole `>` déclare `sort` comme une coercion d'un `eqType` vers son type porteur. C'est la technique standard de sous-typage, tout objet de type `eqType` est aussi de type `Type`. La structure `eqType` ne suppose pas seulement l'existence d'une égalité décidable `==`, en plus elle injecte cette égalité vers celle de Leibniz.

Une propriété majeure des structures d'`eqType` est qu'elles donnent la propriété de la *proof-irrelevance* pour les preuves d'égalités de leurs éléments. Ainsi il n'y a qu'une seule preuve de l'égalité pour chaque paire d'objets égaux.

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

Le type des listes sur un `eqType` `d` se définit de façon inductive par :

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

`Adds` et `Seq0` correspondent respectivement aux constructeurs `cons` et `nil` du type standard `list` de Coq. Le type `seq d` définit les listes sur un `eqType` `d`. La fonction `foldr` correspond dans SSREFLECT à l'opération *fold* utilisée en programmation fonctionnelle. Elle est définie par récurrence sur une séquence. L'opération d'extraction d'un élément d'une liste est donnée par la fonction `sub`. Par exemple `sub x0 s i` retourne l'élément d'indice `i` de la séquence `s`, si `i` est strictement inférieure à la longueur de la séquence, et `x0` dans le cas contraire. La fonction `mkseq` permet de construire une séquence de longueur donnée à partir d'une fonction sur les entiers. Par exemple, `mkseq f n` correspond à la séquence `[(f 0), (f 1), ..., (f n-1)]`.

La définition du type liste sur un `eqType` est à la base de celle des types finis. La structure `finType` se compose d'une liste sur un `eqType` et de la preuve qu'aucun élément de cette liste n'apparaît plus d'une fois.

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

Dans cette définition `(set1 x)` est l'ensemble singleton `x` et `(count f 1)` calcule le nombre d'éléments `y` de la liste `l` pour lesquels `(f y)` est vraie. Le paramètre `enum` correspond à la liste des éléments du type fini. Par exemple pour un `finType` `d`, `(enum d)` retourne la liste des éléments de `d`.

Pour représenter les types finis à `n` éléments, la bibliothèque SSREFLECT fournit une famille de types nommée `ordinal` dont les éléments sont des paires composées d'un nombre entier `p` et d'une preuve que `p` est inférieur à `n`. Comme cette preuve est basée sur un test décidable, la propriété d'irrelevance s'applique et les éléments de ce type sont uniquement caractérisés par la composante `p`. La notation `I_(n)` désigne le type `ordinal n`.

## 4. Formalisations Coq

Dans Ce travail de formalisation du théorème de Cayley-Hamilton, nous utilisons des bibliothèques sur les opérations indexées et les déterminants. Ces bibliothèques ont été développées par Y. Bertot et G. Gonthier dans le cadre du projet "Mathematical Components". SSREFLECT contient aussi une bibliothèque qui fournit une formalisation des structures algébriques de monoïde, groupe et anneau. L'utilisation de cette bibliothèque et le mécanisme des Canonical Structure de Coq nous ont permis d'avoir des notations proches de celles utilisées de façon standard en mathématiques. La bibliothèque

sur les polynômes fournit une formalisation des propriétés algébrique des polynômes, du morphisme d'évaluation et du théorème du reste. La définition de l'isomorphisme entre l'anneau des matrices de polynômes et celui des polynômes de matrices est l'étape ultime de la formalisation du théorème de Cayley-Hamilton.

#### 4.1. Les opérations indexées

Dans la définition des opérations sur les matrices, par exemple la multiplication ou le calcul du déterminant, les opérations indexées (somme et produit) sont fréquentes. Factoriser la preuve de propriétés générales sur les sommes et produits indexés permet de réduire considérablement la longueur et la complexité des preuves. Une bibliothèque pour les opérations indexées n'est pas seulement utile dans le développement sur la théorie des matrices, elle pourra l'être aussi dans des développements plus générales que l'algèbre linéaire.

Une opération indexée est la généralisation de la définition d'une opération binaire aux éléments d'une suite finie. Dans le cas particulier de l'addition, c'est la somme de tous les éléments d'une suite donnée.

La définition d'une opération indexée est donnée par :

**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.`

La fonction `reducebig` a comme paramètres un type quelconque `R`, un `eqType` `I`, une opération binaire `op` sur `R`, un élément `nil` de `R` correspondant à l'ensemble vide, une séquence `r` sur `I`, une propriétés caractéristique `P` sur `I` (une fonction de `I` vers `bool` : un ensemble sur `I`) et une fonction `F` de `I` vers `R`. Le résultat de `reducebig` correspond schématiquement à :

$$F p_1 \text{ op } F p_2 \text{ op } \cdots \text{ op } F p_n \text{ op } \text{nil},$$

Les  $p_i$  sont les éléments de la séquence  $r$  pour lesquels la propriété  $P$  est vraie : les éléments de l'ensemble  $P$ . L'utilisation de `reducebig` est plus naturelle lorsque l'opération est associative et commutative et lorsque `nil` est l'élément neutre de cette opération. En d'autres termes, lorsque `op` et `nil` ont des propriétés de monoïde.

La notation `\big[*M/1](i | P i) F i` correspond à l'application de `reducebig` à une opération `*` qui a pour élément neutre `1` et qui a des propriétés de monoïde. Le reste des paramètres sont inférés de façon implicite par COQ.

Un lemme intéressant sur `reducebig` est celui qui permet d'effectuer l'opération usuelle de ré-indexation. Dans le cas d'une somme indexée, ce lemme correspond à l'égalité entre les sommes  $\sum_{i=0}^n (i + m)$  et  $\sum_{j=m}^{n+m} j$ . Une façon de formaliser cette égalité est de considérer que  $i$  et  $j$  sont de types différents et qu'il existe une bijection entre eux. Cette bijection est la fonction  $f : x \leftarrow x + m$ . Le prédicat `ibjective P h` permet de dire que la fonction `h` est bijective sur l'ensemble `P`. Le lemme de ré-indexation s'énonce alors comme suit :

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

Un autre résultat intéressant sur les opérations indexées est celui qui permet de décomposer cette opération suivant une partition de l'ensemble d'indice. Par exemple, dans le cas d'une somme indexée, le résultat s'écrit  $\sum_{i=0}^{n+m} i = \sum_{i=0}^n i + \sum_{i=n+1}^{n+m} i$ . La généralisation de cette propriété peut s'écrire formellement :

```

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.

```

Dans ce lemme, étant donnés deux ensembles  $P$  et  $a$ , une partition de  $P$  est donnée par les deux ensembles  $P \cap a$  et  $P \cap \bar{a}$ . La somme des éléments indexés par  $P$  peut être donc décomposée en deux sommes des éléments indexés par ces deux ensembles.

## 4.2. Structures canoniques

Dans l'assistant de preuve COQ, le mécanisme des **Canonical Structure** permet de définir une instance d'un type structurel (mot clé COQ **Record** ou **Structure**) qui pourra être utilisée lors du processus d'inférence de type dans des équations invoquant des arguments implicites. Par exemple, pour définir une structure de **eqType** sur le type **nat** il faut une égalité décidable sur les entiers et prouver que cette égalité est équivalente à celle de Leibniz sur les entiers.

```

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).

```

Grâce à la définition de **Canonical Structure nat\_eqType**, les arguments  $m$  et  $n$  vont être typés comme des éléments d'un **eqType**, il sera donc correcte de considérer l'égalité booléenne entre eux.

```

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

```

Le mécanisme des **Canonical Structure** peut être appliquée à des structures plus complexes comme les anneaux. Dans les bibliothèques sur les matrices et les polynômes, nous définissons les types de polynômes et matrices sur des structures d'anneaux. Des structures d'anneaux sont ensuite définies sur ces types. Ainsi les mêmes opérations d'anneaux (addition, multiplication et opposé) sont utilisées pour les matrices et les polynômes. Avec la définition des **Canonical Structure** sur ces structures, COQ pourra inférer automatiquement que la structure d'anneau sur les polynômes et les matrices. Ceci nous permet d'avoir des énoncés proches de ceux en mathématique standard et plus lisible du point de vue de l'utilisateur.

## 4.3. Matrices et déterminants

Une matrice sur un anneau  $R$  est une séquence de coefficients doublement indexée. Elle peut être vue comme une fonction qui associe à une position  $(i, j)$  une valeur dans l'anneau  $R$ . Étant donnés  $m$  et  $n$  deux entiers et  $R$  un anneau, une matrice sur  $R$  (un objet de type  $M_{m,n}(R)$ ) peut être représentée par la fonction suivante :  $[0..n \rightarrow [0..m \rightarrow R]$ . Le type des matrices de taille  $(m, n)$  est défini par :

```

Definition matrix (m n : nat) :=
  fgraphType (I_(m) * I_(n)) R.

```

Dans cette définition, les matrices sont des fonctions à deux arguments. La fonction `fgraphType` construit le graphe des fonctions dont le domaine est un type fini et le co-domaine un `eqType`. Avec la définition des graphes de fonctions, les fonctions sont ainsi munies d'une égalité de Leibniz et ceci sans avoir besoin de nouveaux axiomes. Pour deux fonctions `f` et `g`, les notations `f =1 g` et `f =2 g` équivalent respectivement à  $\forall x, f\ x = g\ x$  et  $\forall x\ y, f\ x\ y = g\ x\ y$ .

Les fonctions `matrix_of_fun` et `fun_of_matrix` permettent respectivement de définir un objet de type `matrix` à partir d'une fonction et de convertir un objet de type `matrix` en une fonction à deux arguments. Cette dernière n'est autre qu'une coercion du type `matrix` vers celui des fonctions. Elle nous permet de dire que deux matrices `A` et `B` sont égales si et seulement si nous avons `A =2 B`. Ce qui veut dire que les fonctions associées à ces matrices sont égales : `fun_of_matrix A =2 fun_of_matrix B`.

Dans la suite, les notations `M_(n)` et `\Z` correspondent respectivement au type des matrices carrées et à la multiplication d'un scalaire par la matrice identité.

La bibliothèque sur les déterminants utilise la formule de Leibniz pour définir le déterminant. Ce choix est motivé par le fait que dans cette formule il est question de groupe de permutations et qu'une formalisation des permutations (groupe, signature ...) a été déjà développée dans le cadre du travail sur les groupes finis [8]. Etant donné une matrice carrée  $A$  de dimension  $n$ , le déterminant est donnée par :

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

Dans cette formule, il est question d'opérations indexées pour les sommes et les produits, mais les notations mathématiques cachent plusieurs autres éléments. Dans la bibliothèque sur les déterminant, pour pouvoir formaliser la formule (5) :

- l'indexation des lignes et colonnes de la matrice par des entiers est remplacée par une indexation par les éléments du type `I_(n)`,
- l'ensemble des permutations sur cet ensemble fini est décrit comme un ensemble fini qui pourra être énuméré.

Avec ces choix, grâce aux développements sur le calcul de la parité des permutations et à celui sur les groupes de permutations, la formule (5) s'écrit :

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

Les notations `\sum` et `\prod` représentent respectivement la somme et le produit indexées. Ce sont des instances de `reducebig` pour les opérations internes (addition et multiplication) de l'anneau des coefficients de la matrice. La notation `S_(n)` représente le groupe des permutations de taille  $n$ . Dans la suite, la notation `\det` représentera la fonction `determinant`.

Pour exprimer la règle de Cramer, la co-matrice d'une matrice est définie à l'aide de la fonction `row'`. Cette dernière prend en entrée un nombre  $i$  inférieur à  $m$  (un élément de type `I_(m)`) et une matrice  $(m, n)$ ; elle retourne la matrice  $(m - 1, n)$  où la rangée  $i$  a été enlevée. Avec les mêmes arguments, la fonction `row` retourne la matrice  $(1, n)$  (une rangée et  $n$  colonnes) qui contient la rangée  $i$ . La transposée de la co-matrice est représentée par la fonction `adjugate`. L'égalité de Cramer s'énonce alors :

```
Definition cofactor n (A : M_(n)) (i j : I_(n)) :=
```

$(-1)^{i+j} \cdot \det(\text{row}' i (\text{col}' j A))$ .

Definition `adjugate n (A : M_(n)) := \matrix_(i, j) (cofactor A j i)`.

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

La preuve utilise la formule de Laplace ( $\det(A) = \sum_{i=1}^n a_{i,j} \text{co-matrice}_{i,j}$ ) sur les co-matrices et le lemme selon lequel le déterminant est une forme alternée (le déterminant d'une matrice, où au moins deux lignes sont identiques, est nul). Ces propriétés sont données respectivement par les lemmes suivants :

Lemma `expand_determinant_row : forall n (A : M_(n)) i0,`

`\det A = \sum_(j) A i0 j * cofactor A i0 j.`

Lemma `alternate_determinant : forall n (A : M_(n)) i1 i2,`

`i1 != i2 -> A i1 =1 A i2 -> \det A = 0.`

Le terme `A i1 =1 A i2` correspond à l'égalité entre les lignes d'indice `i1` et `i2` de la matrice `A`.

#### 4.4. Polynômes

Un polynôme est défini par la liste de ses coefficients  $a_i$  qui appartiennent à un anneau  $R$  :

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

Cette représentation n'est malheureusement pas unique, en effet les polynômes  $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$  et  $0x^{n+1} + a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$  ont des listes de coefficients différentes mais représentent le même polynôme. Pour avoir une égalité de Leibniz pour cette représentation, il est nécessaire de ne considérer que les polynômes normalisés, c'est-à-dire ceux dont le coefficient de plus grand degré est non nul, et avoir une égalité de Leibniz sur les coefficients. Les polynômes sont donc représentés par la structure suivante :

Structure `polynomial (R : ring) : Type := Poly {`

`p :> seq R;`

`normal : last 1 p != 0`

`}.`

Avec cette définition nous pouvons donc définir une structure de `eqType` sur les polynômes. Mais il est parfois utile de voir les polynômes juste comme une fonction qui donne les coefficients. Nous avons donc défini la fonction `coefficient` des polynômes par :

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

La fonction `coef p` est de type `nat -> R`. Le lemme suivant permet d'avoir l'équivalence entre l'égalité entre les polynômes et celles entre les fonctions de coefficient :

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

Avec ce lemme, nous pouvons passer de notre représentation structurelle des polynômes vers celle qui ne considère que la fonction des coefficients. L'avantage de la second représentation est de rendre les preuves des propriétés algébriques des polynômes plus intuitives. Par exemple, la multiplication de deux polynômes est définie par :

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



La preuve de l'associativité de cette multiplication se ramène à des raisonnements sur des sommes indexées, sans avoir besoin de faire des récurrences sur les polynômes.

Dans la suite, les notations  $\backslash X$ ,  $\backslash C\ c$  et  $\backslash C0$  correspondent respectivement au monôme  $x$ , au polynôme constant  $c$  et au polynôme nul (la séquence vide).

L'opération de multiplication d'un polynôme par  $x$  (décalage à droite) et addition d'une constante est l'une des opérations de base sur les polynômes. Dans la bibliothèque elle est réalisée par la fonction suivante :

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

A partir de cette définition, la fonction de construction d'un polynôme à partir d'une liste de coefficient se définit simplement par :

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

La notation  $\backslash poly$  permet de construire un polynôme à partir d'une fonction de coefficients. Par exemple, le polynôme correspondant à  $\backslash poly_ ( i < n ) i$  est le suivant :

$$n - 1x^{n-1} + \dots + 1x + 0$$

Les opérations de base sur les polynômes sont définies par récurrence sur la séquence des coefficients. La séquence résultat est ensuite normalisée par la fonction  $mkPoly$ . Par exemple la multiplication de deux polynômes est définie comme suit :

```
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).
```

Dans la seconde définition, la conversion de type entre les types `polynomial` et `seq` permet d'écrire `mult_poly_seq p1 p2` bien que `p1` et `p2` sont de type `polynomial`. Le lemme `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).
```

donne une relation entre les coefficients des deux polynômes et le résultat de leur multiplication. Il correspond à la relation de la formule (6).

Une autre opération importante sur les polynômes est la fonction d'évaluation d'un polynôme. Elle consiste à remplacer sa variable par une valeur donnée. Cette fonction peut être décrite avec le schéma de Horner pour un polynôme  $p$  et une valeur  $x$  par :

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

Suivant le schéma (7) l'évaluation ne dépend que de la séquence des coefficients et de la valeur où nous évaluons. Elle se définit par récurrence sur la séquence des coefficients. La fonction d'évaluation peut être définie par récurrence comme suit :

```

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

```

La notation  $p.[c]$  correspond à l'application de la fonction `eval_poly` en  $p$  et  $c$ .  
 Les propriétés de morphisme de la fonction d'évaluation sont utilisées implicitement dans la preuve du théorème de Cayley-Hamilton. Ces propriétés sont données par les lemmes suivants :

```

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,
  (forall i, (coef q i) * x = x * (coef q i)) ->
  (p * q).[x] = p.[x] * q.[x].

```

Dans le second lemme, la condition de commutativité entre les coefficients de  $q$  et  $x$  est nécessaire dans le cas d'un anneau non commutatif (les matrices par exemples).

$$\text{Soient } p(x) = \sum_{i=0}^n a_i x^i \text{ et } q(x) = \sum_{j=0}^m b_j x^j$$

$$(p * q)(x) = \sum_{k=0}^{m+n} \left( \sum_{i+j=k} a_i b_j \right) x^k = \dots a_i b_j x^{(i+j)} \dots \quad (8)$$

$$p(x) * q(x) = \left( \sum_{i=0}^n a_i x^i \right) \left( \sum_{j=0}^m b_j x^j \right) = \dots a_i x^i b_j x^j \dots \quad (9)$$

La condition de commutativité s'explique donc par le fait que pour passer de (8) à (9), il faut que tous les  $b_j$  commutent avec  $x$ .

Après ces développements, le théorème du reste peut s'énoncer comme suit :

```

Theorem factor_poly : forall p c,
  (exists q, p = q * (\X - \C c)) <-> (p.[c] = 0).

```

Dans la preuve de ce théorème, pour pouvoir dire que  $p.[c]$  est égale à  $q.[c] * (\X - \C c).[c]$ , il faut prouver que les coefficients du polynôme  $(\X - \C c)$  commutent avec  $c$ . Ce qui se prouve facilement car 1 et  $c$  commutent avec  $c$ .

## 4.5. Preuve de Cayley-Hamilton

Le morphisme entre l'anneau des matrice de polynômes et celui des polynômes de matrices est la partie centrale de la preuve du théorème de Cayley-Hamilton. Les autres composantes de la preuve : la règle de Cramer et le théorème de factorisation, sont des propriétés qui se rattachent respectivement aux matrices et aux polynômes.

Ce morphisme que nous allons appeler **phi** prend en entrée une matrice de polynômes, lui applique le procédé décrit dans (4), et retourne un polynôme de matrices. La longueur de la séquence des coefficients du polynôme résultat est la taille maximale des polynômes de la matrice de départ. La taille d'un polynôme correspond à la longueur de la séquence de ces coefficients, en d'autre terme son degré plus un. Dans la suite, les notations  $M(R)$ ,  $M(R[X])$  et  $M(R)[X]$  représentent respectivement l'anneau des matrices, celui des matrices de polynômes et celui des polynômes de 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.

```

Le lemme donnant la relation entre le résultat de `phi` et sa valeur d'entrée est utilisé dans la preuve des propriétés de morphismes. Ce lemme dit que pour toute matrice de polynômes  $M$ , le coefficient du polynôme  $M_{i,j}$  en  $k$  est égale à l'élément d'indice  $(i,j)$  de la matrice coefficient en  $k$  de l'image de  $M$  par `phi`.

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

Pour pouvoir définir l'évaluation d'une matrice en son polynôme caractéristique, nous avons défini l'injection de l'anneau des polynômes vers celui des polynômes de matrices. Cette fonction prend un polynôme sur un anneau de base  $R$  et multiplie ses coefficients par la matrice identité pour obtenir un polynôme à coefficients matriciels. La notation `Zpoly` correspond à cette fonction.

Le polynôme caractéristique d'une matrice est défini en appliquant la définition du déterminant à la matrice de polynômes  $xI_n - A$ .

**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).

La fonction `matrixC` est l'injection canonique de l'anneau des matrice vers celui des matrice de polynômes.

Après ces définitions, le théorème de Cayley-Hamilton est prouvé formellement de la façon suivante :

**Theorem** `Cayley_Hamilton` : forall A, (Zpoly (char\_poly A)).[A] = 0.

**Proof.**

`move=> A; apply/eqP; apply/factor_theorem.`

`rewrite -phi_Zpoly -mulmx_adj1 phi_mul; move: (phi _) => q; exists q.`

`by rewrite phi_add phi_opp phi_Zpoly phi_polyC ZpolyX.`

**Qed.**

La preuve se déroule exactement comme décrit dans la seconde section. Après avoir appliqué le théorème du reste, le polynôme facteur est donné en récrivant avec la règle de Cramer. Le résultat du théorème de Cayley-Hamilton est alors obtenu par des réécritures et simplifications dans le terme obtenu. L'utilisation de `SSREFLECT` a permis d'aboutir à une preuve aussi concise : 3 lignes de codes. C'est l'un des avantages du langage de tactique de `SSREFLECT` par rapport à celui de `COQ` standard.

## 5. Conclusion

Nous avons présenté une formalisation du théorème de Cayley-Hamilton qui adopte une approche modulaire. La preuve que nous avons présenté dans la section 4.4 peut paraître très simple ; mais la conception a été assez longue que ce soit pour choisir l'architecture globale de la preuve ou le bon type de données pour représenter les structures manipulées (polynômes et matrices). Ceci a été d'autant plus difficile car dans la preuve les structures utilisées vont être combinées les unes avec les autres. Les choix ont été motivés par des soucis de lisibilité et de réutilisabilité. L'utilisation des Canonical Structure de `COQ` nous a permis d'avoir des énoncés proches de ceux utilisés en mathématiques classiques. Le découpage des différentes composantes de la preuve sous forme modulaire ( les opérations indexées, les matrices et les polynômes) favorise la réutilisation de ces bibliothèques dans des développements indépendants. Les bibliothèques sur les opérations indexées et les matrices seront réutilisées dans nos prochains travaux sur la théories des caractères, une des composantes de la preuves du théorème de Feit-Thompson.

Ce travail que nous avons présenté ici est la première formalisation du théorème de Cayley-Hamilton. Ce n'est pas la première formalisation des matrices ou des polynômes. Des formalisations de ces

structures sont présentées respectivement dans [9, 11] et [5, 10]. Mais c'est le premier développement qui regroupe une formalisation des matrices et polynômes et où ces objets sont assemblés pour former de nouveaux objets : les matrices de polynômes et les polynômes de matrices. Dans ce développement la bibliothèque sur les polynômes comprend 83 objets (définitions et lemmes) pour environ 490 lignes de codes. Les définitions et lemmes propres à la preuve du théorème de Cayley-Hamilton sont au nombre de 15 pour 125 lignes de codes. Les sources du développement sont disponibles à l'adresse suivante : <http://www-sop.inria.fr/marelle/Sidi.Biha/cayley/>.

Dans la formalisation du théorème de Cayley-Hamilton, présentée dans cet article, nous avons choisi de construire nos structures de données sur des types munis d'une égalité décidables : les `eqType`. En plus du fait qu'en mathématiques classiques tous les types sont décidables, notre preuve sur les types décidables peut être généralisés vers les types quelconques. Ceci se fait en remarquant que tout anneau est une  $\mathbf{Z}$ -algèbre et en considérant le morphisme d'évaluation des polynômes à  $n^2$  variables et à coefficients dans  $\mathbf{Z}$  qui est un type décidable. Puisque les opérations algébriques sur les matrices et les polynômes n'agissent qu'en fonction des indices des coefficients, ce morphisme commute avec ces opérations. Par conséquent le théorème de Cayley-Hamilton sur les types décidables peut être généralisé à des types quelconques. Cette démarche n'est pas difficile mais elle est fastidieuse et d'aucune utilité pour le travail que nous faisons dans le cadre du projet "Mathematical Components". L'avantage de construire nos structures de données (anneaux, matrices et polynômes) sur des types décidables est d'avoir une égalité de Leibniz sur ces structures. Nous pourrions alors profiter de la puissance de la réécriture avec cette égalité, qui est la règle de réécriture par défaut dans Coq.

## Références

- [1] Paul et Jack ABAD, *The Hundred Greatest Theorems*, Disponible à <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 paraître.
- [3] Yves BERTOT, Pierre CASTÉRAN, *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*, Disponible à <http://research.microsoft.com/~gonthier/4colproof.pdf>.
- [7] Georges GONTHIER, Assia MAHBOUBI, *A small scale reflection extension for the Coq system*, Disponible à <http://www.msr-inria.inria.fr/~assia/rech-eng.html>.
- [8] Georges GONTHIER, Assia MAHBOUBI, Laurence RIDEAU, Enrico TASSI et Laurent THÉRY, *A Modular Formalisation of Finite Group Theory*, Rapport de Recherche 6156, INRIA, 2007.
- [9] Nicolas MAGAUD, *Ring properties for square matrices* contribution à Coq, <http://coq.inria.fr/contribs-eng.html>.
- [10] Piotr RUDNICKI, *Little Bezout Theorem (Factor Theorem)*, Journal of Formalized Mathematics volume 15, 2003, Disponible à <http://mizar.org/JFM/Vol15/uproots.html>.
- [11] Jasper STEIN, *Linear Algebra* contribution à 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/>.