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

Sidi Ould Biha [1]

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 constituée 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 de nature différente (algèbre linéaire, multilinéaire, combinatoire, ..). Ces objets ne sont pas uniquement utilisées, de façon indépendante ; mais au contraire ils s'emboîtent les uns avec 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'organiser la preuve en 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 brièvement 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 ${}^t\mathrm{com}B$ la transposée de la co-matrice de B, la règle de Cramer est :

$$B * {}^t\mathrm{com}B = {}^t\mathrm{com}B * B = \det B * I_n \tag{1}$$

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

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

L'anneau $M_n(R[x])$ des matrices de polynômes est aussi celui des polynômes à coefficients matriciels $(M_n(R))[X]$. L'égalité (2) s'écrit ainsi dans $(M_n(R))[X]$:

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

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 tache la plus complexe dans ce travail de formalisation. En second lieu, il faut avoir des énoncés compréhensible pour un mathématicien. Ils doivent en fait être le plus proche possible de ceux utilisés dans la littérature 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 :

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

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

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

Les propriétés de morphisme de ϕ sont alors utilisées pour obtenir :

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

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

3. SSReflect

SSReflect [7, 8] (pour *Small Scale Reflection* ou réflection à petite échelle) est une extension de Coq [3] qui introduit de nouvelles tactiques et des bibliothèques Coq adaptées pour travailler sur des types avec une égalité décidable et équivalente à l'égalité structurelle de Coq, dite égalité de Leibniz. 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-Frobenuis.
Nous allons présenter en premier lieu la méthode de SSReflect pour réfléchir entre les prédicats booléens dans la logique de Coq. Nous introduisons par la suite le langage de tactique de SSReflect. Enfin, nous présentons 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].

Réflection

Dans le système de preuve Coq la logique par défaut est intuitionniste. Dans cette logique, les propositions logiques et les valeurs booléennes sont distinctes. SSReflect permet de combiner le meilleur des deux visions et passer de la version propositionnelle d'un prédicat décidable vers la version booléenne. La version propositionnelle permet d'avoir des preuves structurées alors que celle booléenne permet de faire des calculs. 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:bool, reflect (a /\ b) (a && b).
```

Des lemmes de même nature que `andP` sont définis lorsque nous voulons avoir l'équivalence entre la représentation calculatoire d'une fonction (pouvant être calculer) donnée et sa représentation logique.

Langage de tactiques

Les scripts de preuve écrits avec SSReflect diffèrent de ceux écrits dans Coq standard. Le langage de tactique de SSReflect permet de faciliter les opérations d'interprétation et le développement des scripts. En pratique, les scripts de preuve écrits avec SSReflect se révèlent plus concis que ceux écrits dans Coq standard.

Toutes les opérations fréquentes qui consistent à déplacer ou généraliser depuis ou vers le contexte courant des formules sont regroupées dans la tactique `move`. Par exemple la tactique "`move: (H1 a)`" permet de placer dans le but courant une instance de l'hypothèse `H1` pour la variable `a`. Un autre exemple est la tactique "`move=> x y H2`" qui correspond à l'introduction des variables `x` et `y`, et d'une nouvelle hypothèse `H2` dans le contexte courant. La tactique `move: (H1 a) => H2 x y` correspond à la combinaison des deux exemples précédents dans une seule et unique tactique.
La tactique `rewrite` permet de combiner toutes les opérations de réécriture conditionnelle, de dépliage de définition, de simplification et de réécriture pour une occurrence ou un pattern donné. Ces opérations peuvent être utilisées ensemble ou séparément. Par exemple la tactique `rewrite /def H1 ?H2 !H3` permet de déplier la définition `def`, de récrire avec l'hypothèse `H1`, de récrire zéro ou plusieurs fois avec l'hypothèse `H2`, et de récrire au moins une fois avec l'hypothèse `H3`. Un autre exemple est celui de la tactique `rewrite {2}[_ * _]H4 //` qui permet de récrire dans la seconde occurrence du pattern `[_ * _]` avec l'hypothèse `H4` et de simplifier le but courant.
Le mécanisme de réflection entre les propositions décidables et les booléens décrit plus haut est intégrée au nouveau langage de tactique. Par exemple, étant donné un contexte avec une proposition `H` de type `a && b`, la tactique `move/andP : H => H` applique le lemme `andP` à `H` et introduit dans le contexte une hypothèse `H` de type `a /\ b`. En revanche, lorsque le but est de la forme `a && b`, la tactique `apply/andP` change le but par `a /\ b`. Enfin, lorsque le but est de la forme `(a && b) -> G`, la tactique `case/andP => H1 H2` change le but en `G` et introduit deux hypothèses `H1 : a` et `H2 : b`.

Structures

Des bibliothèques de base sont définies dans SSReflect. C'est une hiérarchie de structure pour travailler avec les théories décidables et en particulier les types finis. La structure `eqType` définit les types munis d'une égalité décidable et équivalente à celle 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 une forme d'héritage. La structure `eqType` ne suppose pas seulement l'existence d'une égalité décidable `==`, en plus elle injecte cette égalité vers celle de Leibniz avec le proposition `eqP`. Nous pouvons ainsi profiter de la puissance de réécriture de Coq.
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'.
```

Un ensemble sur une structure d'`eqType` est représenté par sa fonction caractéristique :

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

Une propriété booléenne sur un `eqType` correspond donc à l'ensemble des éléments qui l'a satisfont. Avec cette définition les opérations ensemblistes comme l'intersection ou le complémentaire se définissent avec les fonctions booléennes correspondantes.

```
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.
```

Il est nécessaire d'avoir un type des listes sur un `eqType d` pour pouvoir définir plus naturellement des opérations qui se base sur un test booléen comme la recherche d'un élément dans une liste. 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 définit par récurrence sur une liste. 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 liste `s`, si `i` est strictement inférieure à la longueur de la liste, et `x0` dans le cas contraire. La fonction `mkseq` permet de construire une liste de longueur donnée à partir d'une fonction sur les entiers. Par exemple, `mkseq f n` correspond à la liste `[(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 l)` 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 le types finis des élément de l'intervalle $0..n-1$, 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 booléen, 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 "Composants Mathématiques". 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, et d'avoir des énoncés plus lisibles. 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 liste `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 :

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

Les p_i sont les éléments de la liste 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 `(R, op, nil)` est un monoïde.
La notation `\big[*%M/1]_(i| P i) F i` correspond à l'application de `reducebig` à une opération `*` monoïde qui a pour élément neutre `1`. Le reste des paramètres sont inférés de façon implicite par Coq. Par exemple `\big[*%M/1]_(i| i < n) i` equivaut à la notation mathématique $\sum_{i<n} i$.

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, respectivement $[0..n]$ et $[m..n+m]$, et qu'il existe une bijection entre ces deux types. 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,
  ibijective 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 `a`, une partition de d'un ensemble `P` est donnée par les deux ensembles $\mathtt{P} \cap \mathtt{a}$ et $\mathtt{P} \cap \bar{\mathtt{a}}$. 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 enregistrement (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).
```

Le lemme suivant montre un exemple simple d'utilisation des `Canonical Structure`.

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

Rappelons que `==` dénote l'égalité dans un `eqType`. Dans cet énoncé Coq s'attend à ce que `m` et `n` soient d'un type `eqType`. Hors ils sont de type `nat`. Il cherche alors une définition d'un `eqType` dont le paramètre `sort` est `nat`. Grâce à la définition de `Canonical Structure nat_eqType`, Coq peut inférer automatiquement au arguments `m` et `n` le type `nat_eqType`.

Le mécanisme des `Canonical Structure` est puissant et très utile dans le travail avec les structures algébriques 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 la structure d'anneau sur les polynômes et les matrices. Ceci nous permet d'avoir des énoncés proches de ceux utilisés 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 liste 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 : $[0..n[\to [0..m[\to R$. Pour définir un `eqType` sur les matrices, qui sont des fonctions, nous avons besoin de l'extensionnalité. La fonction `fgraphType` construit le graphe des fonctions dont le domaine est un `finType` et le co-domaine un `eqType`. Avec la définition des graphes de fonctions, les fonctions sont ainsi munies d'une égalité de Leibniz. Pour deux fonctions `f` et `g`, les notations `f =1 g` et `f =2 g` correspondent respectivement à $\forall \texttt{x}, \texttt{f x} = \texttt{g x}$ et $\forall \texttt{x y}, \texttt{f x y} = \texttt{g x y}$ et sont équivalente à `f = g`.

Le type des matrices de taille (m, n) est défini par :

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

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 x` correspondent respectivement au type des matrices carrées et à la matrice scalaire en `x`. la notation `\matrix_(i,j) E` où `E` est une expression en `i` et `j` correspond à l'application de `matrix_of_fun` à la fonction `f i j => E i j`. Par exemple la matrice scalaire correspondant à un élément `x` est donné par la formule suivante :

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

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 est bien adapté et que nous disposons des ingrédients nécessaires à sa formalisation. En effet, une formalisation des permutations (groupe, signature ...) a été déjà développée dans le cadre du travail sur les groupes finies [8]. Etant donné une matrice carrée A de dimension n, le déterminant est donnée par :

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

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 et donc servir d'ensemble d'indice.

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 sur un ensemble à n élément. 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 dernier prend en entrée un nombre i inférieur à m (un élément de type `I_(m)`) et une matrice de taille (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 est alors formellement représentée par le lemme `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).
```

La preuve utilise la formule de Laplace ($\det(A) = \sum_{i=1}^{n} a_{i,j}\text{co} - \text{matrice}_{i,j}$) sur les co-matrices. Celle-ci est formellement donnée par :

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

Le lemme selon lequel le déterminant est une forme alternée (le déterminant d'une matrice, où au moins deux lignes sont identiques, est nul) s'énonce formelement comme suit :

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

Rappelons que les matrices sont des fonctions à deux arguments. Le terme `A i1` est une fonction à un argument et il correspond à la ligne d'indice `i1` 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
}.
```

La propriétés normal dit que le dernier élément de la liste des coefficients est non nul. 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 :

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

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 `\X` et `\C` ccorrespondent respectivement au monôme x et au polynôme constant c.
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 `\poly` permet de construire un polynôme à partir d'une fonction de coefficients. Par exemple, le polynôme correspondant à `\poly_ ( i < n ) i` est le suivant :

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

Les opérations de base sur les polynômes sont définies par récurrence sur la liste des coefficients. La liste 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 :

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

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

Rappelons que dans la définition de la structure `polynomial` nous avons une coercion entre elle et le type de la liste des coefficients. Ceci nous permet de définir l'évaluation d'un polynôme de la manière suivante :

```
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,
   x * q.[x] = q.[x] * x ->
  (p * q).[x] = p.[x] * q.[x].
```

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

```
Theorem factor_theorem : forall p c,
  reflect (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 argument une matrice de polynômes et retourne un polynôme de matrices. La longueur de la liste 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 liste de ces coefficients, en d'autre terme son degré plus un. Pour une matrice de polynômes `A`, `phi A` est le polynômes de matrices dont le coefficient d'indice `k` est la matrice dont le coefficient en `i` et `j` est `coef (A i j) k`. Dans la suite, les notations `R[X]`, `M(R)`, `M(R[X])` et `M(R)[X]` représentent respectivement l'anneau des polynômes, celui 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 `coef_phi` permet d'exprimer la relation entre une matrice de polynômes et son image 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.

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

Le polynôme caractéristique d'une matrice est définit 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_adjl 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, des mécanismes des `Canonical Structure` et des notations, ainsi que la définition hiérarchique des structures de donnée ont permit d'aboutir à une preuve aussi concise : 3 lignes de codes.

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). 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 usuelles. 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 assemblées pour former de nouveaux objets : les matrices de polynômes et les polynômes de matrices.

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 où l'égalité est décidables peut être généralisés vers les types quelconques. Ceci se fait en remarquant que tout anneau est une **Z**-algèbre et en considérant le morphisme d'évaluation des polynômes à n^2 variables et à coefficients dans **Z** qui est un type où l'égalité est décidable. Il va falloir alors travailler avec les `Setoid`.

Dans ce développement la bibliothèque sur les polynômes comprend 83 objets (définitions et lemmes) pour environs 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/`.

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/.