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 [5] ou celle du théorème des nombres premiers [1], 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 de structures mathématiques et 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.

Le théorème de Cayley-Hamilton, est l'un des théorèmes présents dans la liste des 100 théorèmes à formaliser [13]. Ce papier en présente une première formalisation. Le fait qu'il n'avait pas été jusqu'à ce jour formalisé peut s'expliquer par le fait qu'il fait intervenir de nombreuses structures et propriétés mathématiques. Ces structures ne sont pas uniquement utilisées, de façon indépendante ; mais elles sont aussi emboîtées les unes sur les autres. L'idée principale de notre travail est l'adoption d'une approche modulaire pour la formalisation des composantes utilisées dans la preuve, en particulier les polynômes et les matrices. Cette philosophie rentre dans le cadre de celle du projet "Mathematical Components" [8] dont l'un des objectifs est la formalisation du théorème de Feit-Thompson sur les groupes solvables. Dans ce cadre, le travail présenté dans cet article sera le point de départ du travail de formalisation de la théorie des caractères, une des composantes de la preuve du théorème de Feit-Thompson.

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

L'énoncé du théorème de Cayley-Hamilton [3] est le suivant :

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. Le polynôme caractéristique de A, défini par : $p_A(x) = \det(xI_n - A)$, s'annule en A ; donc $p_A(A) = O_n$.

La preuve du théorème de Cayley-Hamilton [3] découle de la formule de Cramer selon laquelle la multiplication d'une matrice B par la transposée de sa co-matrice est égale au déterminant de cette même matrice B multiplié par la matrice identité :

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

En appliquant la formule (1) à la matrice $(xI_n - A)$ on obtient que :

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

Dans la formule (2), la partie droite de l'égalité est égale au produit du polynôme caractéristique de A par la matrice identité. La formule devient donc :

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

Il est tentant de remplacer le x par A dans la formule (3). Dans la partie droite de l'égalité, nous obtiendrions $p_A(A) * I_n$, qui n'est autre que $p_A(A)$. Dans la partie gauche, la formule s'écrirait alors ${}^t\mathrm{com}(AI_n - A) * (AI_n - A)$, qui n'est autre que ${}^t\mathrm{com}(AI_n - A) * O_n$ (O_n est la matrice nulle). En conclusion, nous obtiendrions que $p_A(A) = O_n$. En réalité, cette substitution n'est pas légale car dans (3) le x est un scalaire et ne peut pas être substitué par A, qui est une matrice.
En fait, implicitement, la preuve classique du théorème utilise l'isomorphisme entre l'anneau des polynômes de matrices et celui des matrices de polynômes. 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 :

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

Soit ϕ cet isomorphisme :

$$\phi : M_n(R[x]) \to (M_n(R))[x]$$

Il est clair que l'image par ϕ de la matrice de polynôme $(xI_n - A)$ est le polynôme de matrice $(X - A)$. On utilise le X pour différencier les polynômes de matrices de ceux sur l'anneau de base R. L'image par ϕ de la matrice de polynômes $p_A(x)I_n$ est le polynôme de matrices suivant :

$$p_A(X) = a_n I_n X^n + a_{n-1} I_n X^{n-1} + \cdots + a_1 I_n X + a_0 I_n.$$

Dans la formule (3) les deux termes de l'égalité sont des matrices de polynômes. L'application de ϕ à ces termes nous donne l'égalité suivante :

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

Les propriétés de morphisme de ϕ nous donnent alors que :

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

Par construction de ϕ la formule ci-dessus s'écrit alors :

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

Le polynôme $p_A(X)$ se factorise en $(X - A)$. Par le théorème du reste, nous obtenons que $p_A(A) = O_n$.

Une formalisation de cette preuve du théorème de Cayley-Hamilton consiste à vérifier tous ses détails. Ce travail de formalisation n'est malheureusement pas aussi facile à écrire que l'énoncé du théorème ou sa preuve papier. En effet, et de manière générale, dans les preuves papiers plusieurs étapes sont implicites, ce qui ne facilite pas le travail de formalisation de telles preuves.

L'énoncé du théorème de Cayley-Hamilton donne une relation entre les polynômes et les matrices, deux objets mathématiques différents. Pour pouvoir formaliser l'énoncé du théorème, il va falloir tout d'abord définir ces deux structures. Il est aussi question de polynôme caractéristique. La définition du polynôme caractéristique fait appel à celle de déterminant. En effet, le polynôme caractéristique est le déterminant d'une matrice de polynômes. La définition du déterminant nécessite à son tour celle des sommes et produits indexés, ainsi que celle des groupes de permutations.
Le point départ de la preuve est la règle de Cramer. La formalisation de cette dernière nécessite la définitions des notions de co-matrice, de transposé et de déterminant, et la preuve de propriétés de multi-linéarités et de morphisme du déterminant. Dans la preuve, l'isomorphisme entre l'anneau des polynômes de matrices et celui des matrices de polynômes est présent de façon implicite. Lors du passage à la preuve formelle, cet isomorphisme devra être formalisé et ses propriétés de morphisme devront être prouvées. Les propriétés de morphisme de l'évaluation des polynômes sont utilisées par le théorème du reste. Il est aussi intéressant de noter que si dans l'énoncé il est question d'évaluation de polynôme de matrices, l'anneau des matrices n'est pas commutatif. Ainsi dans les propriétés de morphisme de l'évaluation, il a été nécessaire de considérer des hypothèses de commutativité entre les coefficients du polynôme évalué et la matrice que l'on évalue.

En résumé, pour arriver donc à une formalisation et à la preuve du théorème de Cayley-Hamilton, il a fallu développer une librairie sur les matrices (déterminant, formule de cramer, ...) et une autre sur les polynômes (morphisme d'évaluation, théorème du reste, ...). La formalisation de l'isomorphisme entre les matrices de polynômes et les polynômes de matrices sera l'étape ultime pour arriver à la preuve du théorème.

3. SSReflect

SSReflect [6, 7] (pour *Small Scale Reflection* ou réflection à petite échelle) est une extension de Coq [2] qui introduit de nouvelles tactiques et des librairies 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 [7] 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 avoir la réflection entre les prédicats décidables et les booléens. Nous allons présenter par la suite le langage de tactiques introduit par SSReflect et finalement nous présenterons les structures spécifiques de la librairie SSReflect et que nous avons utilisées dans notre développement.

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. 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` permet d'avoir une équivalence 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.

Langage de tactiques

Les scripts de preuve écrits avec SSReflect diffèrent de ceux écrits dans Coq standard. SSReflect ajoute une nouvelle couche au langage de tactique de Coq. 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 intégré 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`.

Librairies

Des librairies 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.

```
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, toute 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'.
```

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

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

Avec cette définition, les opérations ensemblistes comme l'intersection ou l'union se définissent avec les fonctions booléennes correspondantes.

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 décidabilité de l'égalité sur `d` permet de définir les opérations d'appartenance et de recherche d'occurrence dans une liste. L'opération d'appartenance à une liste va permettre la définition d'une coercion de liste vers `set`. Pour une liste `l`, `(l x)` peut être représentée par $\texttt{x} \in \texttt{l}$.
Une fonction utile sur les séquences est la fonction `filter`. Elle prend en paramètre un ensemble `a` et une séquence `s` et ne retourne que les éléments de `s` qui appartiennent à `a`. 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 `foldr` correspond à l'opération fold utilisée en programmation fonctionelle. Elle est définie par récurrence sur une séquence :

```
Variables (d : eqType) (R : Type) (f : d -> R -> R) (z0 : R).
Fixpoint foldr (s : seq d) : R :=
  match s with | Seq0 => z0 | Adds x s' => f x (foldr s') end.
```

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 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`.

Dans le système Coq, l'axiome d'extensionnalité n'est pas admis par défaut. En effet lorsque deux fonctions `f` et `g` sont égales sur tous les éléments de leurs domaines ($\forall$`x, f x = g x` ce qui correspond à la notation `f =1 g`), ceci n'implique pas que `f = g`. Lorsque les fonction sont à deux arguments la notation `f =2 g` correspond à la proposition ($\forall$`x y, f x y = g x y`). Dans le cas où les domaines des fonctions manipulées sont finies et les co-domaines sont des eqType, elles peuvent être représentées par leur graphe et ainsi nous obtenons une égalité de Leibniz sur ces fonctions sans avoir besoin d'ajouter d'axiomes. Etant donnés un type fini `d1` et un type `d2` muni d'une égalité décidable, un graphe est défini par :

```
Inductive fgraphType : Type :=
  Fgraph (val: seq d2) (fgraph_sizeP: size val = card d1): fgraphType.
```

Le type contient une liste `val` d'éléments de `d2` et la preuve que la taille de cette liste est égale au cardinal de `d1`. Une structure de eqType et une autre de finType sont définies sur les graphes de fonction. Elles sont respectivement représentées par `fgraph_eqType` et `fgraph_finType`. La fonction `fgraph_of_fun` permet de calculer le graphe d'une fonction donnée alors que la fonction `fun_of_fgraph` permet de voire le graphe comme une fonction. Grâce à ces définitions, l'extensionnalité fonctionnelle peut être prouvée :

```
Lemma fgraphP : forall (f g : fgraphType d1 d2), f =1 g <=> f = g.
```

Il est à noter que dans la partie gauche de l'équivalence, `f =1 g` correspond en réalité à `(fun_of_fgraph f) =1 (fun_of_fgraph g)`. En effet `fun_of_fgraph` est une coercion entre `fgraphType` et le type des fonctions.

4. Formalisations Coq

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 librairie pour les opérations indexées n'est pas seulement utile dans le développement sur la théorie des matrices, elle pourra l'être aussi dans les développements sur l'algèbre linéaire.

Une opération indexée est l'application d'une opération binaire aux éléments d'une suite indexée par un ensemble fini. C'est une généralisation de la définition d'une opération binaire en une opération n-aire. Dans le cas particulier de l'addition, c'est la somme de tous les éléments d'une suite donnée.
La fonction qui généralise la définition d'une opération binaire `op` sur un type `R` en une opération n-aire est la suivante :

```
Definition iop (d : finType)(a : set d)(f : d->R) :=
  foldr (fun x => op (f x)) nil (filter a (enum d)).
```

Pour utiliser cette fonction, on fixe d'abord l'opération binaire `op` et la valeur `nil` qui sera associée aux ensembles vides. Dans ce contexte `iop d a f` représente :

$$f\ a_1\ op\ f\ a_2\ op\ \cdots\ op\ f\ a_n\ op\ nil,$$

lorsque les a_i sont tous les éléments de l'ensemble a. L'utilisation de `iop` est plus naturelle lorsque l'opération est associative et commutative et lorsque `nil` est l'élément neutre de cette opération.

Donnons quelques lemmes intéressants sur `iop`. Le premier permet de changer le type des indices de l'opération. 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 locale entre ces types. Cette bijection est la fonction $f : x \leftarrow x + m$, qui est localement bijective sur l'interval $[0..n]$.
Le prédicat `ibjective r h` permet de dire que la fonction `h` de type `d' -> d` est localement bijective sur l'ensemble `r`. Le lemme de ré-indexation s'énonce alors comme suit :

```
Lemma reindex_iop :
  forall (d d' : finType) (h : d' -> d) r (f :d->R),
   ibijective r h ->
    iop d r f = iop d' (fun i => r (h i)) (fun i=> f (h i)).
```

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 iopID :
  forall (d : finType)(c r : set d) f,
   iop d r f =
   op (iop d (fun i => r i && c i) f)
    (iop d (fun i => r i && ~~ c i) f).
```

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

Bien sûr, ces deux lemmes ne sont prouvés que dans le cas où l'opération `op` est commutative, associative et la valeur `nil` est élément neutre.

4.2. Structures canoniques

Le théorème de Cayley-Hamilton parle de polynômes de matrices et de matrices sur des anneaux commutatifs. Les structures algébriques utilisées dans la preuve sont toutes des structures d'anneaux. Les anneaux sont définis par une structure qui regroupe les propriétés standards d'anneau.

```
Structure ring : Type := Ring {
  element :> eqType;
  0 : element;
```

```
  1 : element;
  - _ : element -> element;
  _ + _ : element -> element -> element;
  _ * _ : element -> element -> element;
  zeroP : forall x, 0 + x = x;
  oppP : forall x, - x + x = 0;
  plusA : forall x y z, x + (y + z) = (x + y) + z;
  plusC : forall x y, x + y = y + x;
  onePl : forall x, 1 * x = x;
  onePr : forall x, x * 1 = x;
  multA : forall x y z, x * (y * z) = (x * y) * z;
  plus_mult_l: forall x y z, x * (y + z) = (x * y) + (x * z);
  plus_mult_r: forall x y z, (x + y) * z = (x * z) + (y * z);
  one_diff_zero_ : 1 <> 0 (* exclure l'anneau trivial *)
}.
```

Il est à remarquer que dans la structure ring ne contient pas la propriété de commutativité de la multiplication de l'anneau. Si l'anneau des matrices n'est pas commutatif, celui des polynômes peut être commutatif si l'anneau de ses coefficients l'est aussi. Dans la preuve, d'une part la règle de Cramer sur les matrices n'est vraie que si l'anneau des coefficients de la matrice est commutatif. Pour appliquer Cramer sur l'anneau des matrices de polynômes, il faut que ce dernier soit commutatif, ce qui nécessite que l'anneau des coefficients soit à son tour commutatif. D'autre part, les polynômes de matrices sont construits sur l'anneau des matrices, qui n'est pas commutatif. L'anneau des polynômes peut se définir sur une structure d'anneau commutatif ou non commutatif.
La structure d'anneau commutatif est définie au dessus de celle d'anneau en lui ajoutant la propriété de commutativité de son opération de multiplication.

```
Structure commutative_ring : Type := CommutativeRing {
  element :> ring;
  multC :  forall x y, x * y = y * x
}.
```

Le système de coercion de Coq permet à la structure d'anneau commutatif d'hériter de celle d'anneau simple tous les lemme standards.

Grâce à ces définitions, les structures d'anneau de polynômes et de matrices se construisent sur celle d'anneau. Les constructeurs de ces structures prennent en paramètre un anneau et retournent l'anneau correspondant. Ainsi, pour définir les polynômes de matrices il suffira de passer comme paramètre au constructeur de la structure de polynômes l'anneau des matrices. La même méthode s'applique dans le sens inverse pour définir les matrices de polynômes.
Un autre avantage est d'avoir la même notation pour les opérations d'anneau (addition, multiplication et opposé) sur les polynômes et les matrices. Ceci rend l'énoncé de nos théorèmes très proches des notations utilisées en mathématiques classiques où il n'y a pas de différence entre l'addition de polynômes ou celle de matrices. Ceci se fait à l'aide du mécanisme de `Canonical Structure` [12] de Coq. Un exemple d'utilisation de ce mécanisme dans l'exemple des matrices est présenté dans la prochaine section.

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 : $[0..n[\to [0..m[\to R$. Le type des matrices de taille (m, n) est défini par :

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

Dans cette définition, les matrices sont des fonctions à deux arguments. Nous avons donc défini les fonctions `matrix_of_fun` et `fun_of_matrix` qui 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.
Dans la suite, les notations `M_(n)`, `+m`, `*m`, `*sm`, `\0m` et `\1m` correspondent respectivement au type des matrices carrées, à l'addition de deux matrices, la multiplication de deux matrices, la multiplication d'une matrice par un scalaire, la matrice nulle et la matrice unité.

Après avoir défini le type des matrices et pour construire la structure d'anneau sur le type des matrices carrées, il va falloir prouver les axiomes d'anneaux sur ce type.

```
Variable (n : nat) (Hn : 0<n).
Lemma mx_plus0x: forall (A : M_(n)), \0m +m A = A.
Lemma mx_scale_oppl: forall (A : M_(n)), (- 1 *sm A) +m A = \0m.
Lemma mx_plusA: forall (A B C : M_(n)), A +m (B +m C) = (A +m B) +m C.
Lemma mx_plusC: forall (A B : M_(n)), A +m B = B +m A.
Lemma mx_mult1x: forall (A : M_(n)), \1m *m A = A.
Lemma mx_multx1: forall (A : M_(n)), A *m \1m = A.
Lemma mx_multA: forall (A B C : M_(n)), A *m (B *m C) = (A *m B) *m C.
Lemma mx_distrL: forall (A B C : M_(n)), A *m (B +m C) = (A *m B) +m (A *m C).
Lemma mx_distrR: forall (A B C : M_(n)), (A +m B) *m C = (A *m C) +m (B *m C).
Lemma mx_0_diff_1: \0m <> \1m.
Canonical Structure matrix_ring: ring:=
  Ring mx_plus0x m_scale_oppl mx_plusA mx_plusC mx_mult1x mx_multx1
       mx_multA mx_distrL mx_distrR mx_0_diff_1.
```

L'utilisation de `Canonical Structure` dans la dernière déclaration permettra à Coq d'inférer automatiquement la structure d'anneau de matrice lorsque nécessaire. C'est par exemple le cas dans le lemme suivant qui dit que la trace de la somme de deux matrice est la somme des traces des deux matrices.

```
Lemma trace_plus_mx : forall n (A B : M_(n)), \tr (A + B) = \tr A + \tr B.
```

Dans l'énoncé du lemme les matrices `A` et `B` ne sont pas déclarées de type `matrix_ring` mais simplement de type `matrix`, Coq a inféré automatiquement la structure d'anneau et a accepté l'utilisation de la notation pour l'addition d'anneau dans la partie gauche de l'égalité.

Pour définir les déterminants, nous avons utilisé la formule de Leibniz où nous supposons que A est une matrice carrée de dimension n.

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

Dans cette formule, nous utilisons des opérations indexées pour les sommes et les produits, mais les notations mathématiques cachent plusieurs autres éléments qui doivent être formalisés précisément :

- 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 doit également être décrit comme un ensemble fini qui pourra être énuméré,
- pour toute permutation il est nécessaire de calculer sa parité.

Pour représenter les permutations, la bibliothèque SSReflect utilise les `fgraphType`. Le type permutation est facilement décrit comme un type fini qui peut être énuméré et sur lequel on peut faire des opérations indexées finies. Pour calculer la parité d'une permutation σ, on veut compter le nombre d'inversions de la permutation, c'est à dire l'ensemble des paires (i, j) telles que $i < j$ et $\sigma(j) < \sigma(i)$. Pour ce travail, nous utilisons une théorie des multiplets sur des types finis, un type fini et l'ordre naturel des éléments induits par l'ordre dans la séquence `enum` qui définit ce type. Cet ordre est utilisé pour comparer i et j d'une part et $\sigma(i)$ et $\sigma(j)$ d'autre part.

Grâce à ces développements la formule (5) peut alors s'écrire :

```
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 `iop` 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 montrer que le déterminant est une forme linéaire, nous avons eu besoin de montrer que la structure d'anneau sur l'ensemble des coefficients où la multiplication est associative, l'addition est commutative, et la multiplication distributive sur l'addition se retrouve sur les opérations indexées. Ces preuves se font assez simplement. Nous arrivons donc au point où nous voulons exprimer des relations entre les lignes de plusieurs matrices. Pour cela nous utilisons deux fonctions : la fonction `row` prend en entrée un nombre i inférieur à m (un élément de type `I_(m)`) et une matrice (m, n) ; elle retourne la matrice $(1, n)$ (une rangée et n colonnes) qui contient la rangée i. Avec les mêmes arguments la fonction `row'` retourne la matrice $(m - 1, n)$ où la rangée choisie a été enlevée. Grâce à ces fonctions la multilinéarité s'écrit de la façon suivante :

```
Lemma determinant_multilinear : forall n (A B C : M_(n)) i b c,
   row i A =2 b *sm row i B +m c *sm row i C ->
   row' i B =2 row' i A -> row' i C =2 row' i A ->
 \det A = b * \det B + c * \det C.
```

Nous devons également démontrer que le déterminant est une forme alternée. Pour cela il est nécessaire de montrer plusieurs propriétés sur la parité des permutations. En particulier, nous avons établi que pour la composition de toute permutation avec une transposition entre deux éléments distincts établit une bijection entre les permutations paires et les permutations impaires. Les lemmes de partitionnement et de réindexation des opérations indexées suffisent alors pour obtenir l'énoncé suivant :

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

Des calculs dans le même esprit permettent d'obtenir la propriété de morphisme du déterminant :

```
Lemma determinantM :
 forall n (A B : M_(n)), \det (A * B) = \det A * \det B.
```

Puis nous pouvons définir la co-matrice d'une matrice à l'aide de la fonction `row'` et d'une fonction similaire sur les colonnes, puis la transposée de cette co-matrice (représentée par la fonction `adjugate`) et prouver l'égalité de Cramer :

```
Lemma mult_adugateR :
 forall n (A : M_(n)), A * adjugate A = \det A *sm \1m.
```

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. 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 non totalement nulle sur un domaine fini. Nous avons donc défini la fonction coefficient des polynômes par :

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

La fonction `coef` est de type `nat -> R`. Étant donné un entier i, `coef` retourne l'élément d'indice `i` dans la séquence des coefficients du polynômes p si i est inférieur à la taille de cette séquence. Elle retourne 0 dans le cas contraire.
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`, `\C c` et `\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 librairie 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.
```

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 : poly) := 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_mult_poly`

```
Lemma coef_mult_poly : forall p1 p2 k,
  coef (mult_poly p1 p2) k =
  \sum_(i : I_(k.+1)) (coef p1 i) * coef p2 (k - i).
```

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 séquence des coefficients et se définit par récurrence sur cette séquence. 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_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).

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

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

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

La condition de commutativité s'explique donc par le fait que pour avoir (8) égale 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. L'idée de l'algorithme pour construire ce morphisme est d'écrire la matrice de polynômes sous la forme d'une somme : $M+XM'$, avec M une matrice sur l'anneau de base, M' une matrice de polynômes et X la matrice $xI_(n)$. Cette opération sera itérée autant de fois que 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 `\Mp_(n)` et `\Pm[x]` représentent respectivement l'anneau des matrices de polynômes et celui des polynômes de matrices.

```
Definition phi (M : \Mp_(n)) : \Pm[x] :=
  foldr (fun k p =>
            (horner p (matrix_of_fun (fun i j=> coef (M i j) k))))
  \C0 (iota 0 (size_mx_of_poly M)).
```

Pour définir `phi`, nous passons à `foldr` la fonction qui étant donnée une matrice de polynômes `M`, un polynôme de matrices `p` et un indice `k`, décale le polynôme `p` à droite et lui ajoute comme coefficient de plus bas degré la matrice des coefficients d'indices `k` des polynômes de la matrices `M`. La fonction

`iota a b` construit une séquence d'entiers commencant par `a` et de longueur `b` ou une séquence vide si `b` est nulle.
Par ailleurs il est aussi nécessaire d'avoir une relation entre le résultat de `phi` et sa valeur d'entrée. La propriété de certification de la fonction `phi` est la suivante :

```
Lemma phi_coef : forall (M : \Mp_(n)) i j k,
  coef (M i j) k = (coef (phi M) k) i j.
```

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`.

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 `p2pm` correspond à cette fonction. Après ces définitions, le théorème de Cayley-Hamilton s'énonce formellement de la façon suivante :

```
Theorem Cayley_Hamilton : forall A, (p2pm (poly_car A)).[A] = 0.
```

La preuve se déroule exactement comme décrit dans la seconde section. Après avoir généralisé la règle de Cramer, nous lui appliquons le morphisme `phi`. Le résultat du théorème de Cayley-Hamilton est alors obtenu par des réécritures et simplifications dans le terme obtenu.

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 indexes, les matrices et les polynômes) favorise la réutilisation de ces librairies dans des développements indépendants. Les librairies 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 [4, 10]. Mais c'est le premier développement qui regroupe une formalisation des matrices et polynômes et où ces structures sont assemblées pour former de nouvelle structure : les matrices de polynômes et les polynômes de matrices. Dans ce développement la librairie sur les polynômes comprend 83 objets (définitions et lemmes) pour environs 490 lignes de codes. Celle pour les matrices comprend 89 objets pour 700 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 **Z**-algèbre et en considérant le morphisme d'évaluation des polynômes à n^2 variables et à coefficients dans **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 pourrons alors profiter de la puissance de la réécriture avec cette égalité, qui est la règle de réécriture par défaut dans Coq.

Références

[1] 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.

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

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

[4] 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.

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

[6] Georges Gonthier, *Notations of the four colour theorem proof*, Disponible à http ://research.microsoft.com/ gonthier/4colnotations.pdf.

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

[8] Georges Gonthier, Benjamin Werner, Yves Bertot, *Mathematical Components Manifesto*, Disponible à http ://www.msr-inria.inria.fr/projects/math/manifesto.html.

[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/.