Composants mathématiques pour la théorie des groupes

Sidi Ould Biha

UNIVERSITE de NICE-SOPHIA ANTIPOLIS

Ecole Doctorale STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H E S E

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice - Sophia Antipolis

Spécialité : **Informatique**

Présentée et soutenue par

Sidi OULD BIHA

Composants mathématiques pour la théorie des groupes

Thèse dirigée par **Laurent THÉRY**

Préparée à l'INRIA Sophia Antipolis, projet MARELLE, et le centre de recherche commun INRIA-Microsoft Research, projet Math Components

soutenue le 24/02/2010

Jury :

M. **Herman**	**Geuvers**	Professeur, Nijmegen	Examinateur
M. **André**	**Hirschowitz**	Professeur, Nice Sophia-Antipolis	Examinateur
M. **Jean-François**	**Dufourd**	Professeur, Strasbourg	Rapporteur
M. **Renaud**	**Rioboo**	Professeur, ENSIIE Evry	Rapporteur
M. **Laurent**	**Théry**	Chargé de recherche à l'INRIA	Directeur

A mes parents...

Remerciements

Je tiens à remercier ici tous ceux qui ont eu un rôle déterminant dans la réalisation de ce travail de thèse, en mettant à ma disposition leurs compétences scientifiques et leur soutien humain.

Je souhaite adresser mes plus vifs remerciements à Laurent Théry, mon directeur de thèse, pour ses conseils judicieux et sa pédagogie tout au long de ce travail. Il a su guider mon esprit parfois confus et m'a permis d'acquérir à la fois autonomie et détermination dans le travail. Son enthousiasme et sa sympathie ont été pour moi un exemple précieux.

Mes remerciements vont aussi à tous les membres du jury : à Messieurs Jean-François Dufourd et Renaud Rioboo pour s'être acquittés de la lourde tâche de rapporteurs ; à Messieurs Herman Geuvers et André Hirschowitz pour l'intérêt qu'ils ont bien voulu porter à ce travail en participant au jury. Merci André Hirschowitz pour votre cours de lambda calcul.

Merci Yves pour m'avoir donné l'opportunité de faire un stage de Master dans l'équipe Marelle. Merci surtout pour ta bonne humeur et les soirées billards.

Merci Laurence pour tes conseils et encouragements. Merci pour avoir consacré plein de temps à relire ce manuscrit et m'aider à corriger les multiples fautes d'orthographes. Merci aussi pour les *Crunch*.

Merci Ioano pour ta patience, ta bonne humeur et ta sympathie. Durant ces trois années de thèse, j'ai eu la chance de partager avec toi d'innombrables moments joyeux au bureau et en dehors. Je te remercie surtout pour avoir assurer mon approvisionnement en chocolat durant les périodes critiques de la rédaction de cette thèse.

Merci Anne, Nathalie, Benjamin, Loic, Michael, Minh, Nicolas pour les casses têtes et Sylvain pour les torrents. Thank you Tom for your patience with my bad spoken english. Gracias Jorge y George.

Merci aussi Amel, Bhatti, Adan, Alice et toute la communauté roumaine de l'INRIA.

Je tiens à remercier ici Georges Gonthier pour ses nombreux conseils et explications. Merci pour m'avoir accueilli au centre de recherche *MSR Cambridge*. Il a eu une patience infinie pour arriver à m'expliquer des points qui lui paraissaient triviaux alors que je n'en saisissais pas toute la nature. Qu'il soit assuré de ma reconnaissance et de mon profond respect.

Merci Assia pour tes conseils scientifiques et tes encouragements. Merci à Martine pour avoir su bien gérer ma situation administrative particulière.

Enfin merci et encore merci à tous ceux qui ont contribué directement ou indirectement à l'aboutissement de ce travail.

Résumé

Les systèmes de preuves formelles ont connu ces dernières années des évolutions importantes. Des travaux récents, comme la preuve formelle du théorème des quatre couleurs ou celle du théorème des nombres premiers, ont montré que ces systèmes ont atteint un niveau de maturité leur permettant de s'attaquer à des problèmes mathématiques non triviaux. Malgré cela, l'utilisation des systèmes de preuves formelles en mathématique reste très limitée. Un des arguments qui est avancé pour expliquer cette situation est le manque de bibliothèques de preuves formelles. Cette thèse s'intéresse au développement de composants mathématiques pour la théorie des groupes finis. Elle entre dans le cadre du travail de formalisation du théorème de Feit-Thompson sur la classification des groupes finis. L'objectif principal dans ce travail est d'appliquer les techniques de génie logiciel pour faciliter la réutilisation et l'organisation des développements mathématiques formelles de grande échelle, comme la formalisation du théorème de Feit-Thompson.

Cette thèse présente une première formalisation du théorème de Cayley-Hamilton sur les polynômes et les matrices. Elle présente aussi des développements sur la théorie des représentations des groupes finis qui est une composante nécessaire à la formalisation de la preuve du théorème de Feit-Thompson. En particulier, elle présente une formalisation de la théorie des modules sur un corps ou sur une algèbre ainsi qu'une formalisation du théorème de Maschke. Ces développements ont été faits dans le système Coq et avec l'extension SSReflect.

Mots clés : Formalisation des mathématiques, Coq, SSReflect, Théorie des représentations, Groupes finis, Théorème de Cayley-Hamilton.

Abstract

Formal proof systems have evolved considerably in recent years. Success stories like formal proofs of the four color theorem or the prime numbers theorem have shown that formal proof systems have reached a level of maturity that enables them to tackle non-trivial mathematical problems. Nevertheless, the use of formal proof systems in mathematics is very limited. One of the reasons that explains this situation is the lack of formal proof libraries. This thesis focuses on the development of mathematical components for finite groups theory. It is part of a project that aims to formalize the Feit-Thompson theorem on the classification of finite groups. The main goal of this work is to apply software engineering techniques to facilitate the reuse and organization of large scale formal mathematics developments like the formalization of the Feit-Thompson theorem.

This thesis presents the first formalization of the Cayley-Hamilton theorem on polynomials and matrices. It presents also developments on the representation theory of finite groups which is a necessary component for the formalization of the Feit-Thompson theorem. In particular, it presents a formalization of the theory of modules over a field or an algebra and a formalization of the Maschke theorem. These developments have been done in the Coq system and with the SSReflect extension.

Key words : Formalization of mathematics, Coq, SSReflect, Representation theory, Finite group, Cayley-Hamilton theorem.

Table des matières

Table des figures

Introduction

Les preuves sont une composante essentielle des mathématiques. Une preuve mathématique valide fournit une assurance parfaite qu'un énoncé est vrai. Tout au long de l'histoire des mathématiques les preuves mathématiques n'ont cessé d'augmenter en longueur et en complexité. La compréhension de la preuve du théorème de Pythagore ou de la résolution d'une équation quadratique est à la portée du commun des mortels. Les développements immenses qu'ont connus les mathématiques durant les deux derniers siècles ont fait que les preuves mathématiques sont devenues si complexes qu'elles ne peuvent plus être vérifiées que par des spécialistes. L'exemple le plus récent de cette tendance est la preuve du théorème de Fermat par Wiles. Rares sont les mathématiciens qui prétendent comprendre la totalité de la preuve de Wiles. La classification des groupes simples finis, aussi appelée le "théorème énorme", est un autre exemple de preuves complexes et longues. Cette preuve correspond à un ensemble de résultats publiés séparément et qui totalisent plus de 10.000 lignes. Tout au long des vingt dernières années la preuve a suscité de nombreux débats dans la communauté mathématique sur le fait qu'elle soit complète et correcte. Des mathématiciens comme J. P. Serre pense que la preuve nécessite encore des vérifications [1]. La formalisation est une méthode de vérification de l'exactitude de ces preuves longues et complexes.

La formalisation d'une preuve mathématique est la production d'une preuve formelle qui explicite toutes les étapes du raisonnement logique et ceci en partant des axiomes fondamentaux des mathématiques. Ceci est en contraste avec une preuve mathématique traditionnelle qui contient de nombreux arguments et des suppositions qui sont laissés à la charge du lecteur ou sont implicites. Les preuves formelles sont moins susceptibles de contenir des erreurs que les preuves traditionnelles. Étant donné qu'il n'est pas envisageable de faire de telles preuves à la main vu leur longueur, l'utilisation de l'ordinateur pour cette tâche s'impose. Les systèmes de preuves formelles sont des programmes informatiques qui permettent de construire ou d'aider à construire de telles

1. interview au *European Mathematical Society Newsletter* datée de Septembre 2003, pages 18-20

preuves formelles. Les origines théoriques des systèmes de preuves formelles se trouvent dans les travaux sur les fondements des mathématiques et dont Fredge peut être considéré comme le père moderne. Le système logique introduit par Fredge vers la fin du $19^{ème}$ siècle contenait un paradoxe, connu sous le nom de paradoxe de Russell. C'est pour corriger cette inconsistance du système de Fredge que la théorie des types et celle des ensembles ont été développées. Ces deux théories sont le fondement des systèmes de preuves formelles contemporains. La majorité des systèmes de preuves formelles sont une implémentation de l'une de ces deux théories.

Malgré leurs origines mathématiques, les systèmes de preuves formelles ont connu leurs plus grands succès applicatifs en informatique. Dans cette discipline ils ont su répondre au besoin croissant dans le domaine de la sureté du logiciel et du matériel. Ils ont permis ainsi d'accroitre considérablement la qualité de tels produits. En mathématiques, les récents succès comme [2, 16, 19] ont montré que les systèmes de preuves formelles ont atteint un niveau de maturité suffisant pour traiter de problèmes mathématiques avancés. Malgré cela, leur utilisation par les mathématiciens reste limitée. Parmi les raisons avancées pour expliquer cet état de fait il y a le manque de bibliothèques de théories mathématiques. En effet, et en faisant le parallèle avec les langages de programmation, la richesse et la facilité d'utilisation des bibliothèques d'un langage de programmation sont des facteurs importants de son succès. Le langage de programmation JAVA en est un exemple. Dans le contexte de l'application des systèmes de preuves formelles en mathématiques, le développement de bibliothèques de théories mathématiques réutilisables est nécessaire. De manière générale, une théorie mathématique comprend des objets, des opérations calculatoires sur ces objets et des preuves sur ces objets et opérations. Une théorie mathématique peut faire appel de manière explicite ou implicite à d'autres théories. La formalisation de telles théories demande d'adopter des techniques de génie logiciel pour assurer la généricité et la réutilisation.

Composants mathématiques

Ce travail de thèse se place dans le cadre du projet *Mathematical Components* du laboratoire commun INRIA/Microsoft Research. L'un des objectifs de ce projet est de formaliser la preuve du théorème de Feit-Thompson [13]. La formalisation de ce théorème est un vrai défi pour les systèmes de preuve formelle. Déjà à sa publication, la compréhension de la preuve a posé problème pour la communauté mathématique. La vérification de la preuve a été assurée par un groupe de spécialistes de la théorie des

groupes a demandé une année de travail intense. Malgré la multitude de travaux pour réduire les 255 pages de la preuve initiale, ce nombre n'a pas diminué. La longueur et la complexité de la preuve ne sont pas les seuls défis, la variété des théories mathématiques couvertes par la preuve en est un autre. En effet, la preuve couvre la théorie des groupes finis et celle des caractères. Ces deux théories font appel à des résultats sur l'analyse locale, l'algèbre linéaire, la théorie de Galois et la théorie des représentations. Ces théories font, elles aussi, appel à d'autres théories mathématiques et sont utilisées dans d'autres importants résultats mathématiques indépendants du théorème de Feit-Thompson. L'objectif dans le projet *Mathematical Components* n'est pas uniquement la formalisation de la preuve du théorème de Feit-Thompson, mais le développement des différentes composantes de la preuve en adoptant l'approche de la programmation par composants. Les théories mathématiques sont ainsi organisées sous forme de composants logiciels ou composants de preuves.

Dans le cadre du projet *Mathematical Components*, un composant mathématique est défini comme une bibliothèque qui modélise une théorie mathématique donnée. Cette théorie peut être explicite comme la théorie des polynômes ou la théorie des matrices : chaque ouvrage d'algèbre comprend en général des chapitres sur les polynômes et les matrices. Elle peut aussi être implicite comme les opérations indexées. Ces dernières sont utilisées par les mathématiciens sous différentes formes sans qu'une théorie ne soit explicitement donnée. Dans le contexte de l'environnement de preuve formelle Coq, un composant mathématique va être représenté par un *module*. Celui-ci est composé principalement de deux parties :

- **Statique** : Objets et Axiomes. Ce sont les points de départ de la théorie. Par exemple, un polynôme est une liste de coefficients qui respectent un certain ordre.
- **Dynamique** : Preuves et Opérations. Ce sont les règles logiques et calculatoires que l'on peut appliquer aux éléments de la théorie. Par exemple, les polynômes forment un monoïde pour leur opération d'addition.

En plus des deux parties ci-dessus, le composant mathématique doit définir des méthodes d'accès à la théorie qu'il implémente. Ces méthodes doivent fournir à l'utilisateur un moyen d'utiliser le développement de façon transparente. Elle doivent assurer la séparation entre le contenu et la présentation de ces théories. L'utilisateur ne doit pas nécessairement avoir besoin de savoir comment les objets de la théorie sont construits. Il ne doit avoir besoin que d'utiliser des structures et des constructions de base du système. L'utilisation de ces techniques de génie logiciel facilite la maintenance et l'extension du développement. Dans le cadre d'un travail de formalisation de grande

envergure, comme pour la preuve du théorème de Feit-Thompson, la maintenance et l'extension sont des paramètres importants dans le processus de développement.

Contributions et organisation

L'objectif principal de ce travail de thèse est de développer des composants mathématiques formels pour la théorie des groupes. En particulier, nous nous intéressons à la théorie des représentations des groupes finis. C'est une des théories mathématiques sur lesquelles se base la preuve du théorème de Feit-Thompson. Notre travail de formalisation de cette théorie utilise les méthodes et outils utilisés dans le développement de la preuve formelle du théorème des quatre couleurs. En se basant sur ces méthodes et outils, cette thèse apporte principalement les contributions suivantes :

- un développement de composants pour la théorie des représentations des groupes finis. En particulier, nous avons développé une théorie formelle des structures d'espaces vectoriels, d'algèbres et de modules sur une algèbre. Ces structures sont étudiées dans le cas de la dimension finie. Ceci nous permet de profiter des propriétés de décidabilité correspondante. Au dessus de ces développements, nous réutilisons un autre développement indépendant sur les groupes finis pour formaliser le théorème de Maschke qui est un résultat de base sur les représentations des groupes finis.
- La première formalisation du théorème de Cayley-Hamilton. Pour cela, nous avons développé une théorie des polynômes. Ce développement est combiné avec un autre développement indépendant sur les matrices pour arriver à une preuve formelle Coq du théorème de Cayley-Hamilton. Ce théorème possède des applications dans la preuve du théorème de Feit-Thompson notamment dans la partie algèbre linéaire de la preuve.
- Pour les besoins des formalisations citées ci-dessus, nous avons aussi participé au développement de bibliothèques formelles pour les opérations indexées et les structures munies d'un opérateur de choix. Ces développements répondent à des besoins qui se sont manifestés durant ce travail de thèse.

La suite de cette thèse est organisée comme suit. Dans le Chapitre 1 nous donnons une brève introduction à Coq. Cette introduction met l'accent sur les fonctionnalités du système qui jouent un rôle important dans notre développement. Le Chapitre 2 est une présentation de SSReflect, l'extension Coq qui est l'environnement de base pour le développement des preuves formelles dans le projet *Mathematical Components*. Nous

présentons en particulier le mécanisme de réflexion booléenne défini par SSReflect et ses bibliothèques de base sur lesquelles est construit notre développement. Dans le Chapitre 3 nous décrivons notre formalisation de la preuve du théorème de Cayley-Hamilton. Ce chapitre présente la preuve papier du théorème de Cayley-Hamilton et les bibliothèques formelles qui ont été développées pour aboutir à la formalisation de cette preuve. Le Chapitre 4 expose les différents composants développés pour la formalisation d'une théorie des représentations des groupes finis. Ceci couvre le développement d'une théorie des modules sur les corps et les algèbres ainsi qu'une formalisation du théorème de Maschke. Avant de conclure, nous présentons aussi des perspectives d'évolution de ce travail pour les représentations.

Chapitre 1

Préliminaires

Ce chapitre est une introduction au système Coq. Nous ne présentons ici que certains aspects du système qui jouent un rôle important dans notre développement. Après une brève introduction, nous présentons les mécanismes fournis par le système pour définir de nouveaux types. Par la suite, nous présentons les fonctionnalités du système qui permettent d'assouplir la rigueur imposée par le système de type de Coq. Pour faire un développement mathématique dans le langage typé de Coq, ces fonctionnalités permettent de modéliser la pratique mathématique où les informations implicites jouent un rôle important. Elles sont aussi utiles d'un point de vue génie logiciel puisqu'elles permettent de faire des développements génériques. Pour plus détails sur la programmation avec Coq, nous renvoyons au livre *Coq'Art* [6] et au tutoriel sur les types inductifs du système [17].

1.1 Coq : introduction générale

Coq est une implémentation du Calcul des Constructions Inductives (CCI) [41, 55]. Le Calcul des Constructions [12] est un lambda calcul typé [4] d'ordre supérieur. Il permet, entre autres, la définition de types dépendants et le polymorphisme. Le CCI est une extension du Calcul des Constructions pour permettre la définition des types inductifs.

Le système Coq définit un langage de programmation fonctionnelle avec des types dépendants. Le langage de spécification de Coq, appelé *Gallina*, fournit les moyens pour faire de la programmation avec le système. Coq est aussi un environnement de preuves formelles. En effet, grâce à l'isomorphisme de Curry-Howard qui fournit une correspondance entre preuve (raisonnement logique) et programme (calcul), Coq permet d'exprimer des preuves mathématiques. Coq fournit ainsi un environnement pour

programmer le raisonnement mathématique.

Les types jouent un rôle central dans Coq. Dans le système, chaque objet ou terme doit avoir un type. Comme ces types sont eux aussi des termes, ils sont eux aussi typés. Le système de type de Coq est construit sur deux *sortes* de base : `Prop` et `Type`. La sorte `Prop` est l'univers des propositions logiques. La sorte `Type` est l'univers des types des spécifications : programmes ou types de données comme les entiers ou les listes. En plus des types de base Prop et Type, Coq fournit des constructeurs de nouveaux types. Le constructeur de base pour les types est la flèche ou le produit. Par exemple, pour deux types données `T1 T2: Type`, le terme `T1 → T2` représente le type des fonctions de `T1` dans `T2`. Dans le cas d'objets dans `Prop` la flèche va représenter l'implication. Par exemple, pour deux propositions `A B : Prop`, le terme `A → B` correspond à la proposition `A ⇒ B`. Un objet du type `A → B` est une preuve de la proposition que `A` implique `B`.

En plus du constructeur de base →, le système fournit d'autres mécanismes pour définir des types plus riches. Ces mécanismes sont présentés dans la prochaine section.

1.2 Définition de types

Coq fournit un environnement pour formaliser les théories mathématiques. Cet environnement se base sur la théorie des types. Chaque objet définit dans cet environnement doit avoir un type. Pour formaliser des concepts mathématiques avancés, les types fournis par défaut avec le système ne suffisent pas. Il est alors nécessaire de définir de nouveaux types. Le système fournit des mécanismes pour définir de nouveaux types concrets ou abstraits. En particulier, Coq permet de définir des types inductifs et des types enregistrements.

Types inductifs

Les types inductifs jouent un rôle important dans le système Coq. Ils peuvent être utilisés pour définir des types de données ou des prédicats logiques. La définition d'un type inductif correspond à la donnée d'un ensemble fini de constructeurs à partir duquel on peut définir tous les éléments du type. Ces constructeurs peuvent être des constantes ou des fonctions. Pour définir un type inductif il faut utiliser le mot clé `Inductive` et ensuite fournir la liste des constructeurs. Un premier exemple de type inductif dans Coq est le type des booléens. Celui-ci est représenté par le type suivant :

```
Inductive bool : Type := true : bool | false : bool
```

Le type `bool` est un type qui contient deux éléments ou constructeurs : les constantes `true` et `false`. Ces constantes correspondent aux valeurs de vérité *vrai* et *faux*. Avec l'exemple du type `bool`, les types inductifs permettent de définir des types finis dont les constructeurs correspondent au éléments du types. Un second exemple de définition de type inductif infini est le type des entiers naturels. La définition Coq du type des entiers naturels est la suivante :

```
Inductive nat : Type := O : nat | S : nat → nat.
```

Dans la définition ci-dessus, les termes `O` et `S` représentent les constructeurs du type `nat`. Le constructeur `O` représente l'entier 0. Le constructeur `S` représente la fonction successeur qui étant donné un entier retourne son successeur. Par exemple, 1 est représenté par `(S O)`, 2 est représenté par `(S (S O))` et ainsi de suite.

Les types `bool` et `nat` définis ci-dessus sont des type inductifs simple. Coq permet aussi la définition de type inductifs dépendants. Un exemple est le type des listes.

```
Inductive list (T : Type) : Type :=
 | nil : list T | cons : T → list T → list T.
```

Le type `list` représente l'ensemble des listes dont les éléments appartiennent à un type arbitraire `T`. C'est un type inductif avec deux constructeurs. Le premier constructeur `nil` représente la liste vide. Le second constructeur `cons` représente la fonction d'ajout d'un élément à une liste.

De façon générale, le mécanisme des types inductifs de Coq permet de définir de nouveaux types finis ou infinis dépendants ou non. Le schéma général de la définition de tels types correspond à donner la liste des constructeurs d'éléments du type. Ces constructeurs sont des constantes ou objets initiaux et des fonctions qui permettent de construire récursivement de nouveaux objets du type.

Type enregistrement

Coq fournit un mécanisme qui permet de définir des types d'enregistrements. Ils sont définis comme des types inductifs avec un seul constructeur. Les différents champs de l'enregistrement vont correspondre aux arguments du constructeur. Les deux macros équivalentes `Structure` et `Record` permettent de définir de tels types.

Dans le contexte d'un développement mathématique, les types enregistrements sont très utiles. Ils permettent en particulier de définir des interfaces pour les structures algébriques [16, 42]. Généralement une structure algébrique ou combinatoire se compose d'un ensemble, des constantes et opérations dans cet ensemble, et des axiomes

```
Structure monoid : Type := Monoid {
 sort : Type;
 op : sort → sort → sort; e : sort;
 opA : forall x y z, op x (op y z) = op (op x y) z;
 opIe : forall x, op x e = x; opeI : forall x, op e x = x }.
```

FIGURE 1.1 – Définition de l'interface de monoïde.

sur ces constantes et opérations. Dans le cadre d'une théorie des types, l'ensemble va correspondre au type des éléments, les types des opérations vont dépendre de ce type et les axiomes vont dépendre de ces opérations et transitivement du type des éléments. Par exemple, une structure de monoïde est définie par :

- un ensemble E,
- une loi $+$ dans E qui est associative,
- $e \in E$ un élément neutre à droite et à gauche pour $+$.

Les types enregistrement fournissent un mécanisme pour regrouper les composants d'une structure algébrique dans une même entité, ce qui permet plus d'abstraction et de généricité. Une interface pour les monoïdes peut être définie comme dans la figure 1.1. Dans cette définition et plus généralement dans la définition d'un type enregistrement, nous avons deux classes de champs de l'enregistrement. En premier lieu, nous avons les constantes et opérations définis par la structure associée au type enregistrement. Dans l'exemple du type `monoid`, le champ `op` représente l'opération interne de la structure de monoïde. En second lieu, nous avons les propriétés logiques que doivent satisfaire les constantes et opérations de la structure. Par exemple, la proposition `opA` permet de dire que l'opération interne de la structure de monoïde est associative. En dehors de la déclaration du type enregistrement, les champs associés prendront comme premier argument une instance de l'enregistrement. Le champ `sort` de `monoid` a pour type complet `monoid → Type`. Pour l'opération interne du monoïde, le type complet correspondant est `op : forall M : monoid, sort M → sort M → sort M`.

Dans un type enregistrement, l'ensemble des champs correspond aux objets qui composent la structure. Ce sont les objets nécessaires à la définition d'une instance du type. Dans l'exemple du type `monoid`, la fonction `Monoid` permet de construire une telle instance du type. Elle prend en paramètre des objets qui correspondent aux différents champs de l'enregistrement et retourne un objet de type `monoid`. Par exemple, le monoïde additif des entiers est déclaré comme suit :

```
Definition nat_monoid := Monoid nat + 0 plusA plusn0 plus0n.
```

L'objet `nat_monoid` est de type `monoid`. C'est un nouvel objet qui modélise le monoïde additive des entiers. Il est différent du type `nat` des entiers naturels. Par contre, nous avons que le type `(sort nat_monoid)` est égal à `nat` et que la fonction `(op nat_monoid )` est de type `nat → nat → nat` et est égale à la fonction `+` d'addition des entiers.

Les types enregistrements de Coq fournissent un mécanisme semblable aux enregistrements et classes présents dans différents langages de programmation. Ils représentent une alternative au système de modules de Coq [10]. Par contre ils ne prennent pas en compte la gestion des noms des champs. En effet deux enregistrements distincts ne peuvent pas partager un même nom de champ. Une solution partielle à cette limitation est de définir les types enregistrements dans des modules séparés.

1.3 Assouplissement de la discipline de type

La théorie des types de Coq fournit un environnement logique pour faire des preuves formelles. Cette théorie introduit une discipline d'utilisation des types dans le système. Ainsi chaque objet définit dans le système doit avoir un seul et unique type. Par défaut dans le système les preuves et énoncés logiques doivent être énoncés explicitement. Coq définit de nouveaux mécanismes pour introduire plus de souplesse dans son système de types. Ces mécanismes permettent de déléguer au système des taches d'inférence d'information de types. Ils jouent un rôle important dans le contexte d'un développement mathématique. En particulier, ils permettent de modéliser la pratique mathématique où les informations implicites jouent un rôle important. Ces mécanismes sont aussi utiles d'un point de vue génie logiciel puisqu'ils permettent de faire des développements génériques.

Arguments implicites

Dans Coq, et plus généralement en théorie des types, chaque terme possède un type unique. Des informations de type sont associées à chaque objet du système. Une fonctionnalité de Coq est de permettre à l'utilisateur d'omettre certaines informations de type. Le système va alors inférer automatiquement ces informations à partir du contexte. Par exemple, l'opération de composition de fonctions est définie dans Coq par :

```
Definition compose (A B C : Type) (g : B → C) (f : A → B) :=
 fun x : A => g (f x).
```

Dans cette définition la fonction `compose` prend cinq arguments : trois types et deux fonctions entre ces types. Lors de l'application de la fonction `compose` il n'est pas nécessaire de donner tous ses arguments. En effets, les trois premiers arguments peuvent être déduits à partir des types des deux derniers arguments.

Pour laisser à Coq la tâche d'inférer certains arguments de fonctions, l'approche standard est d'utiliser le `_` à la place de ces arguments. Par exemple, pour deux fonctions `f g : nat → nat` sur les entiers naturels, la fonction composition est donnée par `compose _ _ _ g f`. Dans cette exemple, à la place des `_` le système va placer le type `nat`.

Une autre approche plus élégante est d'utiliser les arguments implicites. Ce mécanisme du système permet d'omettre certains arguments lors de l'application d'une fonction. Il doit être activé avec la déclaration suivante :

```
Set Implicit Arguments.
```

Avec cette déclaration, la liste des arguments implicites de chaque fonction définie sera automatiquement calculée par le système. La composition des deux fonctions `f g` de l'exemple précédent est donnée par `compose g f`. Avec l'activation du mécanisme des arguments implicites seuls les arguments non implicites doivent être mentionnés lors de l'application de fonction. Par ailleurs, il est possible de désactiver localement ce mécanisme et d'expliciter tous les arguments d'une fonction donnée. Il faut alors utiliser un `@` avant le nom de cette fonction. Dans la suite, le mécanisme des arguments implicites sera activé par défaut.

Coercion implicite

En mathématique un même objet est souvent associé à des types différents. Par exemple, un anneau est aussi un groupe et un corps est aussi un anneau. Dans la littérature mathématique ce changement de point de vue est souvent présent de façon implicite. Dans Coq, un mécanisme de conversion de type implicite ou *coercion* [50] est défini. Ce mécanisme permet de convertir le type d'un objet donné de façon transparente à l'utilisateur. Il définit une notion d'héritage entre types.

Pour définir une *coercion* entre deux types donnés, il faut avoir une fonction qui, à chaque objet du type de départ, associe un objet du type d'arrivée. Cette fonction de conversion doit être définie pour tous les objets du type de départ. Cette condition sur la fonction de conversion est appelée *condition d'héritage uniforme*. Par exemple, pour déclarer la *coercion* standard entre les booléens et les entiers naturels, il faut définir la fonction de conversion suivante :

```
Structure monoid : Type := Monoid {
 sort :> Type;
 op : sort → sort → sort; e : sort;
 opA : forall x y z, op x (op y z) = op (op x y) z;
 opIe : forall x, op x e = x; opeI : forall x, op e x = x }.
```

FIGURE 1.2 – Définition de l'interface de monoïde avec les *télescopes*.

```
Definition nat_of_bool (b : bool) := if b then 1 else 0.
```

La fonction `nat_of_bool` prend en entrée un booléen et retourne `1` s'il est égale à `true` et `0` sinon. Cette fonction de conversion est déclarée comme *coercion* implicite avec le mot clé `Coercion` et la syntaxe suivante :

```
Coercion nat_of_bool : bool >-> nat.
```

Avec cette déclaration, nous allons pouvoir utiliser des objets de type `bool` comme des `nat`. Par exemple, nous pouvons écrire `forall n, n + true = S n`. Dans cet exemple une *coercion* est introduite automatiquement par Coq. En effet après la fonction `+` d'addition des entiers, Coq s'attend à des arguments de type `nat`. Si les arguments passés ne sont pas de type `nat`, le système cherche alors une *coercion* du type de ces arguments vers celui attendu. Dans l'exemple ci-dessus, Coq va chercher une coercion de `bool` vers `nat`. Cette *coercion* est définie par `nat_of_bool` et est insérée implicitement. L'exemple précédent correspond explicitement à

```
forall n, n + (nat_of_bool true) = S n.
```

Dans l'exemple ci-dessus, grâce aux *coercions* nous avons pu utiliser la fonction `+` qui est définie pour des entiers sur un argument d'un autre type. D'un point de vue génie logiciel, cette utilisation des *coercions* implicites permet de développer des théories génériques. Les opérateurs définis sur un type donné peuvent ainsi être réutilisées pour un autre type.

Un mécanisme spécial de déclaration de *coercion* sur des *types enregistrements* est défini dans Coq. Ce mécanisme qui est souvent appelé *télescopes* [8] permet de convertir implicitement un objet de *type enregistrement* vers un des champs de cette *enregistrement*. Par exemple, la définition du type `monoid` peut être modifiée comme dans la Figure 1.2 pour définir une *coercion* entre le type `monoid` et le type de ses éléments. Ceci est fait avec la déclaration `sort :> Type` du type des éléments du monoïde. Ainsi, pour un `M : monoid`, lorsque nous écrivons `x : M`, Coq l'interprétera comme `x : sort M`. Cette technique permet d'adopter la pratique mathématique standard d'associer la

structure algébrique au type de ses éléments. Elle permet aussi de modéliser la relation hiérarchique entre les structures algébriques. Ces structures sont définies suivant un schéma hiérarchique. Par exemple, les anneaux sont définis à partir des groupes abéliens et les corps sont définis à partir des anneaux. Les *coercions* permettent de modéliser cette relation hiérarchique entre les structures algébriques.

Structures canoniques

Dans Coq, un mécanisme est défini pour permettre la déclaration d'un type abstrait ou concret donné comme une instance d'un *type enregistrement*. Ce mécanisme est appelé structure canonique. Il a été introduit pour la première fois dans Coq par A. Saïbi pour ses travaux sur la théorie des catégories [27]. Les structures canoniques sont similaires aux *type classes* présents dans différents langages de programmation fonctionnelle. Les *type classes* ont été récemment introduits dans Coq [51].

Une instance déclarée comme structure canonique va être utilisée lors du processus d'inférence de type dans des équations invoquant des objets d'un *type enregistrement*. Pour que la définition d'une instance soit reconnue comme canonique par le système, il faut la déclarer avec le mot clé `Canonical Structure`. Par exemple, la déclaration suivante redéfinit `nat_monoid` comme une instance canonique de `monoid` pour le type `nat` :

```
Canonical Structure nat_monoid := @Monoid nat + 0 ...
```

Après cette déclaration, lorsque nous écrivons `op 2 3`, le système pourra inférer l'argument implicite caché. En effet, le terme `op 2 3` correspondent explicitement à `op _? 2 3`. Dans ce terme, le `_?` est un objet de type `monoid` et dont le champ `sort` est égale à `nat`. Grâce aux structures canoniques, Coq va inférer automatiquement que `_?` est égal à `nat_monoid`.

Les structures canoniques sont une forme d'inférence de type pour les enregistrements de Coq qui sont des types enrichis. Elles permettent d'inférer une instance d'un type enregistrement à partir de la valeur de l'un de ses champs. Pour que l'inférence de structure fonctionne normalement, il faut que lors de la déclaration d'une structure canonique les valeurs données aux champs de l'enregistrement soient la seule combinaison possible pour construire l'instance du type enregistrement cible. Lorsqu'il y a plus qu'une combinaison possible, la déclaration de plusieurs structures canoniques est possible mais seule la première structure canonique déclarée est reconnue par le système. Par exemple, pour la structure `monoid`, nous avons déclaré avec `nat_monoid` une instance de `monoid` avec `nat` comme `sort`, `+` et `0` comme `op` et `e`. Il est aussi possible

de déclarer une nouvelle instance avec `nat` toujours comme `sort` mais avec la multiplication des entiers `*` et `1` comme `op` et `e`. Avec ces nouvelles valeurs, les axiomes de monoïde sont toujours valides.

```
Canonical Structure nat_mul_monoid := @Monoid nat * 1 ...
```

Par contre, la structure canonique `nat_mul_monoid` déclarée ci-dessus ne sera pas automatiquement reconnue comme une instance de `monoid`. Seule la structure `nat_monoid` déclarée précédemment avec `+` est reconnue par le système et sera donc utilisée dans le processus d'inférence de type.

Dans notre développement, la condition sur l'unicité de la construction d'une instance de structure algébrique se manifeste principalement avec les monoïdes. Pour les autres structures algébriques comme les groupes ou les anneaux, la construction d'une instance de ces structures pour un type d'éléments donné ne peut se faire que d'une façon canonique. Pour les monoïdes, l'exemple des entiers naturels montrent que l'ont peut construire des monoïdes sur le type `nat` avec l'addition, la multiplication ou le maximum. Une solution à cette limitation est de changer la définition du type enregistrement. Dans le prochain chapitre, nous donnerons une nouvelle définition de la structure de monoïde qui permet de respecter la contrainte d'unicité des structures canoniques.

Les structures canoniques de Coq permettent d'utiliser des opérations et des propriétés génériques pour des instances spécifiques. Elles fournissent un mécanisme pour simuler la pratique mathématique standard d'utiliser les mêmes opérations et propriétés de structures algébriques abstraites pour des objets mathématiques distincts comme les polynômes ou les matrices. La structure algébrique correspondante va alors être inférée à partir du contexte.

Chapitre 2

SSReflect

SSReflect (pour *Small Scale Reflection* ou réflexion à petite échelle) est une extension de Coq. Elle a été initialement développée par G. Gonthier pour sa preuve formelle du théorème des quatre couleurs [19]. Elle introduit un nouveau langage de tactique qui étend celui de Coq et des bibliothèques adaptées à la réflexion à petite échelle. Elle étend aussi le langage de spécification de Coq *Gallina*. SSReflect est distribuée avec des bibliothèques pour travailler avec les types munis d'une égalité décidable et les types finis entre autres.

Ce chapitre est une introduction à SSReflect. Nous commencons par présenter le mécanisme de réflexion booléenne de SSReflect. Nous introduisons par la suite certaines des extensions de *Gallina* définies par SSReflect et son langage de tactique. Enfin, nous décrivons les principaux composants SSReflect sur lesquels se basent notre travail. Certains de ces composants ont été initialement développés pour la preuve du théorème des quatre couleurs comme les bibliothèques pour les types finis et les types décidables. D'autres comme les types avec opérateurs de choix et les opérations indexées ont été développés dans le cadre du projet *Mathematical Components*. Ils répondent à des besoins qui se sont manifestés durant le processus de développement.

Cette présentation de SSReflect n'est pas exhaustive. Elle met l'accent sur les bibliothèques de SSReflect qui n'ont pas encore une documentation complète. Plus de détails sur le développement des preuves formelles avec SSReflect et son langage de tactiques sont disponibles dans son manuel [20] et son tutoriel [22].

2.1 Réflexion booléenne

Par défaut, l'environnement de raisonnement logique de Coq est purement constructif. Dans cet environnement, les propositions logiques sont des objets de type `Prop`. Dans

ce type le tiers exclu n'est pas disponible, *i.e.* la proposition `∀ P : Prop, P ∨¬ P` n'est pas prouvable sans axiome. En plus, les objets de type `Prop` ne peuvent pas être utilisés dans la définition d'une fonction calculatoire. Il y a donc une distinction entre les propositions logiques et les valeurs booléennes. Comme nous l'avons vu dans le chapitre précédent le type des booléens est représenté par le type `bool` qui est un type inductif avec deux constructeurs. Dans le type `bool` le tiers exclu est prouvable car il correspond à une analyse de cas sur une valeur booléenne. Les objets de type `bool` peuvent être utilisés dans la définition de fonction calculatoire.

Dans Coq, les types `Prop` et `bool` fournissent deux points de vue pour représenter les preuves mathématiques. La version propositionnelle permet de profiter de l'expressivité du calcul des constructions pour coder les preuves mathématiques. La version booléenne permet de faire des calculs. Le type `bool` peut être identifié avec la sous-classe du type `Prop` des propositions logiques décidables. Dans le cadre intuitionniste de Coq, faire des preuves dans `bool` permet d'avoir le tiers exclu et donc faire des mathématiques classiques.

SSReflect définit un mécanisme pour combiner les deux visions complémentaires `Prop` et `bool`. Ce mécanisme permet d'associer la version propositionnelle d'une proposition décidable à sa version booléenne. Avec ce mécanisme, le type `bool` est injecté dans `Prop` 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 `b = true`. En plus de la coercion `is_true`, SSReflect définit un prédicat inductif qui permet une manipulation pratique et confortable de la réflexion booléenne. Le prédicat `reflect` donne une équivalence entre une proposition décidable et sa version booléenne :

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

Pour une proposition `P` et une valeur booléenne `b`, `(reflect P b)` veut dire que `P` est logiquement équivalente à `(b = true)`. Le prédicat `reflect` est équivalent à la définition Coq de l'équivalence logique standard. Par contre il permet de mieux gérer l'analyse de cas sur les booléens.

Avec l'injection de `bool` dans `Prop` et la définition du prédicat de réflexion, l'idée générale dans SSReflect est d'utiliser le prédicat `reflect` pour mettre en relation la

version logique et la version calculatoire d'une proposition décidable. La version logique sera utilisée pour faire des preuves, alors que celle calculatoire sera utilisée pour faire des calculs. Par exemple, l'équivalence entre la conjonction propositionnelle $\wedge$ et celle booléenne `&&` 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 donnée et sa représentation logique. Ces lemmes sont appelés les *lemmes de vue*. Par exemple, les lemmes `orP` et `negP` relient respectivement $\vee$ et `~` à `||` et `~~`.

En résumé, la réflexion entre `bool` et `Prop` définit une méthode pour faire des raisonnements mathématiques classiques dans le cadre intuitionniste de Coq. D'un point de vue génie logiciel, elle peut être vue comme une application du principe de séparation entre contenu et présentation aux propositions décidables. La contenu de ces propositions sera codée dans le type `bool` ; alors que la présentation sera codée dans le type `Prop` des propositions logiques de coq.

Théorie des prédicats booléens

Dans Coq et plus généralement dans la théorie des types, les ensembles peuvent être représentés par des prédicats propositionnels. Ce sont des fonctions de type `T → Prop`, avec `T` le type des éléments de l'ensemble. Ces fonctions représentent la propriété d'appartenance à l'ensemble. Lorsque cette propriété est décidable, ces prédicats peuvent être représentés par des fonctions dans bool ou prédicats booléens.

SSReflect définit une théorie générique des prédicats booléens. Cette théorie est fournie par la bibliothèque `ssrbool` qui comprend des définitions et preuves sur ces prédicats. Pour un type arbitraire `T`, le type des prédicats booléens sur `T` est défini par :

```
Definition pred (T : Type) := T → bool.
```

Ce type est le type de base des prédicats booléens. Des notations sont définies pour faciliter la manipulation des prédicats booléens. La notation `x \in A`, avec `x : T` correspond au test d'appartenance à un prédicat `A` sur `T`. La notation `[pred x : T | E]` permet de construire explicitement un prédicat sur `T` à partir de la fonction `(fun x => E)` où E est une expression booléenne pouvant contenir la variable `x`. C'est l'interface standard et conseillée pour définir des prédicats sur un type donné.

Certaines des définitions dans la bibliothèque `ssrbool` font appellent à des mécanismes avancés de Coq. Pour l'utilisation de la bibliothèque, la compréhension de ces

mécanismes n'est pas une condition nécessaire. La bibliothèque fournit des notations et définitions qui facilitent l'utilisation de cette théorie des prédicats booléens d'un point de vue utilisateur.

2.2 *Gallina* et tactiques

2.2.1 Extensions *Gallina*

SSReflect introduit une extension du langage de spécification de Coq. En particulier, elle définit une nouvelle syntaxe pour la déclaration d'arguments anonymes dans la définition des types et l'analyse de cas sur les types inductifs avec deux constructeurs.

Arguments anonymes

Dans Coq, lors de la définition d'un type inductif et lorsque le nom d'un argument du constructeur n'est pas nécessaire, il est possible d'utiliser le `_` devant le type de cette argument. Par exemple, un type pour les entiers strictement positifs peut être défini comme suit :

```
Inductive pos_nat := PosNat n (_ : 0 < n) : pos_nat.
```

Dans la définition ci-dessus, nommer la preuve que l'entier est strictement positif n'est pas nécessaire. Pour ce genre de définition, SSReflect introduit une nouvelle syntaxe qui est construite au tour des opérateurs `of` et `&`. Avec cette syntaxe, la définition du type `pos_nat` ci-dessus est équivalente à la définition suivante :

```
Inductive pos_nat := PosNat n of (0 < n).
```

Un second exemple d'utilisation de cette syntaxe SSReflect est la définition du type des listes.

```
Inductive list (T : Type) : Type := nil | cons of T & list T.
```

La définition ci-dessus du type `list` est équivalente à celle donnée dans 1.2. Cette nouvelle définition est plus proche de la syntaxe ML standard.

Analyse de cas

SSReflect définit une nouvelle syntaxe pour l'analyse de cas sur des types inductifs avec deux constructeurs.

`if` $\langle terme \rangle_1$ `is` $\langle pattern \rangle_1$ `then` $\langle terme \rangle_2$ `else` $\langle terme \rangle_3$`.`

La construction ci-dessus est équivalente dans Coq standard à l'expression suivante :

```
1 move=> v gf; apply: (iffP idP) => Hu; rewrite /mem_gf; last first.
2   by rewrite big1 // nth_nil !mul0v.
3 case: (Hu v); first by exists v; apply/eqP.
4 by rewrite big1 // nth_nil.
```

FIGURE 2.1 – Exemple de script écrit avec des tactiques de SSReflect.

`match` $\langle terme \rangle_1$ `with` $\langle pattern \rangle_1$ `=>` $\langle terme \rangle_2$ `|` `_` `=>` $\langle terme \rangle_3$ `end`.

L'opérateur `if` ... `is` ... fournit une généralisation de la construction `if` pour les types inductifs avec deux constructeurs. Dans SSReflect, l'utilisation de cette dernière construction est restreinte au type des booléens.

2.2.2 Tactiques

L'écriture de script de preuve est une partie importante du travail de formalisation des mathématiques. SSReflect définit un nouveau langage de tactique. Ce langage modifie le fonctionnement de certaines des tactiques standards de Coq et introduit de nouvelles tactiques et tacticielles à cet ensemble. Nous présentons ici brièvement ce langage. Une présentation détaillée est donnée dans le manuel de référence de SSReflect [20].

Un exemple de script écrit avec le langage de tactiques de SSReflect est donné dans la Figure 2.1. Dans ce langage, toutes les opérations qui consistent à déplacer ou généraliser depuis ou vers le contexte courant des formules sont fournies par la tactique `move` et les tacticielles `:` et `=>`. Par exemple, dans la première ligne du script de la Figure 2.1, le `move` correspond à introduire à partir du but et dans le contexte deux objets qui seront nommés suivant les arguments donnés après le `=>`. Plus généralement placer la tacticielle `=>` après une tactique permet d'introduire de nouvelles hypothèses ou variables dans le contexte à partir du but courant.

Dans SSReflect, la tactique de réécriture standard de Coq est modifiée considérablement. Le `rewrite` de SSReflect 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, dans la première ligne du script ci-dessus, le `rewrite` permet de déplier la définition `mem_gf`. Dans la second ligne de ce script, le `rewrite` permet de réécrire avec `big1`, puis simplifier le terme obtenu, ensuite réécrire avec `nth_nil` et enfin réécrire au moins une fois avec `mul0v`.

Le mécanisme de réflexion entre les propositions décidables et les booléens décrit dans la section précédente est intégré au langage de tactique de SSReflect. Par exemple,

dans la troisième ligne de l'exemple de la Figure 2.1, la tactique `apply/eqP` applique le *lemme de vue* `eqP` au but courant. Ceci correspond à remplacer la proposition logique à prouver par sa version booléenne ou l'inverse. Il est aussi possible d'utiliser le mécanisme de vue avec d'autres tactiques. Par exemple, la tactique `move/eqP: H` permet d'appliquer le même lemme de vue précédent à une hypothèse `H` du contexte.

En plus de l'extension des tactiques standards de Coq, SSReflect définit de nouvelles tacticielles. L'idée avec ces tacticielles est de faciliter la gestion des sous bus et l'organisation et la maintenance du script de preuve. En particulier, les tacticielles `first` et `last` permettent respectivement la sélection du premier et du dernier sous but générés après l'exécution d'une tactique donnée. Dans la troisième ligne du script de la Figure 2.1, la tacticielle `first` permet de sélectionner le premier but généré par l'analyse de cas. La combinaison des tacticielles `last` et `first`, comme dans la première ligne du script de la Figure 2.1, change l'ordre des sous buts. Ceci permet de résoudre en premier le sous but le plus facile et assurer la linéarité du script de preuve. La tacticielle `by` permet de clore un but ou sous but donné. Placer un `by` devant une tactique ou une suite de tactiques, comme dans l'exemple précédent, correspond à dire que ces tactiques doivent résoudre le but courant. Ceci joue un rôle important dans la maintenance des scripts de preuve et permet de localiser les erreurs générées par des changement dans la théorie manipulée par le script de preuve.

En résumé, les nouvelles tactiques et tacticielles introduites par SSReflect permettent de réduire la taille des scripts de preuves. En pratique, les scripts de preuve écrits avec SSReflect se révèlent plus concis que ceux écrits avec les tactiques Coq standards. Par ailleurs, SSReflect avec ces nouvelles tacticielles introduit une notion d'organisation du script de preuve dans le langage d'écriture de ces scripts. Dans des développements de script de preuve de grande envergure, cette notion d'organisation est importante. Elle permet de faciliter la maintenance de ces développements.

2.3 Bibliothèques

En plus du langage de tactique et de l'extension *Gallina*, SSReflect définit aussi des nouvelles bibliothèques formelles. Ces bibliothèques contiennent des définitions et interfaces qui prennent en compte la réflexion à petite échelle. L'idée générale dans ces bibliothèques est de représenter les propriétés et relations logiques décidables par des valeurs booléennes. Pour prendre en compte des axiomes mathématiques classiques comme l'axiome du choix, SSReflect définit des interfaces génériques qui modélisent ces axiomes. Avec ces interfaces, les axiomes mathématiques classiques vont être associés à

un type donné. Un exemple de catégorie de types où ces axiomes sont toujours valides sont les types finis. SSReflect définit une interface générique abstraite qui modélise les types finis. Par ailleurs, SSReflect redéfinit des bibliothèques standards de Coq. Ceci est motivé par un objectif d'homogénéité avec les développements sur la réflexion entre `bool` et `Prop`.

Dans la suite de cette section, nous allons présenter certains aspects des bibliothèques de SSReflect qui jouent un rôle important dans notre développement. Les définitions présentées ici sont utiles pour comprendre celles que nous donneront dans les prochains chapitres.

2.3.1 Type avec une égalité décidable

Egalité de Leibniz et *setoid*

La notion d'égalité joue un rôle important dans la formalisation des mathématiques sur ordinateur. Dans le système Coq, l'égalité n'est pas primitive. Elle est définie en utilisant un type inductif.

```
Inductive eq (A : Type) (x : A) : A → Prop := refl_equal : eq A x x.
```

Cette égalité, qui est appelée égalité de Leibniz, correspond à une égalité syntaxique modulo réduction. Deux termes sont égaux s'ils sont exactement les mêmes ou se réduisent, suivant les règles de conversion du CCI, au même terme. Le système définit la notation infixe `x = y` pour la relation `eq x y`. Le premier argument de `eq` est inféré à partir des arguments `x` et `y` qui doivent impérativement avoir le même type.

Dans le système, la réécriture avec l'égalité de Leibniz est prise en charge. Elle découle du principe d'élimination de la définition inductive de `eq`.

```
eq_ind : forall (A : Type) (x : A) (P : A -> Prop),
 P x -> forall y : A, x = y -> P y
```

Une propriété importante de l'égalité de Leibniz est qu'elle est préservée par toute fonction. Cette propriété découle du principe d'élimination de `eq` et est donnée par le lemme suivant :

```
Lemma f_equal :
 forall (A B : Type) (f : A → B) (x y : A), x = y → f x = f y.
```

Nous pouvons ainsi réécrire avec une hypothèse d'égalité dans les arguments de fonctions.

Par ailleurs cette égalité est intentionnelle c'est à dire qu'il n'est pas possible de prouver l'axiome d'extensionnalité. En notant `f =1 g` l'égalité extensionnelle entre fonctions, *i. e.* `(forall x, f x = g x)`, l'axiome d'extensionnalité correspond à :

```
forall (A B : Type) (f g : A → B), f =1 g → f = g.
```

Une autre limitation de cette égalité vient du fait qu'elle ne correspond pas toujours à l'égalité utilisée en mathématiques. En effet, les mathématiciens utilisent souvent une égalité qui correspond en fait à une relation d'équivalence. Par exemple dans la pratique mathématique, le $=$ dans $\frac{1}{3} = \frac{3}{9}$ n'est pas une égalité syntaxique mais une relation d'équivalence.

Pour supporter l'égalité mathématique générale, Coq implémente le mécanisme des *setoids* [5]. Un *setoid* est un type avec une relation d'équivalence sur ce type. Pour un type donné `T`, cette relation a le type `T → T → Prop`. Un avantage des *setoids* est de permettre de définir une notion d'égalité extensionnelle sur les fonctions entre *setoids*. Par contre la propriété `f_equal` n'est plus valable dans le cas général lorsque l'égalité est une égalité de *setoid*. Ainsi pour pouvoir réécrire avec une égalité de *setoid* dans les arguments d'une fonction il faut prouver que la propriété `f_equal` s'applique à cette fonction, après avoir remplacé l'égalité de Leibniz par celle de *setoid*. En d'autres termes, il faut montrer que la fonction passe au quotient pour la relation d'équivalence (l'égalité de *setoid*) sur son domaine. Ceci peut être un travail fastidieux surtout dans un travail de formalisation de grande échelle comme pour le théorème de Feit-Thompson.

Egalité décidable

Dans SSReflect, l'idée est de combiner la généralité des relations de *setoid* et la facilité d'utilisation de l'égalité de Leibniz. Pour cela, une notion d'égalité décidable est définie. C'est une relation binaire booléenne qui est équivalente à l'égalité de Leibniz. La Figure 2.2 donne la définition de la structure abstraite `eqType` qui définit les types munis d'une égalité décidable. Dans la définition d'`eqType`, le symbole `:>` déclare `sort` comme une *coercion*. Ceci nous permet d'utiliser un `T : eqType` comme si c'était un `Type`. La propriété `eqP` permet de dire que l'égalité décidable `eqb` est équivalente à celle de Leibniz. Avec l'utilisation des arguments implicites, le constructeur `EqType` du type `eqType` va prendre en paramètre cette propriété. Les autres champs de la structures seront alors inférés à partir de la preuve de la propriété `eqP`.

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

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

FIGURE 2.2 – Définition de l'interface de `eqType`.

seule preuve de l'égalité pour chaque paire d'objets égaux. Le lemme suivant correspond à cette propriété :

```
Lemma eq_irrelevance :
 forall (T : eqType) (x y : T) (e1 e2 : x = y), e1 = e2.
```

Ce lemme pourra être utilisé pour des preuves d'égalités sur des objets d'un type qui possèdent une égalité décidable. Pour cela il suffira de déclarer une structure canonique d'`eqType` pour ce type. Par exemple, le lemme suivant fournit une instance de la propriété de *proof-irrelevance* pour les booléens :

```
Lemma bool_irrelevance : forall (x y : bool) (E E' : x = y), E = E'.
```

La preuve du lemme ci-dessus correspond à appliquer le lemme `eq_irrelevance`. La structure d'`eqType` correspondante va être inférée automatiquemet par Coq grâce aux structures canoniques.

La structure de `eqType` est un cas particulier de *setoid*. Dans cette structure, la relation d'équivalence est décidable et hérite des propriétés de réécriture de l'égalité de Leibniz. Ceci nous permet de profiter de la puissance de réécriture de Coq. Dans la suite, les notations infixes `x == y` et `x != y` vont correspondre à l'égalité et l'inégalité booléenne entre des éléments d'un `eqType`.

2.3.2 Opérateur de choix

Les *setoids* de Coq fournissent un mécanisme pour définir des types quotients. La relation d'équivalence correspondante doit alors être mentionnée explicitement. Ceci posent des problèmes du point de vue de l'utilisation pratique comme nous l'avons mentionné dans la Section 2.3.1. Pour pouvoir définir des types quotients et où l'égalité est décidable, nous définissons une structure pour les types munis d'un opérateur de choix canonique. En utilisant l'interface `eqType`, l'opérateur de choix et l'égalité décidable vont nous permettre d'avoir une égalité de Leibniz sur le type quotient et donc de profiter de la réécriture.

Définitions

Au dessus de l'interface `eqType` et suivant le même schéma, SSReflect définit une interface pour les types munis d'un opérateur de choix : `choiceType`. La structure `choiceType` correspond à un type qui est muni d'un opérateur de choix canonique. Cette structure est munie d'une *coercion* vers `eqType`. Ceci fait qu'un `choiceType` est aussi un `eqType`. Pour un type `T : choiceType`, l'opérateur de choix associé à ce type est représenté par la fonction `xchoose`. Cette fonction est de type :

```
forall P : pred T, (exists x, P x) → T.
```

La fonction `xchoose` prend en argument un prédicat booléen et une preuve qu'il existe un élément qui satisfait ce prédicat. Elle retourne un témoin canonique pour ce prédicat. La propriété de canonicité de `xchoose` est donnée par les lemmes suivants :

```
Lemma xchooseP : forall P xP, P (xchoose P xP).
Lemma eq_xchoose : forall P Q xP xQ, P =1 Q →
 xchoose P xP = xchoose Q xQ.
```

Le lemme `xchooseP` permet dire que le résultat retourné par l'opérateur de choix respecte le prédicat sur lequel se fait le choix. Le lemme `eq_xchoose` quant à lui permet de dire que l'opérateur de choix est extensionnel par rapport au prédicat et à la preuve d'existence.

Un second opérateur de choix est défini à partir de `xchoose` : c'est la fonction `choose`. Pour un type `T : choiceType`, un prédicat `P : pred T` et une valeur `x : T`, `(choose P x)` est un témoin canonique qui respectent le prédicat `P` si `x` respecte `P`. La fonction `choose` est une variante de `xchoose` qui est plus facile à manipuler. En effet, elle ne prend pas en paramètre une preuve mais une valeur par défaut. Cette valeur doit respecter le prédicat sur lequel se fait le choix pour que les deux fonctions coincident.

La structure `choiceType` fournit une interface générique pour faire des raisonnements avec l'axiome du choix sans avoir besoin d'ajouter cet axiome. Dans la pratique l'opérateur de choix pour un type donné sera construit à partir d'une fonction d'énumération. Ceci veut dire que le type doit être dénombrable. Dans Coq la construction de cette fonction d'énumération est possible sans axiome supplémentaire car le *calcul des constructions inductives* valide l'axiome du choix dénombrable.

Types Quotient

Un type quotient est défini par rapport à une relation d'équivalence sur un type donné. Avec la définition de l'interface `choiceType` nous disposons d'une méthode

pour définir des types quotients sur lesquels nous allons avoir une égalité de Leibniz équivalente à la relation du quotient. L'idée avec l'interface `choiceType` est de représenter chaque classe d'équivalence par un représentant canonique qui sera obtenu avec l'opérateur de choix. Le type quotient va correspondre à l'ensemble des représentants canoniques des classes d'équivalences.

Pour un type `T : choiceType` et une relation d'équivalence booléenne `rel : T → T → bool`, la classe d'équivalence associée à une valeur `x` est donnée par le prédicat booléen `[pred y : T | rel x y]`. L'ensemble correspondant à ce prédicat est non vide puisque la relation `rel` est réflexive et donc `x` appartient à cet ensemble. Nous pouvons alors utiliser la fonction `choose` pour avoir un représentant canonique de cet ensemble. Le quotient de `T` par la relation `rel` peut donc être représenté par la structure suivante :

```
Structure quotClass : Type := QuotClass {
 x :> T; _ : x == choose [pred y : T | rel x y] x
}.
```

Un objet de type `quotClass` correspond à un élément `x : T` et une preuve qu'il est égal au résultat de l'application de l'opérateur de choix sur sa classe d'équivalence. En d'autres termes, il est le représentant canonique de l'ensemble des éléments qui sont en relation avec lui par la relation `rel`. La propriété de réflexivité de `rel` permet d'assurer que `(choose [pred y : T | rel x y] x)` retourne un représentant canonique de la classe d'équivalence de `x`.

Le type quotient `quotClass` est un nouveau type qui est différent de `T`. La déclaration `x :> T` définit une coercion de `quotClass` vers `T`. Ceci permet d'utiliser les opérations sur `T` avec des objets de `quotClass`. En particulier nous allons pouvoir utiliser la relation `rel` sur des objets de type `quotClass`. Dans ce type la relation `rel` va être équivalente à l'égalité de Leibniz. En effet, dire que deux objets `a b : quotClass` sont égaux par `rel` veut dire que les ensembles de leur classe d'équivalence sont extensionnellement égaux :

```
rel a b → [pred y : T | rel a y] =1 [pred y : T | rel b y].
```

Ceci est donnée par les propriétés de symétrie et de transitivité de la relation `rel`. Cette égalité entre les ensembles des classes d'équivalences et les propriétés de canonicité de `choose` impliquent que :

```
choose [pred y : T | rel a y] a = choose [pred y : T | rel b y] b.
```

En partant de cette égalité et étant donné que `a` et `b` sont des représentants canoniques de leur classe d'équivalence, nous prouvons le lemme suivant :

```
Lemma quot_eqP : forall a b : quotClass, reflect (a = b) (rel a b).
```

Ainsi sur le type quotient l'égalité de Leibniz est équivalente à la relation `rel` par rapport à laquelle est construit le quotient.

Dans la définition du type quotient, la classe d'équivalence est représentée par un élément canonique. Il est possible de vouloir que cet élément en plus d'appartenir à la classe d'équivalence respecte un prédicat donné. Pour cela il faut avoir une fonction qui construit un élément qui respecte ce prédicat et ainsi avoir la propriété que l'ensemble correspondant est non vide. Cette propriété va nous permettre d'utiliser l'opérateur de choix sur cette ensemble. La définition du type quotient que nous avons donnée précédemment ne va différer qu'au niveau du prédicat sur lequel se fait le choix. Nous reviendrons à ce cas particulier de quotient lors de la définition du quotient par l'égalité entre sous-espaces engendrés dans le chapitre sur la théorie des représentations des groupes.

2.3.3 Bibliothèques standards : entiers et listes

Les entiers et les listes sont deux structures de données de base qui jouent un rôle important dans la formalisation sur ordinateur et la programmation en générale. Pour prendre en compte l'injection de `bool` dans `Prop` et l'interface `eqType`, SSReflect redéfinit les bibliothèques standards de Coq pour les entiers et les listes. Ces bibliothèques appliquent la convention de SSReflect de définir toute proposition ou relation décidable dans le type `bool`. Dans cette section nous présentons certains aspects des ces bibliothèques que nous aurons à utiliser dans la suite.

Entiers

La bibliothèque `ssrnat` de SSReflect définit l'arithmétique des entiers naturels. Elle définit une égalité booléenne sur le type `nat`. Cette égalité permet de déclarer une structure canonique de `eqType` pour les entiers naturels. En partant de cette égalité décidable, les opérations de comparaison entre les entiers sont définies comme des relations booléennes. Ceci n'est pas le cas dans Coq standard où ces opérations sont définies par des prédicats inductifs dans `Prop` [11].

Dans `ssrnat`, les relations de comparaison entre les entiers sont définies à partir de la relation inférieure ou égale. Cette dernière est définie par la relation booléenne suivante :

```
Definition leq m n := m - n == 0.
```

Dans la définition ci-dessus, les arguments `m` et `n` sont de type `nat`. La déclaration d'une structure canonique de `eqType` pour `nat` nous a permis d'utiliser la notation `==` pour l'égalité booléenne entre les éléments d'un `eqType`. La définition de `leq` dans `bool` permet de l'utiliser dans les définitions de fonction. Elle pourra toujours être utilisée dans les énoncés de théorèmes grâce à la coercion de `bool` dans `Prop`.

En partant de la définition de `leq` et de la notation `n.+1` pour le successeur, la relation inférieure strictement correspond à une notation :

```
Notation "m < n" := (m.+1 <= n).
```

Cette définition va permettre d'utiliser des lemmes prouvés pour `leq` avec la relation `<`. Par exemple nous n'avons pas à prouver que :

```
forall n m p, m < n → n <= p → m < p.
```

La propriété ci-dessus est en fait une instance du lemme de transitivité de `leq` car elle correspond explicitement à :

```
forall n m p, m.+1 <= n → n <= p → m.+1 <= p.
```

La définition de `<` comme un cas spécial de `<=` permet à ces deux relations de partager certains lemmes. Ceci permet entre autre de réduire la taille de la bibliothèque.

En plus du développement sur les relations de comparaison entre les entiers, la bibliothèque `ssrnat` comprend aussi des définitions et preuves sur le maximum (`maxn`), le minimum (`minn`) et la puissance (`m^n`).

Listes

SSReflect définit une nouvelle bibliothèque pour les listes. Cette bibliothèque comprend la définition d'un nouveau type pour les listes et des opérations génériques sur les listes. Avec la syntaxe SSReflect pour les arguments anonymes, le type des listes est défini comme suit :

```
Inductive seq (T : Type) : Type := nil | cons of T & seq T.
```

Au-dessus de cette définition, des notations sont définies. La notation `[::]` correspond à la liste vide ou le constructeur `nil`. La notation `(x :: s)` correspond à `(cons x s)` ou la fonction d'ajout d'un élément en tête de liste. Dans `(cons x s)`, le type `T` des éléments de la liste va être inféré à partir du type de `x`. La notation `[:: x]` correspond à liste qui contient un seul élément `x`.

Le type `seq` est paramétré par le type des éléments de la liste. Dans la définition de `seq` le type des éléments ne possède pas de structure particulière. Lorsque ce type est muni d'une égalité décidable, cette propriété peut être transférée au type `seq` associé. En effet, pour un type `T : eqType` nous avons la relation binaire booléenne suivante sur `seq T` :

```
Fixpoint eqseq (s1 s2 : seq T) {struct s2} :=
 match s1, s2 with
 | [::], [::] => true
 | x1 :: s1', x2 :: s2' => (x1 == x2) && eqseq s1' s2'
 | _, _ => false
 end.
```

La relation `eqseq` est une égalité décidable sur `seq`. Elle est équivalente à l'égalité de Leibniz. Le lemme suivant prouve cette équivalence.

```
Lemma eqseqP : forall s1 s2 : seq T, reflect (s1 = s2) (eqseq s1 s2).
```

La preuve du lemme ci-dessus nous permet de déclarer une structure canonique de `eqType` sur les listes.

```
Canonical Structure seq_eqType := EqType eqseqP.
```

Ainsi pour tout type `T : eqType`, le type `seq T` sera automatiquement muni d'une structure de `eqType`. Nous pouvons alors utiliser les définitions et propriétés génériques associées à `eqType` pour les listes sur un `eqType`.

Par ailleurs, lorsque le type des éléments est muni d'un opérateur de choix, le type des listes correspondant peut être muni d'un opérateur de choix. Avec les structures canoniques ce transfert de structure de `choiceType` est pris compte automatiquement comme dans le cas de la structure `eqType`.

En plus de la construction des structures canoniques de `eqType` et `choiceType` pour les listes, la bibliothèque définit aussi des opérations standard sur les listes. Un exemple d'opération sur les listes est la fonction d'accès à un indice d'une liste donnée. Cette fonction est définie comme suit :

```
Fixpoint nth x0 s n {struct n} :=
 if s is x :: s' then if n is n'.+1 then nth x0 s' n' else x else x0.
```

La définition ci-dessus utilise la syntaxe SSReflect pour l'analyse de cas présentée dans la Section 2.2.1. Dans cette définition, `x0` correspond à la valeur par défaut qui sera retournée lorsque la liste `s` est vide ou lorsque l'entier `n` est plus grand que la longueur

de `s`. Dans la suite lorsque le type des éléments de la liste possède un élément par défaut (par exemple le zéro d'un groupe), la notation `s‘_i` correspondra à `nth 0 s i`.

Un second exemple d'opération sur les listes définies dans SSReflect est la fonction qui compte le nombre des éléments d'une liste qui respectent un prédicat donné. Cette fonction est définie comme suit :

```
Variable a : pred T.
Fixpoint count s := if s is x :: s' then a x + count s' else 0.
```

La fonction `count` est définie par récurrence sur la structure de la liste donnée en argument. Elle retourne un objet de type `nat`. Dans la première branche du `if`, la *coercion* de `bool` dans `nat` permet de convertir le terme `a x` qui est un booléen en un entier.

Par ailleurs, la bibliothèque définit les fonctions standard `map` et `foldr` sur les listes. Cette dernière correspond à l'opération standard *fold* utilisée en programmation fonctionnelle. Un nouveau constructeur pour le type `seq` est aussi défini. C'est la fonction `mkseq` qui prend en paramètre une fonction d'indice sur les entiers et une borne qui va correspondre à la longueur de la liste à construire. Par exemple, pour une fonction `f` donnée, `(mkseq f n)` retourne la liste `[(f 0):: (f 1):: ... :: (f n-1)]`.

La bibliothèque sur les listes contient aussi des preuves de résultats génériques sur les listes. Un exemple est le lemme suivant sur l'opération de concaténation de listes :

```
Lemma size_cat : forall (s1 s2 : seq T),
 size (s1 ++ s2) = size s1 + size s2.
```

Le lemme ci-dessus prouve que la taille de la concaténation de deux listes (notation `++`) est égale à la somme des tailles de ces listes.

2.3.4 Types et fonctions finis

Dans un contexte où les objets sont finis, les cadres mathématiques classique et intuitionniste peuvent être confondus. La propriété de décidabilité des proposition logiques sur ces objets permet de prouver le tiers exclus et l'axiome du choix. SSReflect définit une théorie générique des types finis. Cette théorie comprend une interface qui modélise les types finis. En plus de cette interface, SSReflect définit une théorie des fonctions finies. Ce sont les fonctions dont le support est un type fini. Sur ces fonctions nous avons la propriété que l'axiome de l'égalité extensionnelle entre fonctions peut être prouvé. Ceci, en plus des propriétés du tiers exclus et de l'existence d'un opérateur de choix, nous permet de faire des raisonnements mathématiques classiques sur ces objets.

```
Structure finType : Type := FinType {
 sort :> choiceType;
 enum_all : seq sort;
 enumP : forall x : sort, count [pred y | x == y] enum_all = 1
}.
```

FIGURE 2.3 – Définition de l'interface de `finType`.

Définitions

Un type fini correspond à un ensemble fini d'éléments. Il peut donc être représenté par la liste de ses éléments. La propriété de finitude fait qu'un type fini est aussi muni d'un opérateur de choix et d'une égalité décidable. Le type des types finis va donc être un sous-type de `choiceType` et transitivement de `eqType`. Dans SSReflect, le type des types finis est représenté par la structure `finType`. La Figure 2.3 donne la définition de la structure abstraite des types finis [1]. Dans cette définition un type fini est défini par un type de ses éléments, une liste sur ce type et une preuve que cette liste énumère tous les éléments de ce type. Avec les *télescopes*, la projection `sort` définit une *coercion* de `finType` vers `choiceType` et transitivement `eqType`. Dans la définition de `finType`, la proposition `enumP` permet d'assurer que chaque élément du type `sort` apparait une seule fois dans la liste `enum_all`. Ceci veut dire que cette liste énumère tous les éléments du type de représentation.

La structure `finType` définit un type abstrait pour les types finis. Sur ce type nous avons des opérations génériques sur les types finis. En particulier, nous avons les définitions suivantes :

```
Definition enum (T : finType) (P : pred T) := filter P (enum_all T).
Definition card (T : finType) (A : pred T) := size (enum A).
```

La fonction `enum` retourne la liste des éléments d'un type fini donné qui respectent un prédicat booléen sur ce type. La définition de `enum` utilise la fonction `filter` sur les listes. Cette fonction filtre les éléments d'une liste par rapport à un prédicat. La fonction `card` calcul le cardinal d'un ensemble ou prédicat booléen sur un type fini. La notation `#|A|` correspond à `card A`. Le type fini `T` sera inféré à partir du type du prédicat `A`.

1. Dans le code source courant les structures `finType` et `eqType` sont définies en utilisant le mécanisme des *mixin/class* sur lequel nous reviendrons plus tard. Les définitions données ici correspondent à l'idée générale des types représentés par ces structures.

Quantificateurs : dans Coq les quantificateurs universels et existentiels permettent de construire des propositions. Les propositions construites par ces quantificateurs sont dans le type `Prop`. Lorsque ces quantificateurs portent sur un type fini, ils peuvent être décrits avec des valeurs booléennes, *i.e.* des objets de type `bool`. En particulier, SSReflect définit les quantificateurs booléens `forallb` et `existsb` pour représenter les quantificateurs universels et existentiels sur des variables d'un type fini. Les lemmes suivants donnent l'équivalence entre ces quantificateurs booléens et les quantificateurs par défaut de Coq :

```
Lemma forallP : forall T (P : pred T),
 reflect (forall x : T, P x) (forallb x : T, P x).
Lemma existsP : forall T (P : pred T),
 reflect (exists x : T, P x) (existsb x : T, P x).
```

Les quantificateurs booléens `forallb` et `existsb` vont nous permetre de définir des prédicats booléens qui utilisent une quantification. Les lemmes de réflexion ci-dessus vont permettre la transition entre la représentation booléenne et celle propositionnelle de ces quantificateurs.

Instance de type fini : une instance de type fini que nous utilisons intensivement est le type des entiers naturels strictement plus petit qu'une borne supérieure donnée. Pour un entier naturel n, ce type correspond à l'ensemble $\{0, 1, \ldots, n-1\}$. Dans SSReflect, cet ensemble est représenté par le type inductif suivant :

```
Inductive ordinal : Type := Ordinal m of m < n.
```

Dans la définition ci-dessus, le type `ordinal` dépend de l'entier `n` (la borne supérieure). C'est un type inductif dépendant avec un seul constructeur. Ce constructeur correspond à la donnée d'un entier et d'une preuve que cet entier est inférieur à la borne. Le type `ordinal` est un nouveau type qui est différent du type `nat` des entiers. Cependant les objets de type `ordinal` peuvent être manipulés comme des entiers grâce à une *coercion*.

```
Coercion nat_of_ord i := let Ordinal m _ := i in m.
```

Avec la déclaration de la *coercion* `nat_of_ord`, nous allons pouvoir utiliser les opérations sur les `nat` avec les objets de type `ordinal`. Dans la suite la notation `'I_n` représentera le type `ordinal n`.

Une structure canonique de `finType` est déclarée pour `ordinal`. Elle permet l'utilisation des opérations génériques de `finType` sur ce type. En particulier, pour un entier `n` et un prédicat `P : pred 'I_n`, nous pouvons ainsi écrire `forallb x : 'I_n, P x`.

Cette formule correspond à dire qu'une propriété logique donnée est valide pour tous les entiers plus petits qu'un entier donné. Nous aurons souvent à utiliser ce genre de formule pour des définitions en algèbre linéaire de dimension finie.

Fonctions à support fini

Une fonction dont le domaine est un type fini peut être représentée par son graphe. Ce graphe correspond à une liste de valeurs sur le co-domaine. Comme le domaine est fini, le graphe peut être représenté par un vecteur dont la longueur correspond au cardinal du domaine fini. SSReflect définit un type spécial pour les fonctions dont le domaine est un type fini.

Pour un type fini `aT : finType` et un type `T : Type`, le type `{ffun aT → T}` représente le type des fonctions de `aT` dans `T`. Ce type est muni d'une *coercion* vers les fonctions de Coq. Donc, chaque objet de ce type peut aussi être interprété comme une fonction. En plus de cette *coercion*, un constructeur pour le type `ffun` à partir des fonctions est aussi défini. Ce constructeur est donnée par `[ffun x : aT => F]` où est `F` est une expression dépendant de `x`. La *coercion* vers les fonctions de Coq et le constructeur à partir de ces fonctions permettent d'enrichir la définition du type des fonctions finies. Avec ces opérations, le type `ffun` est muni d'une couche de présentation. Cette couche correspond à la notion de fonction standard dans Coq ou lambda expression. Dans le contexte de Coq et du point de vue de l'utilisateur, ces fonctions sont plus faciles à manipuler que le graphe de la fonction.

L'intérêt de définir un type pour les fonctions à domaine fini est de pouvoir prouver l'équivalence entre l'égalité de Leibniz et l'égalité extentionnelle pour ces fonctions. Comme nous l'avons vu dans la Section 2.3.1, cette équivalence n'est pas prouvable pour les fonctions générales de Coq. Par contre pour les fonctions à support fini, comme elles possèdent une représentation finie (leur graphe), cette équivalence peut être prouvée. Le lemme suivant donne cette équivalence :

```
Lemma ffunP : forall f1 f2 : {ffun aT → T}, f1 =1 f2 ↔ f1 = f2.
```

La partie gauche de l'équivalence ci-dessus donne l'égalité extentionnelle entre l'interprétation fonctionnelle des fonctions à support fini. La *coercion* de `ffun` vers le type des fonctions est ici implicitement introduite par le système.

Sur ces définitions des types finis et des fonctions à support fini, SSReflect construit une formalisation de la théorie des groupes finis [21]. Cette formalisation contient en particulier un développement sur les groupes de permutations et des preuves du théorème

de Sylow et du lemme de Cauchy-Frobenius. Ces résultats sont des pré-requis à la formalisation du théorème de Feit-Thompson.

2.3.5 Opérations indexées

En mathématiques et particulièrement en algèbre linéaire, les énoncés et les preuves contiennent souvent des formules de la forme

$$\sum_{0 \le i < n} \alpha_i v_i \text{ ou } \sum_{V_i \simeq W} V_i.$$

Ces formules correspondent à l'itération d'une opération binaire donnée sur un ensemble indexé. Dans la perspective d'un développement formel sur l'algèbre linéaire, une bibliothèque générique pour manipuler ces opérations indexées a été développée [7]. Nous présentons dans cette section certains aspects de cette bibliothèque, appelée `bigops`, qui sont utilisés dans les chapitres suivants.

Définitions

Les opérations indexées sont la généralisation des opérations binaires aux éléments d'une liste d'indice. Généralement, une opération indexée comprend trois composantes : l'opération binaire, l'ensemble d'indices et la fonction d'indexation. Par exemple, pour la formule $\sum_{0 \le i < n} i^2$ l'opération binaire est l'addition des entiers, l'ensemble d'indices est $\{0, 1, \ldots, n-1\}$ et la fonction d'indexation est donnée par $x \mapsto x^2$. L'opération indexée associée à une opération binaire quelconque correspond à appliquer de façon itérative l'opération binaire aux images par la fonction d'indexation de l'ensemble d'indice. La fonction suivante implémente ce processus :

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

Dans la définition ci-dessus, l'opération binaire est représentée par `op : R → R → R`. Le type `I` est celui des indices. La fonction d'indexation est représenté par `F`. L'ensemble des indices est représenté par les éléments de la liste `r : seq I` qui respectent le prédicat `P`. La définition de `reducebig` utilise l'opérateur `foldr` pour définir une récursion structurelle sur la liste `r`. Elle parcourt la liste `r` et teste si le prédicat `P` est satisfait pour décider si la valeur de `F` en cet élément va être combinée suivant `op` à la valeur déjà calculée ; à la fin de la liste `idx` est ajoutée. Si $\{p_1, \ldots, p_n\}$ sont les éléments de la liste `r` qui satisfont le prédicat `P` et avec une notation infixe pour `op`, `reducebig` correspond schématiquement à : `(F` p_1`) op (F` p_2`) op ... op (F` p_n`) op idx.`

Au dessus de la définition de `reducebig` et pour assurer une meilleur lisibilité, des notations Coq sont définies. Elles ont la forme suivante :

`\big [ op / idx ]_ (`*`ensemble des indices`*`) F`

Dans ces notations, la partie "*ensemble des indices*" permet de déterminer le nom de la variable d'indice, dont dépend la fonction `F`, et l'ensemble des indices. Principalement trois méthodes sont fournies pour spécifier ces indices : une liste, un type fini, ou un intervalle d'entier.

- liste : la notation `(i <- r)` permet de dire que l'ensemble des indices correspond à la liste `r`. Le type de ces indices va etre inféré à partir du type de `r`.
- type fini : les notations `(i)` et `(i : t)` avec `t : finType` permettent de dire que l'ensemble des indices est un type fini. Dans le cas de la notation `(i)` le type fini des indices est inféré à partir de la fonction d'indexation `F`. Lorsque le type `t` est le type `'I_n`, la notation `(i < n)` est utilisée.
- intervalle : la notation `(m <= i < n)` permet dire que les indices sont tous les entiers entre `m` et `n`.

Pour ces notations, il est aussi possible de filtrer l'ensemble des indices avec un prédicat. Ceci est fait en ajoutant à la fin de la spécification de l'ensemble des indices le symbole "`|`" suivi d'un prédicat booléen `P` dépendant de la variable d'indice `i`. Par exemple, avec les notations ci-dessus, nous avons les correspondances suivantes :

`\big[addn/0]_(i < n) i^2` $\Leftrightarrow \sum_{0 \leq i < n} i^2$.
`\big[addn/0]_(i < n | even i) i^2` $\Leftrightarrow \sum_{0 \leq i < n \ | \ i \ pair} i^2$.

Dans les notations ci-dessus, `even` est un prédicat booléen pour dire qu'un entier naturel est pair.

Des notations supplémentaires sont aussi définies pour des opérations binaires particulières. Nous avons ainsi que la notation `\max_(i) F` correspond à une instance de `reducebig` avec l'opération binaire égale à `maxn` (le maximum binaire des entiers) et `nil` égale à 0. L'ensemble de ces notations Coq fournit une façon plus compacte et proche de la syntaxe LaTex pour représenter les opérations indexées. Dans le système, ces notations correspondent toutes à la fonction `reducebig` mais avec des arguments particuliers.

Structures pour les opérateurs

Dans la définition de la fonction `reducebig`, l'opération binaire est définie sur un type `R`. Par défaut le type `R`, l'opération binaire `op` et l'élément `nil` ne possèdent pas

```
Variables (T : Type) (e : T).
Structure law : Type := Law {
 op :> T → T → T;
 _ : forall x y z, op x (op y z) = op (op x y) z;
 _ : forall x, op e x = x; _ : forall x, op x e = x
}.
Structure com_law : Type := ComLaw {
 com_op :> law; _ : forall x y, op x y = op y x
}.
```

FIGURE 2.4 – Définitions des interfaces de monoïde et de monoïde commutatif.

de structure particulière. Ceci permet d'avoir une définition générique. Par contre, la preuve d'un bon nombre de résultats sur les opérations indexées nécessite de munir l'opération binaire d'une structure particulière. Dans la pratique, l'utilisation des opérations indexées est plus naturel lorsque l'opération `op` est associative et `nil` est son élément neutre. En d'autre termes `(R, op, nil)` défini une structure de monoïde. Les propriétés associées à cette structure sont nécessaires pour prouver un grand nombre des résultats sur les opérations indexées. Ces résultats peuvent être appliqués à des opérations binaires variées comme l'addition des entiers ou la somme de sous-espaces.

Pour prouver des résultats génériques sur les opérations indexées pour lesquels une structure de monoïde sur l'opération binaire est nécessaire, nous pouvons utiliser l'interface de monoïde définie dans la Figure 1.1. Les lemmes associés à ces résultats seront paramétrés par des variables de type `monoid`. Lors de l'application de ces résultats à des opérations binaires spécifiques, les structures canoniques seront utilisées pour inférer automatiquement la structure de monoïde associée. Par ailleurs, comme nous l'avons vu dans la Section 1.3 avec l'exemple des entiers, la définition de la structure de `monoid` limite l'utilisation des structures canoniques. En effet, dans la structure `monoid` de la Figure 1.1 le type de représentation (projection `sort`) est un champ de l'enregistrement. Ceci implique que nous pourrons définir qu'une seule instance générique de monoïde par type de représentation. Hors pour un anneau abstrait nous avons deux opérations de monoïde : l'addition et la multiplication internes. La solution à cette limitation est de définir la structure directement sur l'opération et non pas le type de représentation. Ce dernier sera un argument du type enregistrement. La Figure 2.4 donne la définition des structures `law` et `com_law` qui représentent respectivement le type des opérations de monoïdes et celui des opérations de monoïdes commutatifs. Avec ces définitions, il est possible de déclarer plus d'une structure canonique d'opération de monoïde par type de représentation. Ceci va nous permettre d'appliquer les mêmes lemmes génériques sur

les opérations indexées à des opérations binaires pas seulement sur des types différents mais aussi sur un même type comme l'addition ou la multiplication d'un anneau.

Les structures d'opérations de monoïde permettent d'enrichir la définition de l'opération binaire associée à l'opération indexée. Comme nous l'avons vu, la définition de l'opération indexée ne nécessite aucune structure particulière pour l'opération binaire. Dans la bibliothèque `bigops`, ces structures sont utilisées dans la preuve des résultats génériques. Ces preuves sont organisées suivants les structures de monoïde nécessaires sur l'opération binaire. Par exemple, le lemme suivant ne nécessite aucune condition sur l'opération binaire :

```
Lemma eq_bigl : forall r (P1 P2 : pred I) F, P1 =1 P2 ->
 \big[op/idx]_(i <- r | P1 i) F i = \big[op/idx]_(i <- r | P2 i) F i.
```

Le lemme `eq_bigl` permet de dire que deux opérations indexées sont égales lorsque leur ensemble d'indices sont extentionnellement égaux. Ceci ne requiert aucune structure particulière sur l'opération binaire `op`. D'autres résultats nécessitent que l'opération binaire soit munie d'une structure de monoïde. Par exemple, en notant `*%M` la loi d'un monoïde donné et `1` l'élément neutre associé, nous pouvons prouver un lemme générique pour décomposer une opération indexée suivant la concaténation (notation `++`) de deux listes :

```
Lemma big_cat : forall I r1 r2 (P : pred I) F,
 \big[*%M/1]_(i <- r1 ++ r2 | P i) F i =
 \big[*%M/1]_(i <- r1 | P i) F i * \big[*%M/1]_(i <- r2 | P i) F i.
```

Le lemme `big_cat` correspond en notation mathématique standard à la formule suivante :

$$\prod_{i \in r_1 \cup r_2, P_i} F_i = \prod_{i \in r_1, P_i} F_i * \prod_{i \in r_2, P_i} F_i.$$

Les deux lemmes ci-dessus et les autres lemmes de la bibliothèque `bigops` prouvent des résultats génériques sur les opérations indexées. Ces résultats peuvent être appliqués à toute instance d'opération indexée dont l'opération binaire respectent les axiomes de la structure associée. Les structures canoniques permettent de rendre automatique l'inférence de ces structures lors de l'application de ces résultats génériques. Ainsi nous allons pouvoir appliquer ces lemmes à des opérations binaires différentes comme l'addition interne dans un espace vectoriel ou dans un anneau ou la somme de sous-espaces. Ce sont des opérations que nous aurons à manipuler souvent dans notre développement sur l'algèbre linéaire.

2.4 Conclusion

SSReflect étend le langage de spécification de Coq et son langage de tactique. L'objectif avec l'extension de *Gallina* est d'avoir une syntaxe proche de celle du langage ML. Le nouveau langage de tactique de SSReflect permet de réduire la taille des scripts de preuve et de mieux gérer l'organisation de ces scripts.

Avec les extensions de Coq, SSReflect définit aussi des nouvelles bibliothèques de preuves formelles. Ces bibliothèques sont adaptées à la réflexion à petite échelle. En particulier, SSReflect définit une réflexion entre les types `bool` et `Prop` de Coq. Cette réflexion permet de voir les propositions logiques décidables à la fois comme des objets dans le type des booléens et dans le type `Prop`. Les propositions décidables vont avoir deux couches de représentations. La première va représenter le contenu de la proposition ou sa partie calculatoire. La seconde couche va correspondre à la vue de cette proposition comme un objet du type abstrait `Prop`. La vue dans `Prop` permet de profiter des mécanismes fournit par le système pour faire de la déduction naturelle. Cette méthode de conception est aussi utilisée pour définir les fonctions à support fini. Dans ce cas le contenu va correspondre au graphe de la fonction et la présentation correspondra à la vue de cette fonction comme un objet du type des fonctions standard Coq ou lambda expression. Cette méthode de séparation entre le contenu et la présentation, ou l'implémentation et l'interface, correspond à une technique standard de génie logiciel. Elle permet entre autres de faciliter la réutilisation et la maintenance du développement. Dans le cadre intuitionniste de Coq, un avantage de cette approche est de pouvoir faire des raisonnements mathématiques classiques.

Dans le contexte des mathématiques sur ordinateur l'égalité et le quotient sont des opérations importantes. Dans Coq, les *setoids* fournissent une méthode pour définir ces deux notions. Néanmoins l'utilisation des *setoids*, surtout pour la réécriture, est moins pratique que l'égalité standard de Coq. Par ailleurs, cette égalité qui est appelée égalité de Leibniz impose des conditions syntaxiques fortes et ne permet pas d'exprimer des égalités mathématiques qui correspondent à des relations d'équivalences. SSReflect définit une notion d'égalité décidable qui fournit un compromis entre la facilité d'utilisation de l'égalité de Leibniz et la généralité des *setoids*. En plus de l'égalité décidable, une interface est définie pour modéliser les types munis d'un opérateur de choix. Cette interface fournit une méthode générique pour définir des types quotients sur les quelles la relation d'équivalence va être équivalente à l'égalité de Leibniz.

L'idée générale dans SSReflect est de développer des composants mathématiques en profitant de l'expressivité de la logique d'ordre supérieure de Coq et en tenant

compte de son cadre logique constructive. Les propositions ou structures décidables seront représentées par des objets qui expriment cette propriété de décidabilité. Au dessus de ces objets une couche de présentation ou des interfaces vont permettre de les voir comme des objets de base du système : des `Type`, `Prop` ou fonctions. Les *coercions* et les structures canoniques de Coq jouent un rôle important dans cette approche. Elles permettent de rendre ce changement point de vue transparent pour l'utilisateur. Elles fournissent aussi des moyens pour assurer la généricité du développement. Dans notre travail de formalisation et plus généralement dans le projet *Mathematical Components*, nous allons appliquer la même approche utilisée avec les propositions décidables et les fonctions finies à des structures mathématiques plus complexes, comme les polynômes et les représentations de groupes finis.

Chapitre 3

Cayley-Hamilton : polynômes et matrices

3.1 Introduction

Le théorème de Cayley-Hamilton, qui porte le nom des mathématiciens Arthur Cayley(1821-1895) et William Hamilton(1805-1865), est un résultat important de l'algèbre linéaire. Il a été énoncé pour la première fois par Arthur Cayley sous la forme suivante :

> *Toute matrice satisfait une équation algébrique qui a le même degré que son ordre*[1].

Les matrices dont il est question dans cet énoncé sont les matrices carrées. L'équation algébrique mentionnée est en fait le polynôme caractéristique de la matrice.

Arthur Cayley est considéré avec Sylvester comme l'un des pères fondateurs de la théorie des matrices modernes. Sylvester fut le premier à utiliser le terme matrice pour designer un tableau bidimensionnel de coefficients. Cayley posa les fondations de la théorie des matrices modernes dans son célèbre ouvrage *Memoir on the Theory of Matrices* qu'il publia en 1858. Le théorème de Cayley-Hamilton fut pour la première fois énoncé dans cet ouvrage et prouvé seulement pour des matrices d'ordre 2 et 3. Il a fallu attendre 1878 pour que Georg Frobenius donne la première preuve complète du théorème. La raison pour laquelle Hamilton figure dans le nom du théorème est qu'il en a donné une preuve dans ses travaux sur la théorie des quaternions. En effet, Hamilton,

1. L'énoncé original est en fait le suivant : *"... any matrix whatever satisfies an algebraic equation of its own order ..."* [31]

indépendamment des travaux de Cayley, a prouvé le résultat du théorème de Cayley-Hamilton pour les opérations linéaires entre espaces de quaternions. Ces opérations peuvent être représentées par des matrices d'ordre 4. Dans ses travaux Hamilton n'utilisa pas de notations spécifiques pour les matrices. C'est Cayley qui introduisit le premier une notation spécifique pour les matrices et les considéra comme entité algébrique [31].

Le théorème de Cayley-Hamilton est utilisé pour faire des calculs sur les matrices carrées (ou les endomorphismes) : calcul de la matrice inverse ou calcul des valeurs propres. Un corollaire de ce théorème est le résultat selon lequel le polynôme minimal d'une matrice donnée est un diviseur de son polynôme caractéristique. C'est un théorème dont nous allons avoir besoin dans les travaux sur la théorie des caractères. Cette dernière fait appel à de nombreux résultats sur les matrices ou les endomorphismes d'espaces vectoriels. Les composantes de la preuve, à savoir les polynômes et les matrices, sont présents à différents niveaux dans la preuve du théorème de Feit-Thompson. La formalisation du théorème de Cayley-Hamilton est une bonne plateforme de test des bibliothèques qui implémentent ces deux théories. Plus anecdotiquement, ce théorème fait partie de la liste des 100 théorèmes à formaliser avec des assistants à la preuve [1]. Dans ce travail nous présentons la première formalisation de ce théorème dans un assistant à la preuve.

Le travail présenté dans ce chapitre a déjà été décrit sous une forme moins récente dans [38] et partiellement dans [7]. En premier lieu, nous expliquons la preuve mathématique du théorème de Cayley-Hamilton. Nous allons par la suite détailler les différentes composantes nécessaires pour aboutir à la formalisation de la preuve : la théorie des anneaux, les polynômes et les matrices. Enfin, nous présentons la formalisation proprement dite du théorème avant de conclure.

3.2 Le théorème

3.2.1 Définitions

Avant d'énoncer le théorème de Cayley-Hamilton nous allons rappeler les définitions de quelques opérations sur les matrices. Ces définitions sont nécessaires pour comprendre l'énoncé et la preuve du théorème. Dans la suite de cette section les matrices manipulées sont par défaut des matrices carrées sur un anneau R pas nécessairement commutatif. L'ensemble des matrices carrées d'ordre n et à coefficients dans R est noté

1. Liste maintenue par F. Wiedijk et disponible à l'adresse : http://www.cs.ru.nl/freek/100/

$M_n(R)$. L'ensemble des polynômes formels à coefficients dans R est noté $R[X]$. Ces deux ensembles possèdent une structure d'anneau.

Polynôme caractéristique : soit $A \in M_n(R)$ une matrice carrée, le polynôme caractéristique de A est défini par

$$p_A(x) = \det(xI_n - A). \tag{3.1}$$

En notant $A = (a_{i,j})_{0 \leq i,j < n}$, la matrice $xI_n - A$ s'écrit sous la forme :

$$\begin{pmatrix} x - a_{0,0} & -a_{0,1} & \cdots & -a_{0,n-1} \\ -a_{1,0} & x - a_{1,1} & \cdots & -a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ -a_{n-1,0} & -a_{n-1,1} & \cdots & x - a_{n-1,n-1} \end{pmatrix}.$$

C'est une matrice de polynômes et donc $p_A(x) \in R[X]$. Par exemple, étant donné la matrice

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 2 & 0 \\ 0 & 1 & 1 \end{pmatrix}, \tag{3.2}$$

le polynôme caractéristique de A est donné par :

$$p_A(x) = \begin{vmatrix} x-1 & 0 & 0 \\ -2 & x-2 & 0 \\ 0 & -1 & x-1 \end{vmatrix} = x^3 - 4x^2 + 5x - 2.$$

La forme générale du polynôme caractéristique peut être obtenue en considérant la formule de Leibniz. Selon cette formule, le déterminant d'une matrice ($B = (b_{i,j}) \in M_n(R)$) est donné par :

$$\det(B) = \sum_{\sigma \in S_n} \epsilon(\sigma) \prod_{i=1}^{n} b_{i,\sigma(i)}, \tag{3.3}$$

où S_n est l'ensemble des permutations de $\{1, 2, \cdots, n\}$. Cette formule correspond à une sommation sur S_n de produits des coefficients de B. Pour une permutation $\sigma \in S_n$, $\epsilon(\sigma)$ désigne sa signature : égale à 1 si elle est paire et -1 sinon. La formule de Leibniz n'est certes pas la plus efficace en terme de calcul du déterminant mais elle permet d'établir bon nombre de résultats théoriques importants.

En appliquant cette formule du déterminant à $xI_n - A$, nous remarquons que $p_A(x)$ est la somme de $n!$ termes. L'un des ces termes est le produit $(x - a_{0,0})(x - a_{1,1}) \cdots (x - a_{n-1,n-1})$ des éléments diagonaux de la matrice considérée. Les autres termes sont des produits de n coefficients de la matrice $xI_n - A$ mais qui contiennent au plus $n - 2$ éléments de sa diagonale. Il en découle que :

$$p_A(x) = x^n - (a_{0,0} + a_{1,1} + \cdots + a_{n-1,n-1})x^{n-1} + \ldots. \tag{3.4}$$

Nous pouvons donc écrire que

$$p_A(x) = \sum_{i=0}^{n} c_i(A)x^n; \tag{3.5}$$

avec les $c_i(A) \in R$ des fonctions polynomiales des coefficients $a_{i,j}$ de A et à coefficient dans $\mathbb{Z}$. Par exemple et d'après 3.4, nous avons que :

$$c_{n-1}(A) = a_{0,0} + a_{1,1} + \cdots + a_{n-1,n-1},$$

la somme des coefficients diagonaux de A : la trace de A notée $Tr(A)$.

Cofacteur : soit $A = (a_{i,j}) \in M_n(R)$, le cofacteur d'indices i et j de A est noté $Cof(A)_{i,j}$ et est égal au déterminant de la matrice obtenu en supprimant de A la ligne i et la colonne j. Il est donné par la formule :

$$Cof(A)_{i,j} = (-1)^{i+j} \begin{vmatrix} a_{0,0} & \cdots & a_{0,j-1} & a_{0,j+1} & \cdots & a_{0,n-1} \\ \vdots & & \vdots & \vdots & & \vdots \\ a_{i-1,0} & \cdots & a_{i-1,j-1} & a_{i-1,j+1} & \cdots & a_{i-1,n-1} \\ a_{i+1,0} & \cdots & a_{i+1,j-1} & a_{i+1,j+1} & \cdots & a_{i+1,n-1} \\ \vdots & & \vdots & \vdots & & \vdots \\ a_{n-1,0} & \cdots & a_{n-1,j-1} & a_{n-1,j+1} & \cdots & a_{n-1,n-1} \end{vmatrix}.$$

Le cofacteur est défini à partir du déterminant. Ce dernier peut aussi être directement calculer en partant des cofacteurs. Ceci est donné par la formule de Laplace qui définit le développement du déterminant par rapport à une colonne j :

$$\det(A) = \sum_{i=1}^{n} a_{i,j}\, Cof(A)_{i,j}, \tag{3.6}$$

ou une ligne i :

$$\det(A) = \sum_{j=1}^{n} a_{i,j}\, Cof(A)_{i,j}. \tag{3.7}$$

La formule de Laplace permet ainsi de ramener le calcul d'un déterminant de taille n au calcul de n déterminants de taille $n-1$. Même si ceci n'est pas toujours pratique d'un point de vue calculatoire, elle permet d'établir des résultats théoriques importants comme la règle de Cramer que nous décrirons dans la suite.

Comatrice : soit $A \in M_n(R)$, la comatrice de A est notée $com(A)$ et correspond à la matrice des cofacteurs de A. Elle est donnée par la formule :

$$com(A) = (Cof(A)_{i,j})_{0 \leq i,j < n}.$$

Cette matrice est toujours dans $M_n(R)$. Cette définition suppose que la matrice est définie sur un anneau R non commutatif mais la majorité des résultats intéressants qui en découlent, comme la règle de Cramer, demandent que l'anneau soit commutatif. En effet, soient R un anneau commutatif et $B \in M_n(R)$, la règle de Cramer affirme que :

$$B * {}^t\mathrm{com}B = {}^t\mathrm{com}B * B = \det B * I_n \tag{3.8}$$

Autrement dit, B et la transposée de sa comatrice commutent et ce produit est égal au déterminant de B multiplié par la matrice identité.

Théorème de Cayley-Hamilton

soient R un anneau commutatif et $A \in M_n(R)$, alors $p_A(A) = O_n$.

Dans cet énoncé O_n représente la matrice nulle de $M_n(R)$. L'énoncé donné ci-dessus est le plus général. Dans la littérature mathématique, le théorème est souvent considéré dans le cas des endomorphismes d'espace vectoriel. Dans ce cas, R n'est plus un anneau commutatif mais un corps.

Un exemple : en considérant le polynôme caractéristique de la matrice A de 3.2, le théorème de Cayley-Hamilton affirme que :

$$p_A(A) = A^3 - 4A^2 + 5A - 2I_3 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

En effet, en remplaçant A par sa valeur nous obtenons que :

$$\begin{aligned} p_A(A) &= \begin{pmatrix} 1 & 0 & 0 \\ 2 & 2 & 0 \\ 0 & 1 & 1 \end{pmatrix}^3 - 4\begin{pmatrix} 1 & 0 & 0 \\ 2 & 2 & 0 \\ 0 & 1 & 1 \end{pmatrix}^2 + 5\begin{pmatrix} 1 & 0 & 0 \\ 2 & 2 & 0 \\ 0 & 1 & 1 \end{pmatrix} - 2\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 0 \\ 14 & 8 & 0 \\ 8 & 7 & 1 \end{pmatrix} - \begin{pmatrix} 4 & 0 & 0 \\ 24 & 16 & 0 \\ 8 & 12 & 4 \end{pmatrix} + \begin{pmatrix} 5 & 0 & 0 \\ 10 & 10 & 0 \\ 0 & 5 & 5 \end{pmatrix} - \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}. \end{aligned}$$

Cet exemple montre que le théorème de Cayley-Hamilton est valide pour la matrice A. Nous donnons une preuve algébrique générale dans la section suivante.

3.2.2 Preuve

Une première approche

La preuve du théorème de Cayley-Hamilton peut sembler immédiate. En effet, on peut être tenté dans la définition du polynôme caractéristique $p_A(x) = \det(xI_n - A)$ de remplacer x par A. Nous obtenons donc que :

$$p_A(A) = \det(AI_n - A) = \det(A - A) = \det 0 = 0 \tag{3.9}$$

Le problème avec cette "preuve" est que dans la définition du polynôme caractéristique de A, l'opération de multiplication que nous avons dans xI_n est la multiplication d'un scalaire par une matrice et non l'opération de multiplication interne des matrices. Ainsi dans la formule 3.9, AI_n et A doivent être interprétées dans $M_n(M_n(R))$. Elles correspondent respectivement à

$$\begin{pmatrix} A & 0 & \cdots & 0 \\ 0 & A & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A \end{pmatrix}$$

et

$$\begin{pmatrix} a_{0,0}I_n & a_{0,1}I_n & \cdots & a_{0,n-1}I_n \\ a_{1,0}I_n & a_{1,1}I_n & \cdots & a_{1,n-1}I_n \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0}I_n & a_{n-1,1}I_n & \cdots & a_{n-1,n-1}I_n \end{pmatrix}.$$

Il en découle que les matrices AI_n et A ne sont pas égales et que donc $AI_n - A$ n'est pas nulle.

Cette fausse preuve montre que les notations mathématiques peuvent facilement induire le lecteur en erreur. Certes ces notations permettent d'améliorer et simplifier la présentation des concepts mathématiques mais elles doivent être manipulées avec prudence.

Une preuve algébrique [1]

Le théorème de Cayley-Hamilton peut être prouvé en partant de la règle de Cramer. Soit la matrice $xI_n - A \in M_n(R[X])$ qui est une matrice de polynômes. L'anneau R étant commutatif, $R[X]$ l'est aussi. Nous pouvons donc appliquer 3.8 à $xI_n - A$. Nous obtenons alors que :

$${}^t\mathrm{com}(xI_n - A) * (xI_n - A) = \det(xI_n - A)I_n = p_A(x)I_n. \tag{3.10}$$

1. La preuve décrite dans la suite s'inspire des preuves présentées dans [18, 29, 33].

L'égalité 3.10 ci-dessus est une égalité dans $M_n(R[X])$ (matrice de polynômes). Cet anneau est isomorphe à l'anneau $M_n(R)[X]$ des polynômes à coefficients matriciels. 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 élément de $M_n(R)[X]$. Par exemple :

$$\begin{pmatrix} x^2+2 & 2x^2+x \\ -x & 2x+1 \end{pmatrix} = x^2 \begin{pmatrix} 1 & 2 \\ 0 & 0 \end{pmatrix} + x \begin{pmatrix} 0 & 1 \\ -1 & 2 \end{pmatrix} + \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}. \tag{3.11}$$

En notant $\phi : M_n(R[X]) \to M_n(R)[X]$ le morphisme qui étant donné une matrice de polynômes retourne un polynôme de matrices suivant l'exemple ci-dessus, l'égalité 3.10 s'écrit dans $M_n(R)[X]$ sous la forme :

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

Les propriétés de morphismes de ϕ permettent de déduire que :

$$\phi({}^t\mathrm{com}(xI_n - A)) * (x - A) = \phi(p_A(x)I_n). \tag{3.12}$$

Nous obtenons donc que $(x - A)$ est facteur de $\phi(p_A(x)I_n)$ dans $M_n(R)[X]$. Il en découle que A est racine de $\phi(p_A(x)I_n)$ qui, rappelons-le, est un polynôme à coefficients matriciels ($\in M_n(R)[X]$). En considérant la forme générale du polynôme caractéristique de 3.5 et par définition de ϕ, $\phi(p_A(x)I_n)$ s'écrit formellement sous la forme :

$$\phi(p_A(x)I_n) = \sum_{i=0}^{n} c_i(A)I_n * x^i \tag{3.13}$$

avec les $c_i(A) \in R$. Nous pouvons alors écrire que :

$$\begin{aligned} p_A(A) &= \sum_{i=0}^{n} c_i(A)A^i = \sum_{i=0}^{n} c_i(A)I_n * A^i \\ &= O_n \quad \text{puisque } A \text{ est racine de } \phi(p_A(x)I_n) \ \blacksquare \end{aligned}$$

3.2.3 Remarques sur la preuve

Polynômes et fonctions polynomiales

Dans la preuve du théorème de Cayley-Hamilton, il est question de remplacer l'inconnu x dans la formule de $p_A(x)$ par A. Ceci correspond à considérer les polynômes comme des fonctions. La question qui se pose alors est : pourquoi ne pas voir les polynômes simplement comme un cas spécial de fonctions ? Il est clair que les polynômes définissent des fonctions, mais dans certains cas ces polynômes ne peuvent pas directement être identifiés avec leur fonction associée. Par exemple dans le cas des polynômes

sur un corps fini, des polynômes différents peuvent être associés à la même fonction. Ainsi, soit le polynôme $f(x) = x^3 - x \in \mathbb{Z}/\mathbb{Z}_3[X]$ à coefficients dans l'anneau des entiers modulo 3. Ce polynôme est différent du polynôme nul mais la fonction qui lui est associée est nulle sur $\mathbb{Z}/\mathbb{Z}_3[X]$. En effet :

$$f(\bar{0}) = \bar{0}^3 - \bar{0} = \bar{0}$$
$$f(\bar{1}) = \bar{1}^3 - \bar{1} = \bar{0}$$
$$f(\bar{2}) = \bar{2}^3 - \bar{2} = \bar{0}$$

Par ailleurs plusieurs opérations sur les polynômes, comme la multiplication ou la division euclidienne, n'ont pas besoin de tenir compte de la valeur du polynôme en un point donné. Ces opérations sont définies en fonction des coefficients du polynôme et de la façon dont ils sont organisés. Ceci fait qu'en mathématiques et principalement en algèbre les polynômes sont souvent considérés comme distincts des fonctions polynomiales. Le x présent dans la définition d'un polynôme ne doit pas donc être interprété comme une variable mais plutôt comme un polynôme particulier : celui dont tous les coefficients sont nuls sauf celui d'indice 1.

Morphisme d'évaluation

La distinction entre polynômes et fonctions polynomiales n'empêche pas la définition d'une correspondance entre ces deux notions. En effet, si R est un anneau commutatif et $p(x) \in R[X]$ un polynôme, nous pouvons associer à p une fonction $f : R \to R$. Pour un élément $a \in R$, la valeur de $f(a)$ est obtenue en remplaçant dans la formule de p les x par a. La fonction $ev_a : p \mapsto p(a)$ est un morphisme d'anneau de $R[X]$ dans R. Elle est souvent appelée le *morphisme d'évaluation*. Parmi les propriétés de ce morphisme, nous avons que :

$$\forall p, q \in R[X], ev_a(p + q) = ev_a(p) + ev_a(q) \tag{3.14}$$
$$\forall p, q \in R[X], ev_a(p * q) = ev_a(p) * ev_a(q) \tag{3.15}$$

Même si l'évaluation des polynômes est effectuée généralement sur des structures d'anneaux commutatifs, on peut parfois effectuer l'évaluation en un élément appartenant à une algèbre associative sur R. Ceci est le cas dans le théorème de Cayley-Hamilton où nous évaluons un polynôme de $R[X]$ en une matrice car $M_n(R)$ est une R-algèbre. La fonction d'évaluation garde toujours ses propriétés de morphisme. Elle sera alors un morphisme d'anneau de $R[X]$ dans $\mathbb{A}$, avec $\mathbb{A}$ une R-algèbre associative et qui en plus est un anneau.

Il est aussi possible d'être plus général et de supposer que l'anneau R n'est pas commutatif. La définition de la fonction d'évaluation reste la même. Le passage d'un anneau commutatif vers un anneau non commutatif n'influe pas sur la définition de l'évaluation. Mais la propriété 3.15 sur la distributivité de l'évaluation par rapport à la multiplication de polynômes n'est plus valide. La fonction d'évaluation n'est plus alors un morphisme. En effet, dans un anneau commutatif, 3.15 découle de la propriété de commutativité de la multiplication des éléments de l'anneau. Dans un anneau non commutatif, il est alors nécessaire d'avoir une condition sur les coefficients des polynômes du produit et la valeur pour laquelle s'effectue l'évaluation. Dans la littérature mathématique, il est rarement question d'évaluer des polynômes sur des anneaux non commutatifs. Quand elle apparait, cette condition s'exprime sous la forme suivante [33] :

$$\begin{array}{c}\textbf{pour que } (p*q)(a) = p(a)*q(a) \textbf{ avec } a \in R \textbf{ et } p,q \in R[X] \\ \textbf{il faut que } a \textbf{ commute avec tous les coefficients du polynôme } q\textbf{.}\end{array} \tag{3.16}$$

Nous reviendrons plus en détail sur cette condition dans la section sur la formalisation et montrerons qu'elle peut être remplacée par une condition plus faible.

A la lumière de la définition de l'évaluation des polynômes, le développement de la preuve à partir de l'égalité 3.12 peut être décrit de façon plus explicite. En effet, en partant de 3.12 et en appliquant la fonction d'évaluation nous obtenons que :

$$ev_A(\phi({}^t\mathrm{com}(xI_n - A)) * (x - A)) = ev_A(\phi(p_A(x)I_n)) \tag{3.17}$$

$$ev_A(\phi({}^t\mathrm{com}(xI_n - A))) * ev_A(x - A) = \tag{3.18}$$

Dans les formules ci-dessus, même si $M_n(R)$ n'est pas un anneau commutatif, le passage de l'évaluation du produit (3.17) au produit des évaluations (3.18) est permis. En effet, la matrice A commute avec les coefficients du polynômes $(x - A)$: A commute avec elle même et la matrice identité I_n. En partant de la formule 3.13 et de 3.18, le théorème de Cayley-Hamilton est alors établi :

$$p_A(A) = ev_A(\phi(p_A(x)I_n)) = ev_A(\phi({}^t\mathrm{com}(xI_n - A))) * (A - A) = 0$$

Preuve sur $\mathbb{Z}$

Une propriété intéressante du théorème de Cayley-Hamilton est qu'il suffit de le prouver pour les matrices sur l'anneau $\mathbb{Z}$. Nous reviendrons dans la section sur la formalisation à cette propriété. La preuve du théorème de Cayley-Hamilton décrite précédemment a été établie pour n'importe quelle matrice carrée à coefficients dans un anneau commutatif R quelconque. En pratique cet anneau peut être remplacé par $\mathbb{Z}$.

En effet et comme décrit dans 3.5, la forme générale du polynôme caractéristique d'une matrice $A \in M_n(R)$ est $p_A(x) = \sum_{i=0}^{n} c_i(A)x^n$. Les coefficients $c_i(A)$ sont des fonctions polynomiales des coefficients $a_{i,j}$ de A. Ce sont des polynômes à plusieurs indéterminées et à coefficients dans $\mathbb{Z}$. La forme de ces polynômes est indépendante de la matrice A et de l'anneau R. Par exemple la formule pour calculer le coefficient du monôme d'ordre $n-1$ est :

$$c_{n-1}(A) = a_{0,0} + a_{1,1} + \cdots + a_{n-1,n-1}.$$

Cette expression, vue comme un polynôme à plusieurs indéterminées et à coefficients dans $\mathbb{Z}$, est la même pour toutes les matrices d'ordre n d'un anneau R quelconque. En considérant la forme des coefficients $c_i(A)$ de $p_A(x)$ et la valeur de :

$$p_A(A) = \sum_{i=0}^{n} c_i(A)A^n,$$

nous remarquons qu'il existe n^2 polynômes $(q_{i,j})_{0 \leq i,j < n}$ à n^2 indéterminées et à coefficients dans $\mathbb{Z}$ (les $q_{i,j} \in \mathbb{Z}[X_1, \cdots, X_{n^2}]$), tels que, pour tout anneau commutatif R et toute matrice $A = (a_{i,j}) \in M_n(R)$, les coefficients de $p_A(A)$ sont obtenus en évaluant les polynômes $q_{i,j}$ en les coefficients $a_{i,j}$ de A. Il en découle que pour montrer que $p_A(A) = 0$, il suffit de montrer que les $q_{i,j}$ sont nuls. Ces derniers sont des polynômes à plusieurs indéterminées et à coefficients dans $\mathbb{Z}$. Pour montrer que les $q_{i,j} \in \mathbb{Z}[X_1, \cdots, X_{n^2}]$ sont nuls, il faut montrer qu'à chaque fois que les indéterminées sont remplacées par des éléments quelconques de $\mathbb{Z}$ le résultat est nul. Par construction des $q_{i,j}$, ceci revient à montrer que $p_A(A) = 0$ pour toute matrice A à coefficients dans $\mathbb{Z}$. Ainsi, pour montrer le théorème de Cayley-Hamilton pour une matrice de $M_n(R)$, avec R un anneau commutatif quelconque, il suffit de le montrer pour une matrice quelconque de $M_n(\mathbb{Z})$.

3.3 La formalisation

Formaliser le théorème de Cayley-Hamilton est une étape importante dans le long processus de formalisation du théorème de Feit-Thompson. Même si ce théorème est un des premiers résultats d'un cours d'algèbre linéaire, sa formalisation fait intervenir deux structures mathématiques indépendantes et importantes : les polynômes et les matrices. Ce théorème servira donc de plateforme de test pour s'assurer que nos bibliothèques pour manipuler les polynômes et les matrices ont été développées avec un degré de généricité suffisant. La formalisation de ce théorème permettra aussi d'enrichir la base de donnée des mathématiques formalisées.

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 objectif deux difficultés sont à surmonter. En premier 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 partie la plus difficile du travail de formalisation. Le deuxième problème qui se pose au moment de la formalisation est celui de la représentation des données et structures mathématiques. Celles-ci peuvent être présentées et définies de différentes manières. Certaines représentations sont pratiques pour faire du calcul, mais moins pour faire des preuves et vice versa. Le choix entre ces différentes méthodes est une question à laquelle il faut répondre au niveau de l'implémentation.

Par ailleurs, et comme dans tout travail de développement logiciel, il faut surmonter les problèmes de conceptions et d'organisation du code. Ceci est d'autant plus important lorsqu'il est question de réutiliser ces bibliothèques de preuves dans d'autres développements. L'intérêt n'est pas simplement de prouver tel ou tel théorème mais de le faire d'une manière qui facilitera la réutilisation de ce travail dans d'autres développements.

3.3.1 Composantes de la preuve

La preuve du théorème de Cayley-Hamilton présentée dans 3.2.2, montre que pour formaliser ce théorème il est nécessaire de développer des composants pour les théories suivantes :

- **polynômes** : la théorie des polynômes est une partie nécessaire et importante de la preuve. Le théorème de Cayley-Hamilton parle de polynôme caractéristique et d'évaluation de polynômes. Un pré-requis à la formalisation du théorème est le développement de cette théorie dans l'assistant à la preuve.
- **matrices** : le théorème de Cayley-Hamilton est principalement un résultat sur les matrices carrées. La preuve que nous avons présentée précédemment découle d'un résultat sur les matrices : la règle de Cramer. Une bibliothèque de preuves formelles sur les matrices est aussi un pré-requis pour la formalisation du théorème.
- **anneaux** : dans l'énoncé du théorème de Cayley-Hamilton il est question de matrice sur un anneau commutatif et de polynômes de matrices, en remarquant que l'anneau des matrices n'est pas commutatif. Les polynômes et les matrices partagent des propriétés algébriques d'anneaux et ces propriétés sont souvent utilisées dans la preuve.

Ces trois théories peuvent être considérées comme les composantes externes de la preuve. Ce sont des théories indépendantes du théorème de Cayley-Hamilton et qui ont des applications vastes et variées dans différents domaines des mathématiques. La preuve proprement dite du théorème de Cayley-Hamilton correspond à la définition du morphisme décrit dans 3.11 et à l'application de certains résultats sur les polynômes et les matrices comme le théorème du reste et la règle de Cramer. Dans la suite nous présentons le développement qui à été nécessaire pour arriver à la formalisation du théorème de Cayley-Hamilton. Ce développement a été fait en se basant sur la bibliothèque sur les matrices développée principalement par Georges Gonthier dans le cadre du projet *Mathematical Components*.

3.3.2 Interfaces pour les anneaux

En mathématiques standards les polynômes et les matrices sont définis sur des ensembles possédant des structures d'anneaux. D'un point de vue génie logiciel et pour pouvoir formaliser une théorie des polynômes, celle-ci doit avoir comme argument un anneau. Cet argument correspond à un paramètre de la théorie. Par ailleurs les polynômes et les matrices possèdent elles aussi une structure d'anneau. Elle partage ainsi certaines propriétés algébriques comme la commutativité de l'addition. Une théorie générique des anneaux permettra de factoriser les propriétés algébriques d'anneaux que ces deux structures mathématiques partagent entre elles et avec d'autres structures.

Le système Coq fournit une tactique appelée `ring` [11, 23] pour résoudre des équations polynomiales sur un anneau abstrait ou concret. Un tel anneau doit être déclaré en utilisant la commande `Add Ring`. Mais cette commande impose que l'anneau soit commutatif, ce qui n'est pas toujours le cas dans les structures qui nous intéressent, en particulier l'anneau des matrices.

En algèbre, un anneau est défini par la donnée d'un ensemble A, d'éléments 0 et $1 \in A$ et d'opérations internes $+, -, *$ dans A qui respectent les conditions suivantes :

- la loi $+$ est associative et commutative,
- 0 est l'élément neutre de $+$,
- chaque élément $x \in A$ possède un opposé, donné par $-x$,
- la loi $*$ est associative et distributive par rapport à $+$.

L'anneau est dit commutatif lorsque la loi $*$ est commutative.

Les types `Structure` de Coq permettent de regrouper tous ces pré-requis dans une même entité et d'avoir ainsi un type pour les anneaux. La définition du type `Structure` d'anneau est donnée dans la Figure 3.1. Dans cette définition, le mécanisme de *telescope*

```
Structure ringType : Type := RingType {
 sort :> eqType;
 zero : sort; one : sort;
 add : sort → sort → sort; opp : sort → sort; mul : sort → sort → sort;
 addrA : forall x y z, add x (add y z) = add (add x y) z;
 addrC : forall x y, add x y = add y x;
 add0r : forall x, add zero x = x; addr0 : forall x, add x (opp x) = zero;
 mulrA : forall x y z, mul x (mul y z) = mul (mul x y) z;
 mulr1 : forall x, mul x one = x; mul1r : forall x, mul one x = x;
 mul_addr : forall x y z, mul x (add y z) = add (mul x y) (mul x z);
 mul_addl : forall x y z, mul (add x y) z = add (mul x z) (mul y z);
 one_neq0 : one != zero
}.
...
Notation "0" := (@zero _) : ring_scope.
Notation "1" := (@one _) : ring_scope.
Notation "- x" := (@opp _ x) : ring_scope.
Notation "x1 + x2" := (@add _ x1 x2): ring_scope.
Notation "x1 * x2" := (@mul _ x1 x2): ring_scope.
```

FIGURE 3.1 – Définition de l'interface d'anneau.

avec la déclaration `sort :> eqType` définit `sort` comme une *coercion* de `ringType` vers `eqType`. Ce dernier et comme nous l'avons vu précédemment est muni d'une *coercion* vers `Type`. Par transitivité, `ringType` est muni aussi d'une *coercion* vers `Type`. Ainsi pour un anneau `A : ringType`, lorsque nous écrivons `x : A`, ceci est interprété par Coq comme `x : sort A`.

Dans la liste des axiomes donnée dans la Figure 3.1, le dernier axiome stipule que l'élément neutre de la multiplication de l'anneau est différent de celui de son addition. Ceci permet d'exclure le cas de l'anneau nul.

Après la définition du type des anneaux nous définissons des notations Coq pour les constantes (`0, 1`) et les opérations (`-, +, *`) associés à la structure d'anneau. Avec l'utilisation du mécanisme des structures canoniques, ces notations et les propriétés algébriques d'anneaux peuvent être utilisées pour n'importe quel type abstrait ou concret à condition qu'il satisfasse les axiomes d'anneau. Les exemples qui nous intéressent dans ce cadre sont spécifiquement les polynômes et les matrices.

Dans la définition du type `ringType`, le type des éléments de l'anneau est déclaré comme un `eqType` : un type muni d'une égalité décidable. En mathématique classique, ceci ne pose pas de problème puisque tous les types sont décidables. Dans un cadre constructif, comme c'est le cas par défaut dans Coq, ceci limite l'utilisation de cette

```
Structure comRingType : Type := ComRingType {
 sort :> ringType;
 mulC : forall x y : sort, x * y = y * x
}.
```

FIGURE 3.2 – Définition de l'interface pour le type des anneaux commutatifs.

interface aux types pour lesquels il est possible de définir une égalité décidable. Ceci est toujours le cas dans les types de données auxquels nous nous intéressons dans la preuve du théorème de Feit-Thompson. C'est aussi le cas avec les types de données concrets qui peuvent être représentés sur machine. Dans le cas du théorème de Cayley-Hamilton et comme nous l'avons fait remarquer dans 3.2.3 la preuve peut se ramener au cas où la matrice est à coefficients dans $\mathbb{Z}$ et où la propriété de décidabilité de l'égalité est vérifiée.

Pour pouvoir utiliser les résultats génériques sur les opérations indexées, nous déclarons des structures canoniques de monoïdes pour les opérations d'anneau. Pour un anneau quelconque, l'addition définit un monoïde commutatif et la multiplication définit elle un monoïde simple.

```
Canonical Structure add_monoid := Monoid.Law addrA add0r addr0.
Canonical Structure add_comoid := Monoid.ComLaw addrC.
Canonical Structure mul_monoid := Monoid.Law mulrA mul1r mulr1.
```

Pour plus de lisibilité des notations sont définies pour les instances de `reducebig` (voir Section 2.3.5) associées aux opérations de l'anneau. Les notations `\sum` et `\prod` représentent respectivement la somme et produit indexées (notations mathématiques $\sum$ et $\prod$) sur un anneau quelconque ou un objet de type `ringType`.

Anneaux commutatifs

L'énoncé du théorème de Cayley-Hamilton parle d'anneaux commutatifs. Certains résultats sur les polynômes ou les matrices peuvent être développés de façon générale sur un anneau non commutatif mais d'autres résultats comme la règle de Cramer ne s'applique qu'à des anneaux commutatifs. Pour cela nous définissons au-dessus du type des anneaux celui des anneaux commutatifs. Ceci est fait de la même façon que pour la définition du type des anneaux sauf que ici le type des éléments sera un `ringType` à la place de `eqType`. La définition de cette interface en utilisant les *telescopes* est donnée dans la Figure 3.2. Avec cette définition le type `comRingType` est un sous-type du type `ringType`. Il hérite ainsi de tous les lemmes et définitions fournis par le type `ringType`.

3.3.3 Matrice

Dans cette section nous présentons la bibliothèque de SSReflect qui implémentent la théorie des matrices. Cette bibliothèque fournit des résultats sur les déterminants, comme la règle de Cramer qui est le point de départ de la preuve du théorème de Cayley-Hamilton.

3.3.3.1 Définitions

Etant donné un anneau R, une matrice de $M_{m,n}(R)$ est une liste doublement indexée d'éléments de R ou en d'autres termes un tableau bidimensionnel de m lignes et n colonnes. Cette définition est équivalente à celle qui considère une matrice comme une fonction de type $\mathbb{N} \times \mathbb{N} \to R$. Cette fonction est à support fini vu que les indices sont inclus dans les intervalles $[0 \cdots n[$ et $[0 \cdots m[$. Ces intervalles correspondent respectivement aux types des entiers inférieurs strictement à m et à n.

Type et notations : en utilisant le développement sur les types finis (spécialement le type ordinal) et les fonctions à support fini, le type des matrices est défini comme un type inductif avec un seul constructeur : la fonction des indices.

```
Variables (R : ringType) (m n : nat).
Inductive matrix := Matrix of {ffun 'I_m * 'I_n → R}.
Definition mx_val A := let Matrix g := A in g.
```

Dans la définition ci-dessus, la fonction des indices est représentée par une fonction à support fini (`finfun`) du type produit de `'I_m` et `'I_n` dans `R`. Les types `'I_m` et `'I_n` étant des types finis alors leur type produit, qui se note dans Coq `'I_m * 'I_n`, est aussi fini. Ici et avec l'utilisation des structures canoniques, Coq arrive à inférer automatiquement que le type produit de deux types finis est aussi fini. La fonction `Matrix` est le constructeur par défaut du type `matrix`. Il permet de construire un élément du type `matrix` depuis un élément de type `{ffun 'I_m * 'I_n → R}`. L'utilisation de ce constructeur n'est pas toujours pratique puisque avant de l'appeler il faut déjà avoir un élément de type `finfun`. Ceci nécessite de la part de l'utilisateur de revenir à la définition du type des `finfun`, ce qui n'est pas toujours une chose pratique pour quelqu'un qui veut faire des preuves sur les matrices. L'idée est de construire une matrice à partir d'une fonction standard Coq et rendre transparent à l'utilisateur la manière dont le type `matrix` est défini. La fonction `matrix_of_fun` permet de réaliser cette tâche. Elle prend en argument une fonction de `'I_m * 'I_n` vers `R` et retourne la matrice correspondante.

```
Definition matrix_of_fun F := Matrix [ffun ij => F ij.1 ij.2].
Definition fun_of_matrix A (i : 'I_m) (j : 'I_n) := mx_val A (i, j).
Coercion fun_of_matrix : matrix >-> Funclass.
```

Par ailleurs, chaque matrice à travers ses coefficients définit une fonction à deux arguments. La fonction `fun_of_matrix` définit cette correspondance. Elle définit aussi une coercion du type des matrices vers celui des fonctions. Ainsi pour une matrice `A` donnée, `A i j` est le coefficient de la $i^{ième}$ ligne et la $j^{ième}$ colonne.

Des notations sont définies au dessus de la fonction `matrix_of_fun` pour fournir à l'utilisateur des méthodes génériques pour construire une matrice à partir d'une fonction de coefficients.

```
Notation "\matrix_ ( i < m , j < n ) E" :=
 (@matrix_of_fun _ m n (fun i j => E)).
Notation "\matrix_ ( i , j < n ) E" := (\matrix_(i < n, j < n) E).
Notation "\matrix_ ( i , j ) E" := (\matrix_(i < _, j < _) E).
```

Dans les notations ci-dessus, `E` est une expression en `i j`. Les arguments de la fonction `(fun i j => E)` doivent appartenir à un type vers lequel il y a une *coercion* depuis `'I_m` et `'I_n`, le type `nat` par exemple. La seconde notation définit les matrices carrées. Dans la troisième, la taille de la matrice va être inférée à partir de `E`. Ce qui veut dire que les arguments de `i` et `j` doivent appartenir à un type `ordinal`. Par exemple, la matrice scalaire correspondant à un élément `x` est définie comme suit :

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

Dans la suite, la notation `'M_(m, n)` va représenter le type des matrices à `m` lignes et `n` colonnes. La notation `'M_(n)` va elle correspondre au type des matrices carrées de taille `n`. La notation `x%:M` correspondra à la matrice scalaire associée à la constante `x`.

Anneau : les opérations algébriques sur les matrices sont définies en utilisant la notation `\matrix_`. L'addition et la multiplication des matrices sont définies comme suit :

```
Definition addmx (A B : 'M_(m, n)) : 'M_(m, n) :=
 \matrix_(i, j) (A i j + B i j).
Definition mulmx (A : 'M_(m, n)) (B : 'M_(n, p)) : 'M_(m, p) :=
 \matrix_(i, k) \sum_(j < n) (A i j * B j k).
```

Ces opérations sont définies pour des matrices pas forcément carrées. Seules les matrices carrées peuvent être munies d'une structures d'anneaux. Après la preuve des axiomes d'anneau correspondants, le type `matrix` qui hérite de la structure d'`eqType` du type `finfun`, est équipé d'une structure canonique de `ringType`.

```
Canonical Structure matrix_ringType n R := @RingType (matrix R n n) ...
```

Ainsi nous allons pouvoir utiliser les notations et les propriétés génériques d'anneaux et écrire par exemple :

```
Lemma mx_trace_add : forall (A B : 'M_n), \tr (A + B) = \tr A + \tr B.
```

Grâce à la déclaration d'une structure canonique d'anneau pour les matrices, nous pouvons utiliser la même opération d'addition d'anneau pour les matrices (partie gauche de l'égalité) et l'anneau des coefficients (partie droite). La notation `\tr` définit la trace d'une matrice : la somme de ses éléments diagonaux.

La déclaration de la structure d'anneau pour les matrices ouvre la voie pour la déclaration de polynômes de matrices. La correspondance entre ces derniers et les matrices de polynômes joue un rôle central dans la preuve du théorème de Cayley-Hamilton. Comme nous allons le décrire dans la section sur les polynômes, ces derniers sont définis avec comme paramètre un anneau, celui de leurs coefficients.

3.3.3.2 Déterminant

Le polynôme caractéristique dont il est question dans le théorème de Cayley-Hamilton est défini à partir du déterminant. Comme nous l'avons décrit plus haut dans la section 3.2.1, le déterminant peut être défini avec la formule de Leibniz. Cette formule permet d'établir certains résultats théoriques sur les déterminants en utilisant les propriétés des sommes et produits indexés et des permutations. Une formalisation des permutations (groupe, signature ...) a été déjà développée dans le cadre du travail sur les groupes finis [21]. Grâce à ce développement et à celui sur les opérations indexées [7], la formule s'écrit en langage Coq sous la forme suivante :

```
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 indexés sur l'anneau des coefficients de la matrice `A`. La notation `'S_n` représente le groupe des permutations sur un ensemble à `n` éléments. La notation `(-1)^+ s` quant à elle représente la puissance, ce qui suppose que le `s` est un naturel. En effet, le `s` dans

`(-1)^+ s` cache plusieurs *coercions*. La formule de parité des permutations fournit une *coercion* depuis le type des permutations vers celui des booléens (permutation impaire associés à la valeur booléenne vrai). Les booléens sont aussi munis de façon standard d'une *coercion* vers les entiers naturels : *true* correspond à 1 et *false* correspond à 0. La valeur de la signature de `s` est ainsi injectée dans R, l'anneau des coefficients, en élevant `-1` à la puissance `s`.

Ainsi et avec l'utilisation des *coercions* et des opérations indexées, le code Coq pour définir le déterminant est très proche de la formule de Leibniz usuelle de 3.3. Dans la suite, la notation `\det` représentera la fonction `determinant`.

Comatrice et formule de Cramer : à partir de la définition des déterminants, les cofacteurs et la comatrice sont construits. La bibliothèque utilise la convention selon laquelle la comatrice (en anglais *adjugate matrix*) est la transposée de la matrice des cofacteurs.

```
Definition cofactor n A (i j : 'I_n) :=
 (-1) ^+ (i + j) * \det (mx_row' i (mx_col' j A)).
Definition adjugate n A := \matrix_(i, j < n) (cofactor A j i).
```

Les fonctions `mx_col'` et `mx_row'` de la définition du cofacteur correspondent respectivement à la suppression d'une colonne ou d'une ligne de la matrice passé en second argument. Leur premier argument est l'indice de la colonne ou de la ligne à supprimer.

Avec ces définitions, les égalités de la formule 3.8 de Cramer sont représentées formellement par les lemmes `mulmx_adjr` et `mulmx_adjl`.

```
Lemma mulmx_adjr : forall n (A : 'M_n), A * adjugate A = (\det A)%:M.
Lemma mulmx_adjl : forall n (A : 'M_n), adjugate A * A = (\det A)%:M.
```

Dans les énoncés ci-dessus, `(\det A)%:M` est la matrice scalaire associée à `\det A`. La preuve du premier lemme utilise la formule 3.6 de Laplace et la propriété selon laquelle le déterminant est une forme alternée (le déterminant d'une matrice où au moins deux lignes sont identiques est nul).

En plus du développement sur le déterminant, la bibliothèque sur les matrices comprend aussi un développement sur la décomposition LUP [14]. Ceci et comme nous allons le voir dans le prochain chapitre ouvre la voix au support de la résolution des systèmes linéaires.

3.3.4 Polynôme

Dans cette section nous présentons le développement sur les polynômes. La principale motivation de ce travail est de développer l'infrastructure nécessaire pour faire de la théorie de Galois, une des composantes de la preuve du théorème de Feit-Thompson. C'est aussi un pré-requis à la formalisation du théorème de Cayley-Hamilton.

3.3.4.1 Définitions

En algèbre, un polynôme p à une indéterminée est défini comme une expression formelle de la forme

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

où les a_i sont appelés les coefficients de p et ils appartiennent à un anneau donné R. Un polynôme est donc représenté par la liste de ses coefficients. Cette représentation qui est appelée pleine 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. Dans un contexte de formalisation sur ordinateur ceci pose problème. Sur machine, les listes $\{a_n, a_{n-1}, \cdots, a_1, a_0\}$ et $\{0, a_n, a_{n-1}, \cdots, a_1, a_0\}$ ne sont pas identiques. Une solution est de prendre un représentant canonique. Dans le cas des polynômes, ce représentant est le polynôme dont le coefficient de plus haut degré est non nul. Par exemple les polynômes $0x^2 + x + 1$ et $x + 1$ ont comme représentant canonique le polynôme $x + 1$.

En utilisant le développement antérieur sur les listes (bibliothèque `seq`), nous définissons le type des polynômes comme un type `Structure` Coq composé de la liste des coefficients et de la preuve que le dernier élément de cette liste est non nul.

```
Variable R : ringType.
Record polynomial : Type :=
 Polynomial {polyseq :> seq R; normal : last 1 polyseq != 0}.
```

Dans la définition ci-dessus, `last` est une fonction à deux arguments un élément de `R` et une liste sur `R`. Le résultat de `last e l` est le dernier élément de `l` si `l` est non vide et `e` sinon. Avec cette définition, le polynôme $3x^2 + 2x + 1$ est représenté par la liste

$\{1, 2, 3\}$. Les coefficients de plus faible degré sont placés les premiers dans la liste. Le polynôme nul va être représenté par la liste vide.

Le type `polynomial` hérite de la structure d'`eqType` du type `seq`, le type des listes, puisque les éléments de la liste appartiennent à `R` qui possède aussi une structure d'`eqType`.

Les polynômes comme suites finies : le paramètre `polyseq` du type `polynomial` définit une *coercion* de `polynomial` vers `seq`, le type des listes dans SSReflect. Ainsi le type `polynomial` hérite des opérations sur les listes. En particulier, nous pouvons appliquer à un polynôme la fonction `nth` (Section 2.3.1) qui retourne la valeur correspondant à un indice donné d'une liste. Pour un type donné `T`, la fonction `nth` est de type `T → seq T → nat → T`. Dans le cas d'un polynôme `p`, `nth 0 p` est donc une fonction de type `nat → R` et qui est toujours nulle à partir d'une certaine valeur. Ceci correspond à une autre définition des polynômes qui peuvent aussi être définis comme des suites finies. Dans la suite la notation `p`_i` représente `nth 0 p i`. Ainsi la fonction `nth 0 p` fournit une interprétation fonctionnelle des polynômes. Elle permet d'oublier la propriété de canonicité du polynôme et ne considérer que les coefficients. Pour avoir l'équivalence entre cette interprétation et la définition du type `polynomial`, nous prouvons que deux polynômes sont égaux si et seulement si leurs suites de coefficients sont égales. Ceci est donné par le lemme `polyP`.

```
Lemma polyP : forall p1 p2, nth 0 p1 =1 nth 0 p2 <-> p1 = p2.
```

Dans l'environnement logique de Coq où l'extensionnalité nécessite un axiome, le lemme ci-dessus nous permet d'avoir une égalité extensionnelle (notation `=1`) entre les polynômes. Cette égalité est plus facile à manipuler pour les preuves des propriétés algébriques des polynômes. Un exemple est la preuve de l'associativité de la multiplication des polynômes qui peut se ramener à des raisonnements sur des sommes indexées. Nous reviendrons à cette exemple lors de la définition de l'anneau des polynômes.

Construction de polynôme : dans la définition du type `polynomial` ci-dessus, le constructeur par défaut du type est la fonction `Polynomial` qui prend en paramètre une liste et une preuve que son dernier élément est non nul. En pratique dans Coq la liste peut être inférée à partir de la preuve. Mais d'un point de vue utilisateur il est plus facile de définir un polynôme à partir d'une liste de coefficients et laisser au système la charge de mettre cette liste sous forme canonique. Ceci correspond à raisonner récursivement sur la liste des coefficients. On peut aussi, comme dans le cas des matrices, définir

un polynôme à partir d'une fonction de coefficients et d'une borne supérieure de cette fonction. Les coefficients au-delà de cette borne ne seront pas pris en compte.

Nous commençons par définir la fonction qui étant donnés une constante `c` et un polynôme `p` (une liste normalisée), retourne le polynôme `c + x*p`. Par exemple pour les arguments 4 et $1 + 2x + 3x^2$, le résultat sera $4 + x + 2x^2 + 3x^3$. La définition Coq de cette fonction est la suivante :

```
Definition poly_cons c p : {poly R} :=
 if p is Polynomial ((a :: q) as s) ns then @Polynomial (c :: s) ns
 else c%:P.
```

La fonction `poly_cons` procède par une analyse de cas sur la liste des coefficient du polynôme passé en argument. Si cette liste est non nulle, alors elle retourne le polynôme dont la liste des coefficients est `(c :: s)` ; sinon elle retourne le polynôme constant `c` noté `c%:P`. Dans la première branche de l'analyse par cas, `ns` est une preuve de type `last 1 (a :: q)!= 0` ; alors que la preuve attendu pour `@Polynomial (c :: s)` est de type `last 1 (c :: (a :: q))!= 0`. Coq arrive à unifier ces deux types après $\beta\iota$-réduction dans la définition de `last`.

Ainsi, étant donnée une liste de coefficients pas forcement normalisée, pour construire le polynôme correspondant, il faut appliquer itérativement la fonction `poly_cons` à cette liste. Ceci est donné par la fonction `Poly` qui utilise la fonction `foldr` de SSReflect.

```
Definition Poly := foldr poly_cons 0%:P.
Notation "\poly_ ( i < n ) E" := (Poly (mkseq (fun i : nat => E) n)).
```

Au dessus de la définition de `Poly`, la notation `\poly_` fournit à l'utilisateur la possibilité de définir un polynôme depuis une fonction de coefficients et une borne supérieure. Cette notation utilise la fonction `mkseq` qui permet de construire une liste à partir d'une fonction d'indice et de la longueur de la liste. Par exemple, le polynôme $nx^{n-1} + n - 1x^{n-2} + \cdots + 2x + 1$ est représenté par `\poly_(i < n)i+1`.

Remarques : la notation `\poly_` et le lemme `polyP` fournissent à l'utilisateur les moyens de définir un polynôme à partir d'une fonction de coefficients et de pouvoir revenir à ces fonctions pour raisonner sur les polynômes qu'elles définissent. Ainsi l'utilisateur n'a pas à se soucier de la manière dont les polynômes sont représentés. Pour faire des preuves sur les polynômes ceci est important puisque ce qui nous intéresse c'est la liste des coefficients. Le fait que cette liste soit normalisée ou non n'influence pas les définitions des opérations sur les polynômes ou les propriétés correspondantes.

C'est principalement nécessaire pour avoir une représentation canonique sur machine et pouvoir utiliser l'égalité de Leibniz.

Dans la perspective de faire des calculs avec les polynômes, il est possible de définir une nouvelle représentation comme celle décrite dans [35]. Il faudra dans ce cas prouver qu'il y a une correspondance entre cette autre représentation et la notre pour pouvoir utiliser les preuves fournies avec celle-ci.

En mathématiques le degré d'un polynôme $p = \sum_{i=0}^{n} a_i x^i$ non nul est donné par : $\max\{i \leq n, a_i \neq 0\}$. Le degré du polynôme nul est lui égale à $-\infty$. Dans notre représentation des polynômes nous ne définissons pas de fonction spécifique pour le degré. Comme nous représentons les polynômes par la liste normalisée de leurs coefficients, le degré d'un polynôme non nul peut être déduit de la longueur de cette liste. En effet le degré est égal à la longueur de la liste moins un. Ceci ne s'applique pas au polynôme nul. Les raisonnements qui utilise le degré peuvent être ainsi ramenés à des raisonnements sur la longueur de la liste des coefficients et une comparaison avec le polynôme nul.

3.3.4.2 Anneau de polynômes

La définition d'une structure d'anneau pour les polynômes est nécessaire pour la preuve du théorème de Cayley-Hamilton puisque nous allons manipuler des matrices de polynômes. Ce qui requiert que notre représentation des polynômes soit munie d'une structure d'anneau puisque le type des matrices est paramétré par un anneau.

Un autre avantage de cette définition est de pouvoir aussi définir les polynômes à plusieurs indéterminées. Une façon standard de définir de tels polynômes est de les voir comme des polynômes sur l'anneau des polynômes : des polynômes dont les coefficients sont des polynômes.

La définition de la structure d'anneau pour les polynômes commence par la définition des opérations algébriques : l'addition, l'opposé et la multiplication. Ces opérations sont définies en utilisant la notation `\poly_(i < n)E`. La borne supérieure `n` de la longueur de la liste des coefficients du polynôme résultat est déterminée en fonction de celles des polynômes arguments.

L'addition de deux polynômes est le polynôme dont les coefficients sont la somme des coefficients de même indice de chacun des deux polynômes. Une borne supérieur de la longueur de la liste des coefficients du nouveau polynôme est le maximum de celles des polynômes de départ. L'opposé d'un polynôme correspond à inverser le signe des coefficients du polynôme. La longueur de la liste des coefficients reste la même. Ces deux fonctions sont définies comme suit :

```
Definition add_poly p1 p2 :=
 \poly_(i < maxn (size p1) (size p2)) (p1`_i + p2`_i).
Definition opp_poly p := \poly_(i < size p) - p`_i.
```

La multiplication de deux polynômes formels est définie théoriquement par la formule suivante :

$$\left(\sum_{i=0}^{m} a_i x^i\right)\left(\sum_{j=0}^{n} b_j x^j\right) = \sum_{i=0}^{m+n}\left(\sum_{j\leq i} a_j b_{i-j}\right) x^i. \tag{3.19}$$

D'après cette formule le degré du polynôme produit est égale à la somme des degrés. Ceci est vrai uniquement si l'anneau des coefficients est intègre, *i.e.* le produit de deux éléments non nul est aussi non nul. En effet, dans anneau général même si les deux coefficients de tête des polynômes de départ sont non nuls leur produit peut être nul. La somme des degrés fournit donc une borne supérieure au degré du polynôme produit. Après traduction de la condition sur le degré en condition sur la longueur de la liste des coefficients, le code Coq pour la définition du produit des polynômes est le suivant :

```
Definition mul_poly p1 p2 := \poly_(i < (size p1 + size p2).-1)
 (\sum_(j < i.+1) p1`_j * p2`_(i - j)).
```

Dans la définition ci-dessus nous utilisons les sommes indexées (la notation `\sum_`). Ce choix permet d'avoir un code Coq relativement proche de la formule mathématique standard 3.19. Il permet aussi de profiter des propriétés génériques fournies par la bibliothèque sur les opérations indexées. Ainsi la preuve de l'associativité de la multiplication des polynômes revient à utiliser les propriétés de ré-indexation et de distributivité des sommes indexées.

Après la définition des opérations algébriques et la preuve des axiomes d'anneaux correspondants, nous définissons une structure canonique d'anneau pour les polynômes.

```
Canonical Structure poly_ringType R := @RingType (polynomial R) ...
```

Nous pouvons ainsi écrire la formule d'expansion d'un polynôme sous la forme d'une somme finie de produits de monômes.

```
Lemma poly_def : forall n E,
 \poly_(i < n) E i = \sum_(i < n) (E i)%:P * 'X^i.
```

Dans cet énoncé, la structure canonique d'anneau permet d'utiliser l'opération de multiplication d'anneau représentée par la notation `*`. La notation `'X^i` représente le monôme x^i. Ce dernier n'est autre que le monôme x à la puissance i. La fonction d'exposant est

définie de façon générique pour les anneaux. Grâce aux structures canoniques, le type des polynômes hérite de cette notation.

Avec la définition de la structure d'anneau pour les polynômes, nous allons être capable de déclarer des matrices de polynômes. Nous pouvons ainsi définir un isomorphisme entre les matrices de polynômes et les polynômes de matrices. Cet isomorphisme permet d'avoir une correspondance entre ces deux structures. Avant ceci, nous avons besoin de développer une théorie des fonctions polynomiales ou évaluation des polynômes. Ceci est un pré-requis pour la formalisation du théorème de Cayley-Hamilton.

3.3.4.3 Évaluation de polynômes

L'opération d'évaluation d'un polynôme $p(x) = \sum_{i=0}^{n} c_i x^i \in R[X]$ en une valeur $a \in R$ consiste à remplacer le x dans la définition de p par a. Cette opération peut être définie en utilisant le schéma de Horner :

$$p(a) = ((((c_n a + c_{n-1})a + c_{n-2})a + ...) + c_1)a + c_0.$$

Suivant ce schéma, l'évaluation ne dépend que de la liste des coefficients et de la valeur où nous évaluons. La fonction d'évaluation peut donc être définie par récurrence sur une liste arbitraire sur l'anneau des coefficients comme suit :

```
Variable R : ringType.
Fixpoint horner s (a : R) {struct s} :=
  if s is c :: s' then horner s' a * a + c else 0.
```

Dans notre définition de l'évaluation de polynôme, l'anneau des coefficients n'est pas commutatif. Ceci, comme nous l'avons vu dans la section 3.2.3, ne correspond pas à la définition mathématique usuelle. Notre choix de définir l'évaluation sur les anneaux non commutatifs vient du fait que ceci couvre les deux cas de l'évaluation sur un anneau commutatif et une algèbre associative. L'évaluation d'un polynôme en un élément d'une algèbre va correspondre à injecter les coefficients du polynôme dans l'algèbre pour obtenir un polynôme sur l'algèbre et appliquer ensuite l'évaluation. Par exemple, soient R un anneau, A une R-algèbre, $p = \sum_{i=0}^{n} c_i x^i \in R[X]$ et $a \in A$. L'application de l'injection canonique de R dans A aux coefficients c_i du polynôme p donne un nouveau polynôme $P \in A[x]$. L'évaluation du polynôme P en a est égale à :

$$P(a) = \sum_{i=0}^{n} (c_i * 1_A)a^i$$

avec 1_A l'identité de l'algèbre et $*$ l'action de R sur A. Mais cette valeur est aussi égale à $\sum_{i=0}^{n} c_i a^i$ et par conséquent $p(a)$.

Dans la suite, la notation `p.[a]` représente l'évaluation du polynôme `p` en `a`. Elle correspond à l'application de la fonction `horner` à la liste des coefficients de `p` qui est donnée par la projection `polyseq` du type `polynomial`.

Comme nous l'avons déjà précisé dans la section 3.2.3 la fonction d'évaluation des polynômes possède des propriétés de morphisme. Ces propriétés sont utilisées de manière implicite dans la preuve du théorème de Cayley-Hamilton. Elles sont données par les lemmes Coq suivants :

```
Lemma hornerC : forall c a, (c%:P).[a] = c.
Lemma horner_add : forall p q a, (p + q).[a] = p.[a] + q.[a].
Lemma horner_mul_com : forall p q a,
 a * q.[a] = q.[a] * a -> (p * q).[a] = p.[a] * q.[a].
```

Dans le lemme `horner_mul_com` et puisque nous définissons l'évaluation sur un anneau non commutatif, il est nécessaire que la valeur `a` où l'on évalue le polynôme `p * q` commute avec l'évaluation de `q` en cette même valeur. Comme nous l'avons déjà précisé dans 3.16, en général dans la littérature mathématique la condition pour avoir cette propriété est que `a` commute avec tous les coefficients du polynôme `q`. Mais cette condition est plus forte que la première. En effet pour un polynôme $p(x) = \sum_{i=0}^{n} c_i x^i$ et une constante a nous avons que :

$$\begin{aligned} \forall i \leq n, ac_i = c_i a \Rightarrow p(a) &= \sum_{i=0}^{n} c_i a^i = \sum_{i=0}^{n} a^i c_i \\ \Rightarrow ap(a) &= \sum_{i=0}^{n} a^{i+1} c_i = \sum_{i=0}^{n} a^i c_i a \\ \Rightarrow ap(a) &= p(a)a. \end{aligned}$$

La réciproque de cette propriété n'est pas toujours vraie dans un anneau non commutatif.

Dans la suite du développement sur l'évaluation des polynômes, nous avons formaliser le théorème du reste. Ce théorème permet de dire qu'une constance c est racine d'un polynôme p si et seulement si le polynôme p peut se factoriser en $(x - c)$ (*i.e.* $\exists q, p = q(x - c)$).

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

Dans la preuve de ce théorème, il faut utiliser le fait que l'évaluation du produit de polynômes `q * ('X - c%:P)` en `c` est égale au produit des évaluations. Cette égalité est vrai dans n'importe quel anneau. En particulier lorsque nous appliquons le lemme

`horner_mul_com` il est automatique de vérifier que la valeur de `('X - c%:P)` en `c` (qui est égale à `0`) commute avec `c`.

Avec la preuve du théorème du reste nous pouvons maintenant formaliser la preuve du théorème de Cayley-Hamilton. Ce travail est présenté dans la prochaine section.

3.3.5 Formalisation du théorème de Cayley-Hamilton

L'isomorphisme entre l'anneau des matrices 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 du reste, sont des propriétés qui se rattachent respectivement à la théorie des matrices et à celle des polynômes. Comme décrit dans l'exemple 3.11, cet isomorphisme que nous notons ϕ prend en argument une matrice de polynômes. Pour un monôme qui apparait dans les polynômes de cette matrice, la fonction ϕ assemble dans une matrice tous les coefficients associés à ce monôme dans les polynômes de la matrice de départ. Le résultat sera un polynôme à coefficients matriciels. La longueur de la liste des coefficients du polynôme résultat est le maximum des longueurs des listes de coefficients des polynômes de la matrice de départ. C'est le plus grand exposant des monômes des polynômes de la matrice de départ. L'isomorphisme entre les matrices de polynômes et les polynômes de matrices est défini comme suit :

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

La notation `\max_` représente le maximum. C'est une instance de la notation `\big` des opérations indexées comme décrit dans 2.3.5. Les bornes supérieures des ensembles des indices sont inférées à partir du type de la matrice et plus précisément sa taille.

Après la définition du morphisme `phi`, nous prouvons la relation entre la matrice de polynômes qu'il prend en argument et le polynôme de matrices résultat. Cette relation est donnée par le lemme suivant :

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

Le lemme `coef_phi` prouve que pour une matrice de polynômes donnée `A`, le coefficient d'indice `i` et `j` de la matrice `(phi A)`_k` (le coefficient d'indice `k` du polynôme `(phi A)`) est égale au coefficient `k` du polynôme `A i j`. En d'autres termes, il prouve que la fonction `phi` respecte ses spécifications. Nous utilisons ce lemme pour prouver que `phi` est un morphisme d'anneau : distributivité par rapport à l'addition, la multiplication et les élément neutres.

```
Lemma phi_add : forall (A B : M(R[X])), phi (A + B) = phi A + phi B.
Lemma phi_mul : forall (A B : M(R[X])), phi (A * B) = phi A * phi B.
Lemma phi_zero : phi 0 = 0.
Lemma phi_one : phi 1 = 1.
```

Dans ces égalités, et avec l'utilisation des structures canoniques, nous utilisons la même opération pour les matrices et les polynômes. Par exemple, dans la partie gauche de la première égalité l'addition est entre matrices alors que dans la partie droite elle est entre polynômes. Grâce à la déclaration d'une structure canonique d'anneau pour les matrices et les polynômes, Coq arrive à inférer automatiquement l'opération correspondante dans chacun des cas. Ceci se fait de façon transparente pour l'utilisateur. L'utilisation des structures canoniques avec l'interface d'anneau, en plus de fournir une théorie générique pour les opérations algébrique, permet aussi d'avoir des énoncés plus lisibles. Ce qui est important pour la maintenance et la ré-utilisation du développement.

Formalisation : pour énoncer le théorème de Cayley-Hamilton il est nécessaire de définir le polynôme caractéristique d'une matrice. Comme nous l'avons décrit dans 3.1 ce polynôme est le déterminant de la matrice $xI_n - A$ qui est une matrice de polynômes. Dans l'anneau des matrices de polynômes xI_n est la matrice scalaire en x et A est la matrice dont les coefficient sont des polynômes constants. Dans la définition mathématique du polynôme caractéristique $A \in M_n(R)$ est vu implicitement comme une matrice de polynômes (*i.e.* $\in M_n(R[X])$) avec l'injection canonique de $M_n(R)$ dans $M_n(R[X])$. Notre définition Coq du polynôme caractéristique est la suivante :

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

La fonction `matrixC` est l'injection canonique de l'anneau des matrices vers celui des matrices de polynômes. Elle correspond à construire à partir d'une matrice donnée `A` la matrice dont les coefficients sont les polynômes constants associés aux coefficients de `A`.

Pour une matrice carrée A son polynôme caractéristique p_A appartient à $R[X]$ alors que A appartient elle à $M_n(R)$ qui est une R-algèbre. Comme nous l'avons fait remarquer dans la section 3.3.4.3, nous définissons l'évaluation sur un anneau non commutatif. Pour pouvoir donc évaluer p_A en A il faut injecter les coefficients de p_A qui appartiennent à R dans la R-algèbre $M_n(R)$. Cette injection est donnée par la fonction suivante :

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

```
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.
```

FIGURE 3.3 – L'énoncé et la preuve formelle du théorème de Cayley-Hamilton.

Une propriété de cette injection est que pour un polynôme donné `p`, `Zpoly p`, qui est un polynôme de matrices, est égal au polynôme de matrices associé à la matrice scalaire de `p`. Ce dernier correspond à l'application de la fonction `phi` à la matrice de polynômes `p%:M`.

```
Lemma phi_Zpoly : forall p, phi p%:M = Zpoly p.
```

Après ces définitions, le théorème de Cayley-Hamilton est énoncé et prouvé formellement comme dans la Figure 3.3. La preuve se déroule exactement comme décrit dans la section 3.2.2. Après avoir appliqué le théorème du reste (`factor_theorem`), le polynôme facteur est donné en récrivant avec la règle de Cramer (`mulmx_adjl`). Le résultat du théorème de Cayley-Hamilton est alors prouvé par des réécritures avec les propriétés du morphisme `phi` et simplifications dans les termes obtenus. L'utilisation du langage de tactique de SSReflect, des mécanismes des `Canonical Structure` de Coq, ainsi que la décomposition modulaire de différentes théories composant la preuve (anneaux, polynômes et matrices) ont permis d'aboutir à une preuve aussi concise : 3 lignes de script.

3.4 Travaux reliés

Le développement que nous avons présenté ici est à notre connaissance la première formalisation du théorème de Cayley-Hamilton dans un assistant à la preuve. Des formalisations des théories des polynômes et matrices ont été développées dans différents systèmes de preuve.

Dans la bibliothèque des preuves formelles du système Mizar [36] plusieurs développements sur les polynômes et les matrices sont présentés. Ces travaux vont jusqu'à la formalisation du théorème du reste [46] pour les polynômes et la règle de Cramer pour les matrices [40]. Cette dernière est prouvée pour des matrices sur un corps. Le théorème du reste est prouvé pour des polynômes sur un anneau commutatif. Ce qui ne permet pas de l'utiliser pour des polynômes de matrices.

Des formalisations du théorème du reste et de la règle de Cramer sont aussi disponibles dans les bibliothèques des preuves formelles des systèmes HOL-Light [25] et Isabelle [37]. Ici aussi le théorème du reste est prouvé pour des polynômes sur un anneau commutatif. L'absence de type dépendant dans le système HOL-Light font que la règle de Cramer n'est prouvée que pour les matrices réelles. Dans le cas du système Isabelle la théorie des anneaux est développée en utilisant les Types Classes qui fournissent des mécanismes semblables aux `Canonical Structure` de Coq.

Dans le système Coq une formalisation des matrices [34] a été développée en utilisant les types dépendants. Les matrices sont représentées comme des listes de vecteurs. Cependant ce travail est principalement un exercice d'utilisation des types dépendants et ne fournit pas de résultats sur les déterminants. Le répertoire de preuve formelle de Nijmegen [9] comprend entre autres une formalisation des polynômes. Ces derniers sont définis pour des anneaux commutatifs qui sont à leur tour définis sur des types munis d'une égalité de `Setoid`. Dans notre cas, étant donnée que les structures concrètes pour lesquelles nous aurons à instancier ces théories vont avoir une égalité décidable, nous avons choisi d'utiliser l'interface `eqType`. Ceci nous permet d'utiliser l'égalité de Leibniz et profiter de la puissance de réécriture de Coq.

3.5 Conclusion

Formaliser un résultat mathématique consiste à écrire la preuve papier (ou en terme logiciel la spécification) de ce résultat dans un langage machine. Ceci nécessite beaucoup plus de détails que ce qui est requis dans une preuve mathématique standard. Par ailleurs le manque de bibliothèques de preuves formelles impose le développement de telles bibliothèques avant d'envisager de prouver des résultats majeurs. L'organisation et la facilité d'utilisation de ces bibliothèques sont des paramètres importants puisque les théories mathématiques ne sont pas connectées seulement de façon verticale mais aussi de façon horizontale. Ces paramètres sont aussi importants vu l'immensité des connaissances mathématiques qui impose un travail collaboratif et de longue durée pour la formalisation de ces connaissances.

Dans ce chapitre nous avons présenté une première formalisation du théorème de Cayley-Hamilton dans un assistant à la preuve. Cette formalisation a été faite en adoptant une approche modulaire. L'architecture de la formalisation de la preuve est donnée dans la Figure 3.4. En partant du développement sur les matrices, nous avons développé une théorie des polynômes. La preuve du théorème de Cayley-Hamilton est construite au dessus de ces deux développements. Nous avons aussi développé une théorie des

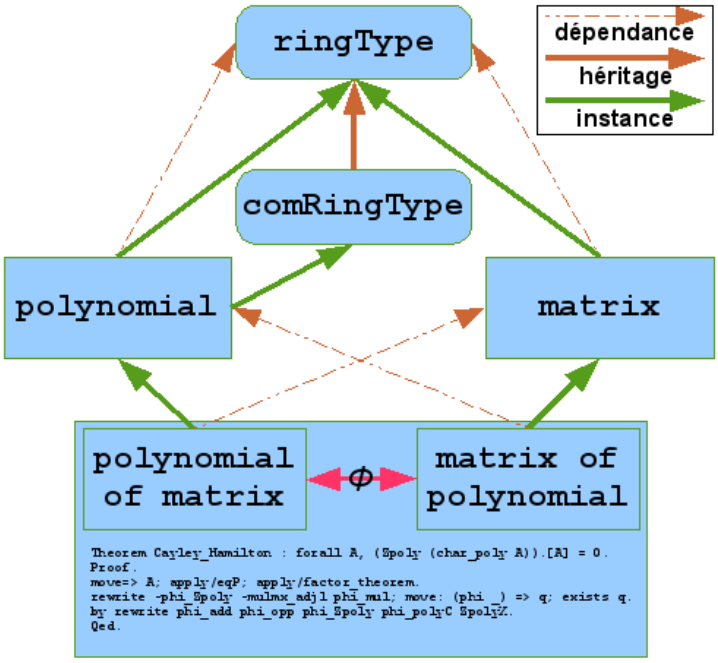


FIGURE 3.4 – Architecture de la formalisation du théorème de Cayley-Hamilton.

anneaux qui nous a permis d'avoir des opérations et théories algébriques génériques pour les polynômes et les matrices. Ces deux structures mathématiques ont été définies comme paramétrées par un anneau. Ceci nous a ainsi permis de combiner ces deux théories pour avoir les matrices de polynômes et les polynômes de matrices. Cette approche modulaire nous a permis d'aboutir à une preuve courte (3 lignes) du théorème de Cayley-Hamilton. Ceci est du au fait que les résultats utilisés par le théorème de Cayley-Hamilton sont prouvés de façon générique et instanciés par la suite pour des cas particuliers. Un autre avantage de cette approche est de faciliter la maintenance et l'extension des bibliothèques. Par exemple, dans le cas des interfaces pour les anneaux et même si la façon dont ces structures sont définies a complètement changé, comme nous le verrons dans le prochain chapitre, ceci n'a pas provoqué des changements majeurs dans la forme des preuves déjà développées. Le développement sur les polynômes a aussi été étendu par la suite par Laurent Théry pour supporter la pseudo division des polynômes. Ce qui ouvre la voix pour des développements sur la théorie de Galois.

La preuve du théorème de Cayley-Hamilton, même si elle n'est pas comparable avec celle du théorème de Feit-Thompson au niveau de la complexité et de la longueur, partage avec celle-ci des théories sous-jacentes. La formalisation de cette preuve nous a permis de tester ces théories qui sont un pré-requis à toute formalisation du théorème

de Feit-Thompson. En particulier les développements sur les matrices et les anneaux ouvrent la voix à une formalisation de la théorie des représentations des groupes finis. Cette dernière et comme nous le verrons dans le chapitre suivant fait appel à de nombreux résultats d'algèbre linéaire.

Chapitre 4

Composants pour les représentations des groupes finis

4.1 Introduction

La théorie des représentations est un outil mathématique pour décrire et analyser une structure algébrique en termes d'applications linéaires et donc de matrices. Les éléments de la structure algébrique vont être représentés par des matrices et les opérations entre ces éléments vont correspondre à des opérations entre matrices. Les structures algébriques étudiées sont principalement les groupes finis, les algèbres de Lie ou plus généralement les algèbres associatives.

La théorie des représentations est l'une des rares théories mathématiques pour lesquelles il est possible d'associer une date "d'invention". L'origine de la théorie est associée à une correspondance entre R. Dedekind(1831-1916) et F. G. Frobenuis(1849-1917) datée d'avril 1896 [32]. R. Dedekind cherchait à factoriser des polynômes spécifiques associés à des groupes finis et appelés déterminants de groupes. En partant de ce problème, Frobenuis développa une théorie générale des groupes finis. Il formula ainsi la notion moderne de représentation d'un groupe par des matrices. Ceci ouvra la voie à de nouveaux travaux de recherche en théorie des groupes. En particulier, W. Burnside(1852-1927), qui est considéré comme le deuxième fondateur de la théorie, développa et prouva plusieurs résultats sur les représentations des groupes finis. Le travail de Burnside sur les représentations posa les fondations pour les travaux sur la classification des groupes finis. Cette dernière a été un sujet de recherche mathématique très actif durant le vingtième siècle.

La preuve du théorème de Feit-Thompson [13] est une des applications de la théorie des représentations à l'étude des groupes finis. Elle fait appel à des résultats sur la théorie des caractères des groupes finis et par conséquent les représentations de ces groupes. La théorie possède des applications variées en algèbre ou en théorie des nombres. Elle est ainsi utilisée dans le programme de Langlands [15], un domaine actif de la recherche mathématique contemporaine. Dans les mathématiques appliquées, la théorie des représentations est utilisée dans des domaines variés comme la cristallographie ou la mécanique quantique [32].

Dans ce chapitre nous présentons notre travail de formalisation de la théorie des représentions des groupes finis. Ce travail a déjà été présenté sous une forme moins détaillée dans [39]. Nous allons commencer par donner une introduction à la théorie des représentations. En particulier nous décrivons la relation entre les représentations et les modules. Nous allons par la suite présenter les différentes composantes du développement formel sur les représentations : les modules et les G-modules. Ce développement contient une hiérarchie de structures d'algèbre linéaire et une formalisation du théorème de Maschke sur les représentions des groupes finis. Enfin, nous conclurons par une discussion sur les perspectives d'évolution de ce développement ainsi qu'une brève présentation des travaux reliés.

4.2 La théorie des représentations

Cette section est une introduction à la théorie des représentations. Nous commençons par définir ce qu'est une représentation et nous donnons certaines définitions auxquelles fait appel la théorie. Nous présentons par la suite la preuve du théorème de Maschke. Nos références pour cette présentation sont les livres [28, 30] et les notes de cours de P. Webb [54].

4.2.1 Définitions

Rappels

Soit un corps F et A un F-espace vectoriel. Supposons que A est aussi un anneau unitaire. Il est donc muni d'une multiplication interne et d'un élément neutre pour cette opération. Si en plus nous avons :

$$\forall c \in F, x\ y \in A;\ (cx)y = c(xy) = x(cy).$$

Alors A est une F-algèbre. L'algèbre A est dite de dimension finie si l'espace vectoriel correspondant est de dimension finie. Les espaces vectoriels et les algèbres qui sont considérés dans cette section et dans la suite du chapitre sont supposés de dimension finie.

Une application linéaire est un morphisme de module entre espaces vectoriels. Pour deux F-espaces vectoriels V et W, une application linéaire de V dans W est une fonction $f : V \to W$ tel que :

$$\forall a \in F, u\ v \in V, f(au + v) = af(u) + f(v). \tag{4.1}$$

Une fonction est dite F-linéaire si elle respecte la propriété ci-dessus. Un homomorphisme d'algèbre entre deux F-algèbres A et B est une fonction $\phi : A \to B$ qui est F-linéaire et satisfait les conditions suivantes :

$$\phi(1_A) = 1_B \ \& \ \forall x\ y \in A, \phi(xy) = \phi(x)\phi(y). \tag{4.2}$$

Dans la suite, pour un entier naturel n donné, l'ensemble des matrices carrés de taille n et à coefficients dans F sera noté $M_n(F)$. Rappelons que cet ensemble possède une structure de F-algèbre.

Représentations

Les représentations sont des homomorphismes mathématiques qui permettent de ramener l'étude de certaines structures algébriques à une analyse de matrices. Généralement une représentation peut être définie pour toute algèbre associative sur un corps. Soit A une F-algèbre, une représentation de A est un homomorphisme d'algèbre $\rho : A \to M_n(F)$. L'entier n est appelé degré de la représentation. Deux représentations ρ et σ sont dites semblables si il existe une matrice inversible $P \in M_n(F)$, tel que $\forall a \in A, \rho(a) = P^{-1}\sigma(a)P$. La relation *"être semblable"* entre les représentations est une relation d'équivalence.

Les représentations des groupes finis sont un cas particulier de représentations. Soit G un groupe fini, une F-représentation de G est un homomorphisme de groupe $\rho : G \to GL(n, F)$, où $GL(n, F)$ est le groupe multiplicatif des matrices $n \times n$ inversibles et à coefficients dans F. Une représentation de G est donc une fonction ρ de G dans $GL(n, F)$ qui possède les propriétés suivantes :

$$\rho(1_G) = I_n \qquad et \qquad \forall g\ h \in G\ \rho(gh) = (\rho g)(\rho h).$$

Exemple : soit p un nombre premier, posons $F = \mathbb{Z}/\mathbb{Z}_p$, $n = 2$ et $G = C_p = \langle g \rangle$ le groupe cyclique d'ordre p et généré par un élément g. La fonction ρ définie par

$$\rho(g^r) = \begin{pmatrix} 1 & 0 \\ r & 1 \end{pmatrix}$$

est une F-représentation de G. En effet, l'image de l'élément neutre de G, qui est égal à g^0, est la matrice identité I_2. Par ailleurs nous avons aussi que :

$$\rho(g^r)\rho(g^s) = \begin{pmatrix} 1 & 0 \\ r & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ s & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ r+s & 1 \end{pmatrix} = \rho(g^{r+s}) = \rho(g^r g^s).$$

4.2.2 Représentations et modules

Dans la littérature mathématique sur la théorie des représentations [28, 30, 54], l'étude des représentations se fait en général en les considérant comme des modules. Le livre de I.M. Issacs [28], la référence pour la théorie des représentations pour la preuve du théorème de Feit-Thompson, commence l'étude des représentations par une introduction à la théorie des modules finiment engendrés sur une algèbre.

Module de représentation

Avant de présenter l'équivalence entre modules et représentations, nous allons rappeler quelques définitions sur les modules. Un module sur un anneau A (ou un A-module) est un groupe abélien V muni d'une action de A sur V. Les éléments de l'anneau A sont appelés scalaires. Pour un A-module V, nous avons les définitions suivantes :

- **sous-module** : un sous espace de W de V est un sous-module de V si et seulement si il est invariant par l'action de A sur V *i.e.* $\forall x \in A, v \in W, vx \in W$. Un sous-module est aussi un module.
- **simple** : V est dit simple si et seulement si ses seuls sous-modules sont 0 et V. C'est un module qui ne pas être décomposé.
- **semi-simple** : V est dit semi-simple si pour tout sous-module $W \subseteq V$, il existe un sous-module $U \subseteq V$ tel que $V = W \oplus U$ *i.e.* $V = W + U$ et $W \cap U = 0$. Cette propriété est équivalente au fait que V peut s'écrire sous la forme d'une somme finie de modules simples.

Un module sur une algèbre est un cas spécial de modules. Dans le cas des modules sur une algèbre, l'anneau des scalaires est une algèbre. Les définitions standard sur les modules sur un anneau s'appliquent aussi aux modules sur une algèbre. En particulier,

les modules simples sur une algèbre sont les modules qui ne peuvent pas être décomposés. Dans certaines algèbres, tous les modules se décomposent en somme de modules simples. L'étude des modules sur cette algèbre se ramène donc à l'étude des modules simples.

L'équivalence : chaque représentation d'une F-algèbre donnée A définit un A-module et vice versa. En effet, étant donnée une représentation ρ de A de degré n, posons $V = M_{1,n}$: c'est un groupe abélien. Pour tout $v \in V$ et $X \in M_n(F)$, nous avons que $vX \in V$. La fonction qui pour tout $a \in A$ et $v \in V$ associe $va = v\rho(a)$ définit une action de A sur V. Avec cette action, V est muni d'une structure de A-module.

Inversement, étant donné un A-module V de dimension n, pour tout $a \in A$ soit la fonction $a_v : v \mapsto va$. C'est une F-application linéaire de V dans V. Notons $M_{a_v} \in M_n(F)$ la matrice de a_v par rapport à une base de V. La fonction $\rho : a \mapsto M_{a_v}$ est un homomorphisme d'algèbre de A dans $M_n(F)$. Elle définit donc une représentation de A.

L'équivalence entre modules sur une algèbre et représentations permet la réutilisation des définitions et résultats généraux sur les modules. Par exemple, une représentation est dite semi-simple si le module associé est semi-simple.

Algèbre de groupe

Une structure algébrique qui joue un rôle important en théorie des représentations des groupes finis est celle de l'algèbre de groupe. Soit G un groupe fini, la F-algèbre de groupe associée à G, notée $F[G]$, est l'ensemble des sommes "formelles" $\{\sum_{g\in G} a_g g \mid a_g \in F\}$. Les opérations d'addition et de multiplication externe dans $F[G]$ sont définies comme suit :

$$\sum_{g\in G} a_g g + \sum_{g\in G} b_g g = \sum_{g\in G} (a_g + b_g)g \quad \text{et} \quad \alpha \sum_{g\in G} a_g g = \sum_{g\in G} (\alpha * a_g)g.$$

Avec ces opérations, $F[G]$ est un F-espace vectoriel. Pour $g \in G$, nous noterons $1g$ la somme $\sum_{h\in G} a_h h$ avec $a_g = 1$ et $a_h = 0$ *si* $h \neq g$. La fonction qui à g associe $1g$ injecte G dans $F[G]$. Par abus de notation, nous pouvons donc écrire que $G \subseteq F[G]$. Les éléments de G forment ainsi une base de $F[G]$ par rapport à F. $F[G]$ est donc un espace vectoriel de dimension finie. La dimension de $F[G]$ est égale à l'ordre de G.

En partant de la multiplication du groupe, l'opération de multiplication interne de $F[G]$ est définie par :

$$(\sum_{g\in G} a_g g)(\sum_{h\in G} b_h h) = \sum_{k\in G} (\sum_{gh=k} a_g b_h)k.$$

Avec cette opération et l'élément $1e$ (avec e l'élément neutre du groupe G), $F[G]$ est aussi un anneau unitaire. Il en découle que $F[G]$ est une F-algèbre.

Avec l'algèbre de groupe, la représentation d'un groupe fini G peut être vue comme un cas particulier de représentation d'algèbre. En effet, la restriction à G de toute représentation ρ de $F[G]$ fournit une F-représentation de G. Par ailleurs toute F-représentation de G définit une représentation de $F[G]$. En effet, soit ρ une F-représentation de G, la fonction σ définie par

$$\sigma(\sum_{g\in G} a_g g) = \sum_{g\in G} a_g \rho(g)$$

est une représentation de $F[G]$. Il en découle que la définition d'une F-représentation d'un groupe fini G est équivalente à la définition d'une représentation de $F[G]$.

Par ailleurs avec l'équivalence entre représentations et modules, les représentations des groupes finis vont correspondre aux modules sur l'algèbre de groupe. Ces modules sont appelés les G-modules. Des résultats sur les représentations des groupes finis comme le théorème de Maschke, que nous verrons dans la section suivante, sont généralement énoncés en terme de modules.

4.2.3 Théorème de Maschke

Nous venons de voir que la notion de représentation est équivalente à celle de module sur une algèbre. Ainsi l'étude des représentations d'un groupe fini G correspond à l'étude des modules sur l'algèbre $F[G]$ ou les G-modules. Un résultat important sur ces modules est le théorème de Maschke. Il établit que :

si la caractéristique de F ne divise pas le cardinal de G, alors tout $F[G]$-module (F-représentation de G) est semi-simple.

Preuve

Soit V un $F[G]$-module et W un sous-module de V. Le sous-module W est aussi un sous-espace de V. Comme nous sommes en dimension finie, il est possible de construire une projection F-linéaire de V sur W. Notons π cette projection. $Ker(\pi)$ est un F sous-espace de V mais pas forcément un $F[G]$ sous-module de V. Nous avons donc que $V = W \oplus Ker(\pi)$. L'idée de la preuve est de construire à partir de π une projection de V sur W dont le noyau soit un $F[G]$ sous-module de V. Soit la fonction $\theta : V \to V$ définie par :

$$\theta(v) = \frac{1}{|G|} \sum_{g\in G} \pi(vg)g^{-1}.$$

La fonction θ est linéaire et c'est une projection de V sur W. En effet, nous avons que :

$$\forall v \in V, \theta(v) = \frac{1}{|G|} \sum_{g \in G} \pi(wg)g^{-1} \in W,$$

puisque $\pi(vg) \in W$ et W est stable par l'action de G. Par ailleurs, nous avons que pour tout $x \in W$, $\pi(x) = x$. Ceci implique que :

$$\theta(w) = \frac{1}{|G|} \sum_{g \in G} \pi(wg)g^{-1} = \frac{1}{|G|} \sum_{g \in G} wgg^{-1} = \frac{1}{|G|} \sum_{g \in G} w = \frac{1}{|G|}|G|w = w.$$

Nous avons donc que $V = W \oplus Ker(\theta)$. Il suffit maintenant de montrer que $Ker(\theta)$ est un $F[G]$ sous-module de V. Si $h \in G$ et $v \in Ker(\theta)$ alors

$$\begin{aligned}
\theta(vh) &= \frac{1}{|G|} \sum_{g \in G} \pi(whg)g^{-1} \\
&= \frac{1}{|G|} \sum_{g \in G} \pi(whg)(hg)^{-1}h \\
&= \frac{1}{|G|} \sum_{g \in G} \pi(wg)g^{-1}h \\
&= \theta(v)h
\end{aligned}$$

puisque l'indexation par hg sur G est la même que l'indexation par g. Nous avons ainsi que $\theta(vh) = \theta(v)h = 0h = 0$. Ceci implique que $vh \in Ker(\theta)$ et donc $Ker(\theta)$ est un $F[G]$ sous-module de V. ■

Le théorème de Maschke établit que si la caractéristique du corps F ne divise pas $|G|$ alors toute F-représentation de G est semi-simple. Ceci est équivalent à dire que toute F-représentation de G se décompose en une somme finie de F-représentations simples. Le théorème de Maschke permet de ramener l'étude des F-représentations de G à celle des F-représentations simples. Ainsi, pour connaitre toutes les représentations d'un groupe fini donné il suffit de connaitre celles qui sont simples.

Remarques

L'énoncé du théorème de Maschke parle de la caractéristique du corps F. De façon générale la caractéristique d'un anneau unitaire A, ou son indicateur de torsion est le plus petit entier naturel non nul n tel que

$$\underbrace{1_A + 1_A + \ldots + 1_A}_{n \text{ termes}} = 0_A$$

avec 1_A et 0_A les éléments neutres de la multiplication et de l'addition de l'anneau A. Si un tel entier n'existe pas la caractéristique est alors égale à 0. Par exemple la caractéristique de l'anneau $\mathbb{Z}/n\mathbb{Z}$ est égale à n et celle de l'anneau $\mathbb{Z}$ est nulle.

Dans la preuve du théorème de Maschke et pour définir la projection θ nous divisons par $|G|$. Cette division est faite dans le corps F. En effet, l'entier naturel $|G|$ est injecté implicitement dans F avec la fonction

$$f(n) = \underbrace{1_F + 1_F + \ldots + 1_F}_{n\ termes}.$$

Donc, pour que la division par $f(|G|)$ dans F soit bien définie il faut que $|G|$ ne divise pas la caractéristique de F. Sinon la division par $f(|G|)$ dans F serait impossible et nous aurions que $f(|G|) = 0_F$.

4.2.4 Choix pour la formalisation

Modules sur une algèbre

Dans les sections précédentes nous avons vu que les modules et particulièrement les modules sur une algèbre, jouent un rôle important en théorie des représentations. Les modules sur une algèbre peuvent être définis à partir d'un espace vectoriel et de l'action d'une algèbre. En effet soit A une F-algèbre, l'ensemble $\{c1|c \in F\}$ noté $F.1$ est une sous algèbre de A. Cet ensemble correspond à l'injection de F dans A et permet de munir tout A-module d'une structure de F-espace vectoriel. Les modules sur une F-algèbre A peuvent donc être vus comme des F-espaces vectoriels avec des propriétés supplémentaires. En effet, soit V un F-espace vectoriel. Pour tous $v \in V$ et $x \in A$, soit $vx \in V$ une action à droite de A sur V. Pour que V soit un A-module il faut que pour tous $x, y \in A, v, w \in V$ et $c \in F$ les conditions suivantes soient respectées :

1. $(v + w)x = vx + wx$,
2. $v(x + y) = vx + vy$,
3. $(vx)y = v(xy)$,
4. $(cv)x = c(vx) = v(cx)$,
5. $v1 = v$

Dans cette présentation, l'action de A sur V est définie à droite pour éviter de la confondre avec l'action de F sur V. Dans la suite, la dimension d'un A-module va correspondre à la dimension du F-espace vectoriel correspondant.

Les modules sur une algèbre sont un cas spécial d'espaces vectoriels. Dans le cas de la dimension finie, ils héritent des espaces vectoriels des propriétés de décidabilité. Par

exemple, alors que l'appartenance d'un vecteur à un sous-module n'est pas toujours décidable dans le cas général, elle l'est dans le cas des sous-espaces. Comme les coefficients sont dans un corps, les algorithmes de résolution de système d'équations linéaires permettent de résoudre ces problèmes. Dans le contexte de notre développement, ces propriétés de décidabilité vont nous permettre d'utiliser la réflexion à petite échelle ainsi que les interfaces pour les types munis d'une égalité décidable et d'un opérateur de choix.

Composants de la théorie

La formalisation d'une théorie des représentations des groupes finis nécessite le développement de composants mathématiques pour la théorie des modules abstraits. Les principales structures algébriques dont nous avons besoin pour faire des raisonnements sur les représentations sont :

- **les espaces vectoriels**
- **les algèbres**
- **les modules sur une algèbre**

Ces structures sont définies d'une façon hiérarchique et sont des cas particuliers de modules sur un corps ou un espace vectoriel. En plus de ces structures, la théorie des sous-structures et morphismes associés est nécessaire pour énoncer et prouver formellement des définitions et résultats sur les représentations. Par exemple, la preuve du théorème de Maschke parle de sous-espaces, de sous-modules, d'applications linéaires et de projections. Au-dessus des structures algébriques listées ci-dessus, et pour travailler avec les représentations des groupes finis, nous avons besoin de la structure d'algèbre de groupe. Une représentation d'un groupe fini est un module sur l'algèbre de groupe associée ou un G-module. C'est un cas particulier de module sur une algèbre.

Notre approche consiste à formaliser une théorie générique des modules. Cette théorie va être construite sur les développements existants sur les égalités décidables, les opérateurs de choix, les structures algébriques et les matrices. Les représentations des groupes finis seront représentées comme des instances particulières de ces modules abstraits. Elles vont être définies en utilisant ce développement sur les modules abstraits et celui existant sur les groupes finis. L'avantage de cette approche est de pouvoir réutiliser les définitions et propriétés sur les modules pour des travaux de formalisation indépendants de la théorie des représentations. Le théorème de Maschke est un résultat sur les représentations qui est utile pour la preuve du théorème de Feit-Thompson. Il

servira donc comme premier test pour s'assurer que les définitions de notre développement fonctionnent ensemble. Dans les deux prochaines sections de ce chapitre nous présentons en détail notre travail de formalisation de la théorie des représentations.

4.3 Les modules

Dans cette section, nous présentons notre développement sur la théorie des modules. Comme nous l'avons vu dans la section précédente, la théorie des représentations est étroitement liée à celle des modules sur une algèbre. La motivation principale de ce développement est la mise en place de l'infrastructure nécessaire pour raisonner sur les représentations des groupes finis. Cependant il peut être utilisé dans des formalisations indépendantes et qui ne font pas appel nécessairement à la théorie des représentations.

Le développement sur les modules couvre les espaces vectoriels, les algèbres et les modules sur une algèbre. Ces structures algébriques linéaires sont traitées dans le cas de la dimension finie. Pour ces structures, la théorie des sous-structures associées a aussi été formalisée. Ceci couvre les sous-espaces, les sous-algèbres et les sous-modules. La formalisation comporte aussi une théorie des applications linéaires ou morphismes d'espaces vectoriels.

4.3.1 Hiérarchie d'algèbre linéaire

La première étape du développement sur les modules est la définition des types modélisant les structures algébriques manipulées. Comme nous l'avons vu dans la Section 1.2, dans Coq les structures algébriques abstraites peuvent être représentées par des *types enregistrements* ou types enrichis. Ces types correspondent à la combinaison d'un type des éléments, de constantes et opérations qui dépendent de ce type, et des propriétés sur ces constantes et opérations. Dans le cas des structures d'algèbre linéaire, les opérations vont dépendre d'un type externe : l'opération de multiplication externe d'un module dépend du type des éléments et du type de l'anneau. Par ailleurs, les structures algébriques sont définies suivant un schéma hiérarchique. Par exemple, un espace vectoriel est un groupe abélien qui est muni d'une action depuis un corps.

Nous avons développé une hiérarchie de structures algébriques linéaires. L'architecture de cette hiérarchie est donnée dans la Figure 4.1. Cette hiérarchie se base sur la bibliothèque `ssralg` du projet *Mathematical Components*, qui fournit une théorie des structures algébriques de base. Les structures définies dans `ssralg` comprennent entre autres les groupes abéliens, les anneaux et les corps. Notre hiérarchie d'algèbre linéaire étend celle de `ssralg` et définit les structures suivantes :

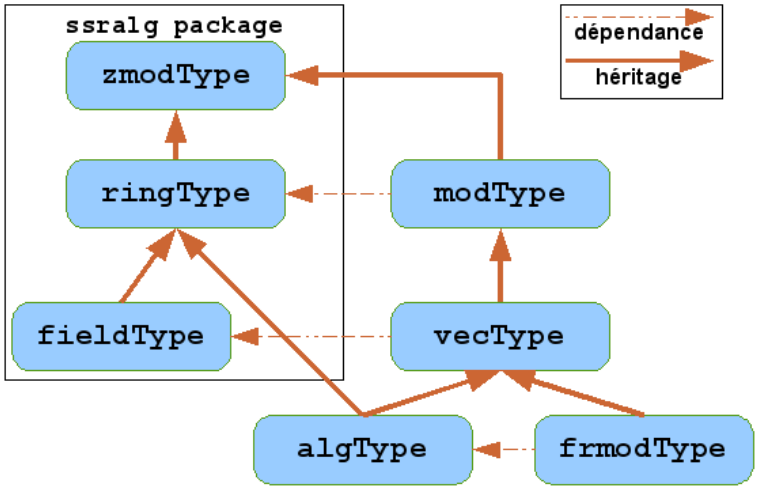


FIGURE 4.1 – Architecture de la hiérarchie pour l'algèbre linéaire.

- `modType` : les modules sur un anneau. C'est un groupe abélien avec une action d'anneau.
- `vecType` : les espaces vectoriels de dimension finie. C'est un module sur un corps avec une base génératrice.
- `algType` : les algèbres de dimension finie. C'est la combinaison d'un espace vectoriel de dimension finie et d'un anneau.
- `frmodType` : les modules finiment engendrés sur une algèbre de dimension finie. C'est un espace vectoriel de dimension finie avec une action d'algèbre.

Les structures listées ci-dessus ont été définies suivant un schéma de conception générique. Ce schéma est présenté dans les deux prochaines sous-sections.

Télescopes

Pour formaliser la hiérarchie de structures algébriques, notre première idée a été d'utiliser les *télescopes*. C'est une méthode standard pour formaliser une telle hiérarchie dans Coq [16, 43]. Pour définir les espaces vectoriels nous avons besoin des groupes abéliens et des corps. En considérant la définition du type des anneaux de la Figure 3.1 de la page 53, nous définissons un type pour les groupes abéliens. C'est un *type enregistrement* qui contient les opérations et propriétés de groupe abélien. Nous redéfinissons alors le type des anneaux comme un sous-type de celui des groupes abéliens. Les définitions des interfaces pour les groupes abéliens et les anneaux sont données dans la Figure 4.2. Dans cette nouvelle définition, les opérations d'addition et d'inversion, l'élément neutre de l'addition et les propriétés associées seront hérités de la structure `zmodType`

```
Module Zmodule.
Structure zmodType : Type := ZmodType {
 sort :> choiceType;
 zero : sort; add : sort → sort → sort; opp : sort → sort;
 (* Axiomes de groupe abélien *)
}.
End Zmodule.
Module Ring.
Structure ringType : Type := RingType {
 sort :> zmodType;
 one : sort; mul : sort → sort → sort;
 (* Axiomes d'anneau *)
}.
End Ring.
```

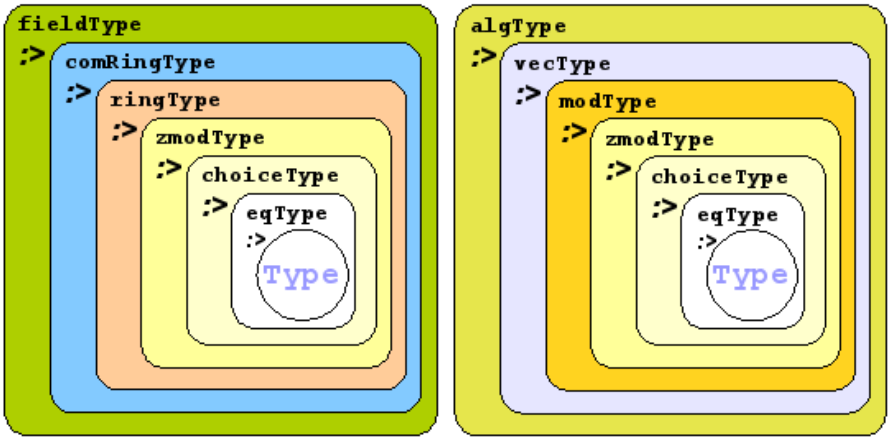
FIGURE 4.2 – Définitions des interfaces de groupe abélien et d'anneau.

qui définit le type des groupes abéliens. Le type des corps est lui défini au-dessus de celui des anneaux commutatifs suivant le même schéma que celui de la Figure 4.2. La même méthode est appliquée aussi pour définir les types des espaces vectoriels et celui des algèbres. Les Figures 4.3a et 4.3b donnent le schéma de la définition de cette hiérarchie avec l'utilisation des télescopes.

Dans la définition du type `zmodType` des groupes abéliens, le type des éléments de la structure est défini comme un `choiceType`. Le type des éléments des structures algébriques définies à partir de `zmodType`, comme les types `ringType` et `vecType`, sont aussi transitivement munis d'une structure de `choiceType`. Nous avons ainsi que le type des éléments d'un anneau ou d'un espace vectoriel est muni d'un opérateur de choix et d'une égalité décidable. Ce choix est motivé par le besoin de faire des raisonnements classiques sur les éléments des structures algébriques manipulées. En particulier, il nous permet de définir des sous-modules munis d'une égalité décidable et de Leibniz. Ceci est décrit par la suite dans la section sur les sous-espaces et les sous-modules.

En utilisant les *télescopes*, nous arrivons ainsi à définir une hiérarchie de structures algébriques. Par contre cette approche avec les *télescopes* pose quelques problèmes du point de vue de l'utilisation pratique. Avec les *télescopes*, les constantes de tête du type de représentation des structures de la hiérarchie sont égales. Par exemple, pour des types `F : fieldType` et `R : ringType`, les types des éléments correspondent explicitement aux types suivants :

```
EqType.sort (ChoiceType.sort (Zmodule.sort (Ring.sort
   (ComRing.sort (Field.sort F))))).
```



(a) *Télescopes* pour les corps. (b) *Télescopes* pour les algèbres.

FIGURE 4.3 – Hiérarchies des types avec les *télescopes*

```
EqType.sort (ChoiceType.sort (Zmodule.sort (Ring.sort R))).
```

Ceci pose des problèmes d'efficacité et de robustesse car les *coercions* et les structures canoniques pour un type donné sont déterminés à partir de sa constante de tête. Un autre problème d'efficacité se pose aussi au niveau de la comparaison de termes. En effet, l'algorithme de comparaison de termes de Coq est exponentiel par rapport à la longueur de la chaine de coercion. Ceci limite la capacité d'extension de la hiérarchie du point de vue de l'utilisation pratique.

Conception : *mixin*, *class* et *type*

Pour remédier aux problèmes avec les *télescopes*, un nouveau schéma de conception a été développé dans le cadre du projet *Mathematical Components*. Il est décrit dans [14] et fournit une nouvelle méthode pour définir une hiérarchie de structures algébriques ou combinatoires. Suivant ce schéma chaque structure algébrique va être définie en trois couches :

– ***mixin*** : une structure algébrique est en général définie en ajoutant de nouvelles propriétés et opérations à une structure algébrique moins riche. Ces propriétés et opérations qui permettent d'étendre une structure algébrique donnée vers une structure plus riche sont regroupées dans un enregistrement Coq appelé *mixin*. Par exemple, pour les modules le *mixin* va contenir l'action de l'anneau sur le groupe abélien et les propriétés correspondantes. Si la structure algébrique ou combinatoire est définie à partir d'une autre structure moins riche (ce qui est en

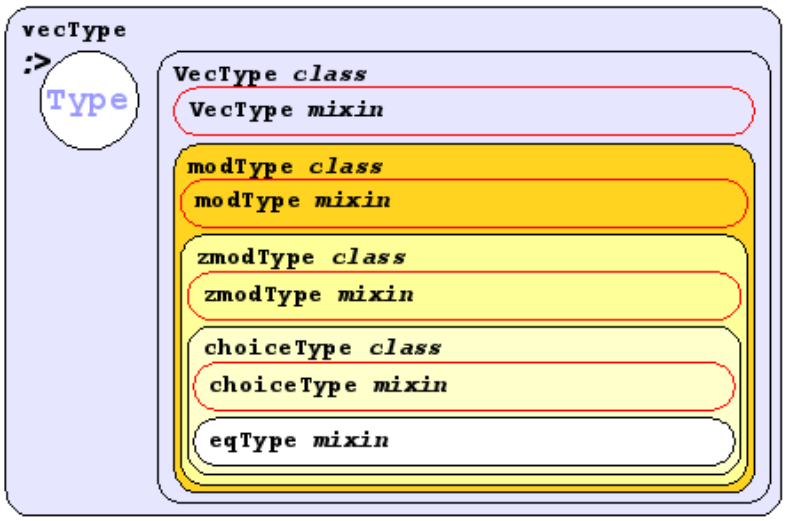


FIGURE 4.4 – Hiérarchie des *mixin* et *class* pour le type des espaces vectoriels.

général le cas), le *mixin* va dépendre de cette structure. Par exemple, le *mixin* des modules va être paramétré par un groupe abélien.

- ***class*** : c'est un enregistrement qui est paramétré par un type des éléments. Il regroupe toutes les propriétés et opérations associées à la structure algébrique. Il va être composé du *mixin* associé à la structure algébrique et de la *class* associé à la structure algébrique sous-jacente. Par exemple, la *class* des modules va être composé du *mixin* des modules et de la *class* des groupes abéliens. Dans certains cas, la structure algébrique est l'extension de deux structures algébriques données. Ceci est le cas pour les algèbres qui étendent les modules et les anneaux. La *class* associée va alors contenir, en plus du *mixin*, les *class* associées à ces structures sous-jacentes.
- ***type*** : c'est l'enregistrement qui regroupe ensemble le type des éléments et la *class* associés à la structure algébrique. La figure 4.4 donne l'exemple du schéma de définition du type des espaces vectoriels. Avec ce schéma de définition, le type des éléments de la structure algébrique est directement associé à `Type`. Ceci est fait avec les *télescopes*.

Les structures algébriques de la hiérarchie présentée dans la Figure 4.1 de la page 83 sont définies suivant le schéma de conception avec les *mixin*, *class* et *type*. La Figure 4.5 donne les enregistrements *mixin*, *class* et *type* de la structure de module sur un anneau.

Chaque structure est définie dans un `Module`. Ceci permet d'avoir des espaces de nom différents. Nous allons ainsi pouvoir utiliser les mêmes noms de structures ou

```
Module ModuleType.
Section ModuleTypeDef.
Variable R : ringType.
Structure mixin_of (V : zmodType) : Type := Mixin {
 mul : R → V → V;
 _ : forall u, mul 1 u = u;
 _ : forall a b u, mul a (mul b u) = mul (a * b) u;
 _ : forall a u v, mul a (u + v) = mul a u + mul a v;
 _ : forall a b u, mul (a + b) u = (mul a u) + (mul b u)
}.
Structure class_of (V : Type) : Type := Class {
 base :> GRing.Zmodule.class_of V;
 ext :> mixin_of (GRing.Zmodule.Pack base V)
}.
Structure type : Type := Pack {sort :> Type; _ : class_of sort }.
...
End ModuleTypeDef.
End ModuleType.
```

FIGURE 4.5 – Enregistrements *mixin*, *class* et *type* de la structure de module.

```
Variable F : fieldType.
Structure mixin_of (V : modType F) : Type := Mixin {
 basis : seq V;
 _ : forall v, exists s, v = \sum_(i < size basis) s`_i *: basis`_i;
 _ : forall s, \sum_(i < size basis) s`_i *: basis`_i = 0 ->
      forall i, i < size basis -> s`_i = 0
}.
```

FIGURE 4.6 – Enregistrement *mixin* de la structure d'espace vectoriel de dimension finie.

champs comme `mixin_of` ou `sort` dans la définition des types des différentes structures algébriques. Lors du passage entre niveau de structures, par exemple des modules vers les espaces vectoriels de dimension finie, c'est principalement le contenu du *mixin* qui change. La Figure 4.6 donne le *mixin* de la structure d'espace vectoriel de dimension finie. Ce *mixin* définit une liste de vecteurs `basis` qui génère tous les éléments du type et qui est libre. La Figure 4.7 donne le *mixin* de la structure de module sur une algèbre. Les composantes de cet enregistrement sont les opérations et propriétés données dans la Section 4.2.4. Dans cette Figure et dans la suite, la notation `*:` représente l'action standard de module ou multiplication externe. C'est la fonction `mul` du *mixin* des modules de la Figure 4.5. La notation `:*` va représenter l'action de l'algèbre sur le

```
Variable (K : fieldType) (A : algType K).
Structure mixin_of (M : vecType K) : Type := Mixin {
 mul : M → A → M;
 _ : forall x v w, mul (v + w) x = mul v x + mul w x;
 _ : forall x y v, mul v (x + y) = mul v x + mul v y;
 _ : forall x y v, mul (mul v x) y = mul v (x * y);
 _ : forall c v x, mul (c *: v) x = c *: (mul v x);
 _ : forall c v x, mul v (c *: x) = c *: (mul v x);
 _ : forall v, mul v 1 = v
}.
```

FIGURE 4.7 – Enregistrement *mixin* de la structure de module sur algèbre.

module. C'est la fonction `mul` du *mixin* des modules sur une algèbre de la Figure 4.7.

Suivant le schéma de conception avec les *mixin*, *class* et *type*, la déclaration d'un type donné comme une instance d'une structure algébrique se fait en deux étapes. Par exemple, une structure canonique de module est déclarée pour les matrices du chapitre précédent comme suit :

```
Definition matrix_modMixin := ModMixin (@scalemxA R m n)
 (@scale1mx R m n) (@scalemx_addr R m n) (@scalemx_addl R m n).
Canonical Structure matrix_modType := ModType matrix_modMixin.
```

En premier lieu, il faut construire une instance du *mixin* de la structure algébrique pour le type des éléments correspondant. Dans l'exemple ci-dessus, ceci est fait avec la fonction `ModMixin` qui est le constructeur de l'enregistrement *mixin* des modules. La définition du nouveau *mixin* suppose qu'une instance de la structure algébrique parente a déjà été déclarée pour le type des éléments. Dans l'exemple avec les matrices, ceci correspond à déclarer une structure canonique de `zmodType` sur le type `matrix`.

En second lieu, il faut construire avec les structures canoniques une instance de *type* pour le *mixin* défini dans la première étape. Dans l'exemple avec les matrices, ceci est fait avec la fonction `ModType` qui est un contructeur de l'enregistrement *type* pour les modules. Elle prend en argument un *mixin* des modules et infère la *class* associée.

Après la définition des types des structures algébriques linéaires, l'étape suivante du développement est la définition des types des sous-structures associées. Ce travail est présenté dans la prochaine section.

4.3.2 Algèbre des sous-espaces

La seconde étape du développement sur les modules est la formalisation d'une algèbre des sous-espaces. Les sous-structures associées aux structures algébriques auxquelles nous nous intéressons correspondent à des cas particuliers de sous-espaces vectoriels. En effet, les sous-algèbres sont des sous-espaces vectoriels avec en plus la propriété de clôture pour la multiplication interne. Les sous-modules d'algèbre sont aussi des sous-espaces avec en plus la propriété de clôture pour la multiplication externe. En théorie des représentations et en mathématique de façon générale, les relations et opérations entre sous-ensembles d'une structure algébrique sont utilisées fréquemment. Par exemple, dans la représentation des groupes finis, nous aurons besoin d'étudier les représentations des sous-groupes normaux d'un groupe donné. Ceci peut être ramené à des raisonnements sur les sous-algèbres et les sous-modules.

En théorie des types, les sous-structures algébriques sont souvent définies comme des propositions logiques. Dans Coq (par exemple dans [16] et [43]), ceci peut être fait avec un type enregistrement dépendant qui contient deux éléments : un prédicat propositionnel et une propriété de clôture. Par exemple, le type des sous-espaces d'un espace vectoriel donné peut être défini comme suit :

```
Structure sub_space K (V : vecType K) : Type := SubSpace {
 set :> V → Prop;
 _ : forall a u v, set u → set v → set (a *: u + v)
}.
```

Le problème avec cette représentation est que pour avoir une égalité de Leibniz entre les sous-espaces nous avons besoin de l'axiome d'extensionnalité. Comme nous travaillons en dimension finie, nous pouvons éviter cela et utiliser la correspondance entre sous-espaces et familles génératrices. Dans les espaces vectoriels de dimension finie, chaque sous-espace V possède une famille génératrice, c'est à dire une liste de vecteurs $\{v_1, v_2, \dots v_n\}$ de V tel que : $\forall x \in V, \exists \{a_1, a_2, \dots a_n\} : x = \sum_{i=1}^{n} a_i v_i$. Par ailleurs, chaque famille de vecteurs $\{v_1, v_2, \dots v_n\}$ définit un sous-espace vectoriel. C'est l'ensemble noté $< \{v_1, v_2, \dots v_n\} >$ des éléments qui peuvent s'écrire sous la forme $\sum_{i=1}^{n} c_i v_i$. Ainsi un sous-espace peut être représenté par une liste de vecteurs : ses générateurs. Les familles génératrices ou listes de vecteurs fournissent une représentation finie des sous-espaces. Cette représentation nous permet de définir une relation d'équivalence booléenne entre les sous-espaces. Avec l'interface `choiceType` et comme nous l'avons vu dans la Section 2.3.2, nous allons pouvoir définir une égalité de Leibniz sur les sous-espaces. Ceci va nous permettre de profiter de la puissance de réécriture de Coq.

Familles génératrices

Avant de définir le type des sous-espaces, nous avons commencé par formaliser une théorie des familles génératrices. Ce travail est une étape intermédiaire avant la formalisation d'une théorie des sous-espaces. L'objectif est de définir les relations et opérations sur les familles génératrices qui ne nécessitent aucune propriété particulière sur les listes de vecteurs. Dans la suite, pour un corps `K` et un espace vectoriel `vT : vecType K` sur ce corps, les vecteurs seront de type `vT` et les listes de vecteurs comme les familles de génératrices, seront de type `seq vT`.

Appartenance : une relation de base importante sur les familles génératrices est la relation d'appartenance. En dimension finie, la question d'appartenance d'un vecteur donné au sous-espace engendré par une famille génératrice se ramène à la résolution d'un système d'équations linéaires. Nous définissons un prédicat booléen d'appartenance à une famille génératrice par :

```
Definition mem_gf g v := v == \sum_(i < size g) (cf v g)‘_i *: g‘_i.
```

Dans cette définition, la fonction `cf` prend en argument un vecteur `v` et une liste de vecteurs `g`. Elle retourne une liste sur le corps `K`. Si le vecteur `v` est généré par `g`, `(cf v g)` est égale à la liste des coefficients de `v` dans la famille `g`. Sinon, elle est égale à la liste vide. La fonction `cf`, qui permet de résoudre un système d'équations linéaires, a été définie grâce aux développements sur les matrices, en particulier la décomposition LUP [14]. La spécification du prédicat `mem_gf` est donnée par le lemme :

```
Lemma mem_gfP : forall v g,
 reflect (exists s, v = \sum_(i < size g) s‘_i *: g‘_i) (mem_gf g v).
```

En notation mathématique standard, le lemme ci-dessus correspond à :

$$v \in < \{v_1, v_2, \dots v_n\} > \Leftrightarrow \exists \{a_1, a_2, \dots a_n\} : v = \sum_{i=1}^{n} a_i v_i.$$

Égalité : une autre relation importante sur les familles génératrices est la comparaison des sous-espaces engendrés. Deux familles génératrices sont égales si et seulement si les ensembles générés par ces familles sont égaux. Pour cela il suffit de tester que chacun des vecteurs de chacune des familles est généré par ceux de l'autre famille. Dans Coq, l'égalité entre familles génératrices est définie par les relations booléennes suivantes :

```
Definition gf_in g1 g2 := forallb i : ’I_(size g1), mem_gf g2 g1‘_i.
Definition gf_eq g1 g2 := gf_in g1 g2 && gf_in g2 g1.
```

La relation `gf_in` retourne `true` si tous les éléments de la famille `g1` appartiennent à l'ensemble généré par la famille `g2`. Ceci est équivalent à dire que l'ensemble généré par `g1` est inclus dans celui généré par `g2`. La relation d'égalité `gf_eq` est définie par la conjonction des inclusions.

Base : chaque sous-espace vectoriel possède une base ou une famille génératrice libre. C'est un ensemble de vecteurs $\{v_1, v_2, \dots v_n\}$ tel que :

$$\forall \{a_1, a_2, \dots a_n\}, \sum_{i=1}^{n} a_i v_i = 0 \Rightarrow \forall i, a_i = 0.$$

La dimension d'un sous-espace vectoriel est par définition la longueur d'une de ses bases. Toutes les bases d'un sous-espaces ont la même longueur. Une base donc correspond au plus petit nombre de vecteurs qui peuvent générer le sous-espace. Pour une famille génératrice quelconque, une base est donnée par le sous-ensemble de ses vecteurs générateurs qui ne peuvent pas être générés par les autres éléments de la liste des générateurs. Une base d'une famille génératrice peut être calculée par récurrence sur la taille de la famille. La fonction suivante permet de calculer une base pour une liste de vecteurs donnée :

```
Fixpoint basis_for (g : seq vT) :=
 if g is v :: s then
  if mem_gf s v then basis_for s else v :: (basis_for s)
 else [::].
```

Cette fonction retourne une liste de vecteurs libres qui génère le même ensemble que celui généré par la liste passée en argument. Nous disposons ainsi d'une fonction qui pour chaque famille génératrice donnée calcule une base. La définition de la fonction `basis_for` nous permet de dire que l'ensemble des listes de vecteurs qui génèrent le même sous-espace qu'une famille génératrice donnée contient au moins une liste de vecteurs libres. Nous utiliserons cette propriété lors de la définition du type des sous-espaces.

Opérations : dans les raisonnements sur les sous-espaces, nous avons besoin de combiner des sous-espaces pour former de nouveaux sous-espaces. Pour ces besoins, nous avons développé des fonctions qui permettent de calculer les générateurs correspondant aux opérations suivantes :

- la somme : pour deux sous espaces V et W, la somme noté $V + W$ correspond à l'ensemble des éléments qui peuvent s'écrire sous la forme $v + w$ avec $v \in V$ et $w \in W$. C'est aussi un sous-espace.

- l'intersection : pour deux sous-espace V et W, l'ensemble $V \cap W$ est un aussi sous-espace.
- le supplémentaire : pour deux sous-espace V et W tel que $W \subseteq V$, un supplémentaire de W par rapport à V, est un sous-espace U tel que $V = W + U$ et $W \cap U = 0$.

Comme nous représentons les sous-espaces par des familles de générateurs, nous avons besoin de construire les générateurs correspondants à ces opérations. Dans notre développement, ces fonctions sont définies pour des familles génératrices, donc des listes de vecteurs. Pour calculer des générateurs de la somme de deux familles génératrices, il suffit de concaténer les deux listes de générateurs. Le calcul de l'intersection et du supplémentaire n'est pas aussi simple. Le supplémentaire d'une famille génératrice par rapport à une autre est calculé avec la fonction récursive suivante :

```
Fixpoint gf_ext g1 g2 {struct g2} : seq vT * seq vT :=
 if g2 is a :: s then
  if mem_gf g1 a then ((gf_ext g1 s).1, a :: (gf_ext g1 s).2)
  else (a :: (gf_ext (a :: g1) s).1, (gf_ext (a :: g1) s).2)
 else ([::], [::]).
```

Cette fonction prend en paramètre deux listes de vecteurs `g1` et `g2`. Elle va décomposer la liste `g2` en deux sous-listes. Elle retourne un couple de liste ou un objet du type produit `seq vT * seq vT`. Dans la définition de la fonction `gf_ext`, les notations `.1` et `.2` permettent d'accéder respectivement à la première et à la seconde composante d'un objet de type produit. La première sous-liste `(gf_ext g1 g2).1` correspond aux vecteurs de `g2` qui ne sont pas inclus dans `g1` et libres entre eux. La seconde sous liste `(gf_ext g1 g2).2` correspond aux vecteurs restants. Si l'ensemble généré par `g1` est inclus dans celui généré par `g2`, alors la première sous-liste donne des générateurs du supplémentaire de `g1` par rapport à `g2`. En partant de cette fonction, nous définissons la fonction `inter_gf` qui calcul des générateurs de l'intersection de deux familles génératrices données.

Pour pouvoir raisonner avec ces fonctions, nous prouvons qu'elles respectent leurs spécifications. Par exemple, nous prouvons que la fonction `inter_gf` respecte sa spécification avec :

```
Lemma inter_gfP : forall v g1 g2,
 mem_gf (inter_gf g1 g2) v = (mem_gf g1 v) && (mem_gf g2 v).
```

Ce lemme prouve qu'un vecteur `v` est généré par l'application de `inter_gf` à deux familles de vecteurs `g1` et `g2` si et seulement si il est généré par chacune de ces deux familles.

Dans les précédents paragraphes nous avons présenté notre développement sur les familles génératrices ou listes de vecteurs. Comme nous représentons les sous-espaces par des listes de vecteurs, la définition des opérations algébriques entre sous-espaces nécessite la construction des listes de générateurs correspondants. Le développement sur les listes de générateurs est une étape intermédiaire. Il est utilisé pour définir les opérations algébriques sur le type des sous-espaces que nous présentons dans la prochaine section. C'est ce type que nous utilisons pour modéliser les sous-espaces vectoriels.

Sous-espaces

Les familles génératrices ou les listes de vecteurs permettent de représenter les sous-espaces vectoriels par un ensemble fini d'éléments. Par contre un sous-espace possède une infinité de familles génératrices. Ces familles ou listes génèrent le même sous-espace mais ne sont pas égales. Par exemple, dans $\mathbb{R}^3$ les listes $\{(1,0,0),(0,1,0)\}$ et $\{(0,2,0),(9,0,0)\}$ génèrent les mêmes sous-espaces mais ne sont pas égales. Pour avoir un représentant canonique pour les sous-espaces, la solution est de quotienter par l'égalité entre les sous-espaces engendrés.

Pour définir un type des sous-espaces, nous allons utiliser la méthode de définition d'un type quotient décrite dans la Section 2.3.2 du second chapitre. Le type des sous-espaces va contenir les représentants canoniques des classes d'équivalence par la relation `gf_eq`. Ces représentants canoniques, qui sont des listes de vecteurs, vont être définis en utilisant l'opérateur de choix `choose`. Comme nous définissons les sous-espaces sur des types munis d'un opérateur de choix, cet opérateur est automatiquement transféré (avec les structures canoniques) au type des listes de vecteurs ou familles génératrices. Par ailleurs nous voulons aussi que le représentant canonique retourné par l'opération de choix `choose` soit libre. Ceci permet de s'assurer que le représentant canonique est le plus petit ensemble de vecteurs qui permet de générer le sous-espace. Nous allons ainsi pouvoir définir la dimension d'un sous-espace comme la longueur de la liste de ses générateurs. Suivant la même méthode que dans 2.3.2, le type des sous-espaces est défini comme suit :

```
Variable (K : fieldType) (vT : vecType K).
Structure vspace : Type := VSpace {
```

```
 gf :> seq vT;
 _ : gf == choose [pred x | freeb x && gf_eq gf x ] (basis_for gf)
}.
```

Dans la définition ci-dessus, `freeb` est un prédicat booléen qui permet de vérifier si les vecteurs d'une liste donnée sont libres. L'opérateur de choix `choose` est utilisé ici avec le prédicat `[pred x | freeb x && gf_eq gf x ]` qui représente l'ensemble des familles génératrices qui sont libres et égales à la familles `gf`. La fonction `basis_for` nous permet de dire que cet ensemble est non vide. En effet, elle retourne une famille génératrice qui est libre et égale à la familles `gf`. C'est un témoin qui nous permet de dire que le résultat de l'application de l'opérateur de choix est un représentant canonique qui est aussi libre.

Les *télescopes* sont utilisés pour définir une *coercion* du type `vspace` vers le type `seq vT` des listes sur l'espace vectoriel global `vT`. Ceci va nous permettre d'utiliser les opérations sur les listes de vecteurs définies précédemment. Dans la définition du type `vspace`, la propriété que la liste des générateurs est canonique permet de définir une structure de `eqType` sur les sous-espaces. Nous avons ainsi une égalité de Leibniz sur le type `vspace` qui est équivalente à la relation d'équivalence `gf_eq` sur les familles génératrices.

Prédicat booléen : les listes de vecteurs fournissent une représentation finie des sous espaces qui permet de définir une égalité décidable. Les sous-espaces vectoriels correspondent aussi à des sous ensembles de l'espace vectoriel global. En théorie des types, ils peuvent être représentés par des prédicats booléens sur le type des éléments de l'espace vectoriel global. Nous utilisons les *coercions* Coq pour définir une interprétation ensembliste des sous-espaces. Ceci permet de munir le type `vspace` d'une couche présentation qui va correspondre à un prédicat booléen. La *coercion* du type `vspace` vers le type `pred vT` est définie comme suit :

```
Coercion pred_of_vs :=
 (fun F (vT : vecType F) (V : vspace vT) => mem_gf V : _ -> _).
```

Ainsi nous allons pouvoir utiliser les notations et définitions génériques sur les prédicats booléens définis dans la Section 2.1. Par exemple, les propriétés de clôture des sous-espaces pour les opérations vectoriels sont données par les lemmes suivant :

```
Variable (V : vspace vT).
Lemma in_vs_add : forall u v, u \in V → v \in V → (u + v) \in V.
Lemma in_vs_mul : forall c v, v \in V → c *: v \in V.
```

Dans ces lemmes, la notation `\in` représente la fonction `mem_gf` qui vérifie l'appartenance d'un vecteur à l'ensemble généré par une liste de vecteurs. C'est une instance de la notation pour les prédicats booléens donnée dans la Section 2.1 du second chapitre.

L'interprétation sous forme de prédicat fournit une méthode plus intuitive pour manipuler les propriétés des sous-espaces. C'est la représentation standard dans la littérature mathématique. Dans notre développement, nous utilisons les listes pour représenter les sous-espaces principalement pour avoir une représentation canonique. Ceci est plus pratique du point de vue des mathématiques sur ordinateur puisqu'il permet d'avoir une représentation finie et unique. Ces deux interprétations sont équivalentes et complémentaires. Le lemme suivant fournit l'équivalence :

```
Lemma eq_vsP : forall V1 V2,
 (forall v, v \in V1 = v \in V2) ↔ (V1 = V2).
```

Avec ce lemme, prouver que deux sous-espaces sont égaux revient à prouver qu'ils génèrent les mêmes sous-ensembles. L'égalité à gauche est celle entre booléens ; alors que celle de droite est entre sous-espaces.

Construction : nous représentons un sous-espace par une liste de générateurs. Dans la définition du type `vspace`, la liste de générateurs est supposée être canonique. La fonction `VSpace` permet de construire un objet du type `vspace`. Elle prend en paramètre une preuve que la liste de générateurs est canonique. Les autres paramètres seront inférés automatiquement par Coq. D'un point de vue utilisateur, ceci n'est pas pratique puisqu'il fait appel à des propriétés de la fonction `choose`. Puisque chaque liste de générateurs définit un sous-espace unique, une approche plus simple est de définir un constructeur du type `vspace` à partir d'une liste de générateurs quelconque. Cette liste de générateurs peut ne pas être canonique ou libre. La construction de la liste canonique et libre associée va être prise en charge automatiquement par le constructeur. Ceci est fait en utilisant les propriétés de l'opérateur de choix et de la fonction de construction d'une base. En effet, pour une liste de générateurs quelconque, nous pouvons construire une base avec la fonction `basis_for`. Par ailleurs, l'opérateur `choose` retourne toujours le même résultat pour des prédicats équivalents. Ces propriétés nous permettent de prouver le lemme suivant :

```
Lemma make_vs_proof : forall (g : seq vT) ,
 let s := choose [pred x | freeb x && gf_eq x g ] (basis_for g) in
  s == choose [pred x | freeb x && gf_eq x s ] (basis_for s).
```

Le lemme `make_vs_proof` prouve que pour toute liste de vecteurs, nous pouvons construire une liste de générateurs (la liste `s`) qui respectent la condition de canonicité spécifiée dans la définition du type `vspace`. Ainsi, nous pouvons définir la fonction :

```
Definition make_vs (s : seq vT) := VSpace (make_vs_proof s).
```

Cette fonction prend un paramètre une liste de générateurs et retourne le sous-espace associé. Elle fournit un second constructeur pour le type `vspace`. Ce constructeur est plus pratique puisqu'il ne fait appel qu'à la notion de liste.

Opérations : le type `vspace` est un nouveau type qui est différents du type `seq vT` des listes sur l'espace vectoriel global. Les opérations standards sur les sous-espaces font appellent à la liste des générateurs. Comme nous l'avons vu dans la section sur les familles génératrices, ces opérations peuvent être définies sur le type `seq vT`. Pour que le résultat de ces opérations soit dans le nouveau type `vspace`, elles sont redéfinies en utilisant le constructeur `make_vs` du type des sous-espaces. En particulier, nous définissons les fonctions et notations Coq suivantes :

```
Definition vspace_sum V1 V2 := make_vs (V1 ++ V2).
Definition vspace_I V1 V2 := make_vs (inter_gf V1 V2).
Definition vspace_S V := make_vs (gf_ext V (vspace_set (basis vT))).1.
Definition vspace_set g := make_vs (g : seq vT).
Definition vspace_dim V := size V.
Definition subvs V1 V2 := gf_in V1 V2.
(* redéfinition de la fonction des coefficients dans vspace *)
Definition coef v V := (cf v V).

Notation "V1 + V2" := (vspace_sum V1 V2). (* somme *)
Notation "V1 :&: V2" := (vspace_I V1 V2). (* intersection *)
Notation "~: V" := (vspace_S V). (* supplémentaire *)
(* sous-espace engendré par une liste *)
Notation "<< g >>" := (vspace_set g).
(* sous-espace engendré par un vecteur *)
Notation "<[ v ]>" := (<<[:: v]>>).
Notation "0" := (<[ 0 ]>). (* sous-espace nul *)
Notation "'dim ( V )" := (vspace_dim V). (* dimension *)
Notation "V \subvs W" := (subvs V W). (* inclusion *)
```

Des propriétés logiques sont prouvées pour ces opérations. Par exemple, nous prouvons les résultats suivants :

```
Lemma sum_vsP : forall v V1 V2,
 reflect (exists2 u, u \in V1 & (v - u) \in V2) (v \in (V1 + V2)).
Lemma dim_sum : forall V W,
 'dim(V + W) = 'dim(V) + 'dim(W) - 'dim(V :&: W).
Lemma sumvsC_prj : forall V v, exists w, (w \in V) && (v - w \in ~: V).
```

Le premier lemme donne une caractérisation de la propriété d'appartenance à la somme de sous-espaces. Le second lemme donne une relation entre la dimension de la somme et la somme des dimensions. Le dernier lemme fournit une propriété du sous-espace supplémentaire. Il prouve que pour tout sous-espace `V` et un élément `v` de l'espace global, `v` peut être décomposé en une somme d'un élément de `V` et d'un élément du supplémentaire de `V`. Ce lemme permet en particulier de définir la fonction de projection sur un sous-espace.

L'opération de sommation des sous-espaces peut être munie d'une structure de monoïde. En effet, elle est associative et possède un élément neutre : le sous-espace nul. Afin de pouvoir utiliser les définitions et preuves génériques sur les opérations indexées, nous déclarons une structure canonique de `monoid` pour l'opération `vspace_sum` :

```
Canonical Structure vs_sum_monoid := @Monoid.Law _ vspace_sum ...
```

Après cette déclaration, nous énonçons et prouvons le lemme suivant :

```
Lemma vs_sum_nth : forall V, V = \sum_(i < 'dim(V)) <[ V`_i]>.
```

Ce lemme donne la décomposition d'un sous-espace sous la forme d'une somme des sous-espaces engendrés par ses générateurs.

En plus des opérations standard sur les sous-espaces, nous définissons une relation booléenne pour décrire que deux sous-espaces sont en somme directe par rapport à un sous-espace donné.

```
Definition dsum V V1 V2 := (V == V1 + V2) && (V1 :&: V2 == 0).
```

La fonction `dsum` prend en paramètre trois sous-espaces `V`, `V1` et `V2`, et retourne une valeur booléenne. Elle vérifie que l'intersection de `V1` et `V2` est nulle, et que `V1+V2` est égale à `V`. Ces propositions sont exprimées sous une forme booléenne avec la notation `==` de l'égalité décidable.

Après la définition du type des sous-espaces, la prochaine étape est la formalisation d'une théorie des sous-algèbres et des sous-modules. Ces sous-structures sont des cas particuliers de sous-espaces. Elles jouent un rôle important dans la théorie des représentations. Dans les deux prochaines sous-sections, nous allons présenter une formalisation de la théorie des sous-algèbres et des sous-modules.

Sous-algèbres

Une sous-algèbre est un sous-espace qui en plus possède une propriété de clôture par rapport au produit interne de l'algèbre. Le sous-espace doit aussi contenir l'élément neutre de la multiplication interne de l'algèbre. Ces propriétés correspondent à la proposition suivante :

```
Variable (K : fieldType) (aT : algType K).
Definition is_algebra (V : vspace aT) :=
 (1 \in V) /\ (forall u v, u \in V → v \in V → u * v \in V).
```

Dans le cas de la dimension finie, cette propriété peut être définie par un prédicat booléen, un objet de type `bool`. En effet, pour que le produit de tous les éléments du sous-espace engendré soit inclus dans le sous-espace, il suffit que ceci soit vrai pour les éléments de la liste des générateurs. Comme cette liste contient un nombre fini d'éléments, nous pouvons utiliser le quantificateur universel booléen `forallb` que nous avons vu dans la Section 2.3.4 du second chapitre. Le prédicat exprimant qu'un sous-espace est aussi une sous-algèbre est alors défini comme suit :

```
Definition algebra_b (V : vspace aT) := (1 \in V) &&
 forallb i : 'I_'dim(V), forallb j : 'I_'dim(V), V`_i * V`_j \in V.
```

Le prédicat `algebra_b` et la proposition `is_algebra` sont équivalents et complémentaires. Le premier retourne un objet de type `bool` qui pourra être utilisé dans les définitions de fonction. En plus il nous permet de profiter de la propriété de *proof-irrelevance* sur les booléens. En effet, nous allons avoir pour tout sous-espace `V`, les preuves booléennes `(A : algebra_b V)` et `(B : algebra_b V)` sont égales : `A = B`. Ceci n'est pas valide pour les preuves `(P : is_algebra V)` et `(Q : is_algebra V)` qui sont dans `Prop`. Par contre ces preuves de la proposition `is_algebra V` sont plus facile à manipuler dans le contexte d'une preuve Coq. Ces deux vues de la notion de sous algèbre sont équivalentes. Cette équivalence est donnée par le *lemme de vue* suivant :

```
Lemma algebraP : forall V, reflect (is_algebra V) (algebra_b V).
```

Après ces définitions, nous déclarons le type des sous-algèbres comme un sous type de celui des sous-espaces.

```
Structure algebra : Type := mkAlgebra {
 alg_vs :> vspace aT; _ : algebra_b alg_vs}.
```

Le type `algebra` est un enregistrement qui contient deux composantes : un sous-espace et une preuve que ce sous-espace respecte les axiomes associés à la structure de sous-algèbre. Ces axiomes sont exprimés par le prédicat booléen `algebra_b`. Grâce à la

propriété de *proof-irrelevance*, la comparaison entre des objets de type `algebra` va correspondre à une comparaison entre les sous-espaces sous-jacents.

Le constructeur par défaut du type `algebra` des sous-algèbres fait appel au prédicat booléen `algebra_b`. Nous définissons un second constructeur pour ce type qui fait appel au prédicat propositionnel `is_algebra`. Ce dernier est plus facile à manipuler puisqu'il ne fait appel qu'aux quantificateurs universels standards de Coq. Le nouveau constructeur est donné par la fonction suivante :

```
Definition mk_algebra V (VP : is_algebra V) :=
 mkAlgebra (introT (algebraP V) VP).
```

Cette fonction prend en entrée une proposition logique (objet de type `Prop`) et non pas un prédicat booléen comme le constructeur par défaut `mkAlgebra`. Elle fournit une méthode pour construire une sous-algèbre à partir d'une preuve de la proposition `is_algebra`.

Opérations : le type des sous-algèbres est un sous type de celui des sous-espaces. Grâce aux *coercions*, les opérations sur les sous-espaces peuvent donc être appliquées aux sous-algèbres. Par contre ces opérations ne retournent pas toujours une sous-algèbre. En particulier, la somme de deux sous-algèbres n'est pas une sous-algèbre. D'autres opérations comme l'intersection sont closes pour les sous-algèbres. Nous prouvons que l'intersection de deux sous-algèbres est aussi une sous-algèbre avec le lemme suivant :

```
Lemma is_algebraI : forall A1 A2, is_algebra (A1 :&: A2).
```

L'opération d'intersection est alors redéfinie avec les structures canoniques.

```
Canonical Structure algebraI A1 A2 := mk_algebra (is_algebraI A1 A2).
```

Cette déclaration permet au système d'inférer automatiquement la structure de sous-algèbre au sous-espace correspondant à l'intersection de deux sous-algèbres. Par exemple, soit le lemme suivant :

```
Variable A : algebra aT.
Lemma in_alg_mul : forall u v, u \in A → v \in A → u * v \in A.
```

Ce lemme permet de dire que le produit de deux éléments d'une sous-algèbre `A` reste dans `A`. Dans l'énoncé du lemme `A` est un objet de type `algebra`. Grâce à la déclaration de structure de canonique pour `algebraI`, nous pouvons appliquer le lemme `in_alg_mul` à `(A1 :&: A2)` avec `A1` et `A2` des sous-algèbres. L'utilité des structures canoniques vient du fait que `(A1 :&: A2)` est de type `vspace` alors que `in_alg_mul` s'attend à un objet

de type `algebra`. Les structures canoniques permettent ici d'inférer automatiquement la structure de sous-algèbre correspondante.

Sous-modules

La correspondance entre représentations et modules sur une algèbre permet de définir une notion de sous-représentation. Les sous-modules vont correspondre aux sous-représentations. Un sous-module est un sous-espace qui est stable par l'action de l'algèbre sur le module. Cette définition peut être généralisée en considérant la propriété de stabilité seulement pour les éléments d'une sous-algèbre donnée. La proposition suivante définit cette propriété :

```
Variable (F : fieldType) (aT : algType F) (mT : frmodType aT).
Definition stable (A : algebra aT) (V : vspace mT) :=
 forall x v, x \in A → v \in V → v :* x \in V.
```

Comme pour les sous-algèbres et vu que nous sommes en dimension finie, la proposition `stable` peut être exprimée avec un prédicat booléen. En effet, il suffit de vérifier que le produit des générateurs du sous-espace par ceux de la sous-algèbre est stable. Nous définissons les sous-modules comme un sous-ensemble des sous-espaces avec le prédicat suivant :

```
Definition module (A : algebra aT) : pred (vspace mT) := [pred V |
 forallb i : 'I_'dim(A), forallb j : 'I_'dim(V), V`_j :* A`_i \in V].
```

Le prédicat `module` définit l'ensemble des sous-espace sur le type `mT` qui sont stables par l'action de la sous-algèbre `A` et donc des sous-modules par rapport à `A`.

Nous caractérisons les sous-modules par un prédicat et non par un type comme nous l'avons fait pour les sous-algèbres. Ce choix est motivé par des raisons pratiques. Comme nous définissons les sous-modules par rapport à une sous-algèbre, une définition par un type ferait qu'un type des sous-modules serait dépendant de celui des sous-algèbres. Un changement de sous-algèbre va donc impliquer un changement de type. Ainsi les sous-modules associés à un sous-espace donné par rapport à des sous-algèbres différentes ne seront pas du même type. Pour comparer de tels sous-modules il faut définir des fonctions de conversion qui dépendent de conditions ou preuves sur les sous-algèbres correspondantes. Comme les sous-modules sont aussi des sous-espaces, il est plus simple de les définir sous forme de prédicat. Lors d'un changement de sous-algèbre, nous n'aurons pas de changement de type. Ainsi, nous n'aurons pas besoin de définir de nouvelles fonctions de conversion entre types.

Propriétés : nous définissons et prouvons des notions et résultats associés aux sous-modules. En particulier, la simplicité et la semi-simplicité des modules, que nous avons définies dans la Section 4.2.2, sont représentées par les propositions suivantes :

```
Definition simple A (V : vspace mT) := module A V /\
 forall W, W \subvs V → module A W → (W = 0) \/ W = V.
Definition semisimple A (V : vspace mT) := module A V /\
 forall W, W \subvs V → module A W →
  exists U, module A U /\ U \subvs V /\ dsum V W U.
```

Les propositions ci-dessus sont définies dans le type `Prop`. Elles ne sont pas définies dans `bool` puisque ceci nécessite que le corps des coefficients de l'espace vectoriel soit muni de certaines propriétés de décidabilité. En particulier le corps des coefficients doit être muni d'une théorie décidable du premier ordre. Dans une première étape du développement la définition dans `Prop` est suffisante puisque nous n'allons utiliser ces propositions que dans des preuves. La définition d'une version équivalente de ces propositions dans `bool` reste possible dans le cadre d'une évolution du développement.

En plus de la définition des propriétés logiques sur les sous-modules, nous prouvons aussi certaines propriétés sur ces structures. Par exemples, les lemmes suivant permettent de dire que la somme et l'intersection de deux sous-modules restent des sous-modules.

```
Lemma moduleS : forall A V W,
 module A V -> module A W -> module A (V + W).
Lemma moduleI : forall A V W,
 module A V -> module A W -> module A (V :&: W).
```

Ces lemmes prouvent que l'ensemble des sous-modules est clos par les opérations de sommation et d'intersection des sous-espaces. Par contre ceci n'est pas toujours le cas pour le supplémentaire. Pour construire un supplémentaire d'un sous-module il faut construire une application linéaire spéciale. Dans la prochaine section, nous allons présenter notre développement pour formaliser une théorie des morphismes entre espaces vectoriels ou applications linéaires.

4.3.3 Morphismes : applications linéaires

La troisième étape du développement sur les modules est la définition d'une théorie des morphismes entre ces structures. Ces morphismes sont des cas particuliers d'applications linéaires entre espaces vectoriels. Dans un premier temps, nous ne nous intéressons

qu'aux applications linéaires. Comme dans le cas des fonctions à support fini, les matrices et les sous-espaces, nous allons définir deux points de vue pour les applications linéaires. Ces deux points de vue vont nous permettre d'avoir des propriétés classiques comme l'axiome d'extensionnalité sur ces fonctions. Elle nous permettrons aussi d'avoir une égalité décidable sur les applications linéaires.

Type de représentation

Avec les définitions des sections précédentes sur le type des espaces vectoriels, la propriété de F-linéarité 4.1 de la page 75 correspond en Coq à la définition suivante :

```
Variables (F : fieldType) (V W : vecType F).
Definition linear (f : V → W) :=
 forall a u v, f (a *: u + v) = a *: f u + f v.
```

La fonction `linear` définit un prédicat propositionnel sur le type des fonctions. Cette proposition permet de caractériser les applications linéaires. Elle fournit une représentation fonctionnelle de ces morphismes. Un type des applications linéaires peut être défini par :

```
Variables (F : fieldType) (V W : vecType F).
Structure linear_type : Type := Linear {map :> V → W; _ : linear map}.
```

Cette représentation fonctionnelle est facile à manipuler dans le contexte de Coq puisque les fonctions sont des objets primitifs du système. Par contre, avec cette représentation il n'est pas possible d'avoir une égalité de Leibniz entre les applications linéaires sans admettre l'axiome d'extensionnalité. Par conséquent, nous n'allons pas pouvoir définir la structure d'espace vectoriel associée aux applications linéaires avec cette axiomatisation.

Définition du type : en dimension finie, chaque application linéaire définit une matrice et inversement chaque matrice définit une application linéaire. Les matrices fournissent donc une représentation des applications linéaires. Cette représentation correspond à une description finie pour décrire une application linéaire. Ceci est plus pratique du point de vue de l'encodage sur machine. Avec cette représentation, le type des applications linéaires va hériter de l'égalité de Leibniz à partir du corps des coefficients des espaces vectoriels. Nous définissons le type des applications linéaires entre deux espaces vectoriels de dimension finie comme un cas spécial de matrices.

```
Variable (F : fieldType) (V W : vecType F).
Inductive linear_map : Type :=
```

```
 LinearMap of (matrix F 'dim(<<basis W>>) 'dim(<<basis V>>)).
Definition lmap_mx f := let LinearMap M := f in M.
```

La fonction `lmap_mx` retourne la matrice associée à une application linéaire. Dans la définition du type `linear_map`, `(<<basis W>>)` et `(<<basis V>>)` correspondent respectivement aux sous-espaces engendrés par les listes `(basis W)` et `(basis V)`. Ce sont les sous-espaces maximaux pour les types `W` et `V` : chaque élément des types `W` et `V` va appartenir à ces sous-espaces. Ceci vient de la définition du *mixin* de `vecType` de la Figure 4.6 de la page 87.

Interprétation fonctionnelle : le type `linear_map` va nous permettre de représenter la notion d'application linéaire. Il va hériter des structures de `eqType` et `choiceType` du type `matrix`. Pour interpréter les objets du type `linear_map` comme des fonctions, nous définissons une *coercion* comme suit :

```
Definition fun_of_lmap f v :=
 \sum_(i < 'dim(<<basis W>>)) (\sum_(j < 'dim(<<basis V>>))
   (coef v <<basis V>>)`_j * lmap_mx f i j) *: <<basis W>>`_i.
Coercion fun_of_lmap : linear_map >-> Funclass.
```

Avec cette fonction, nous allons pouvoir écrire `f v` pour `f : linear_map` et `v : V`. Ceci va être interprété par Coq comme `(fun_of_lmap f v)`. Par ailleurs, nous définissons un constructeur pour le type `linear_map` qui prend en paramètre une fonction entre les types `V` et `W`. Il construit la matrice correspondant à l'image des éléments de la base de `V` par rapport à la base `W`.

```
Definition lmap_of_fun (f : V -> W) :=
 LinearMap (\matrix_(i < 'dim(<<basis W>>), j < 'dim(<<basis V>>))
  (coef (f (<<basis V>>)`_j) <<basis W>>)`_i).
```

Cette fonction fournit un moyen pour construire des objets du type `linear_map`. Elle permet de rendre transparent à l'utilisateur la manière dont est codé le type des applications linéaires. Par contre, pour que l'application linéaire construite à partir d'une fonction donnée soit équivalente à celle-ci, il faut que la fonction de départ soit linéaire. Ceci est prouvé avec le lemme suivant :

```
Lemma lmapE : forall (f : V -> W), linear f -> lmap_of_fun f =1 f.
```

Dans le lemme ci-dessus, la *coercion* `fun_of_lmap` est implicitement introduite. Le terme `(lmap_of_fun f)` correspond en fait à `fun_of_lmap (lmap_of_fun f)`. La notation `=1` correspond à l'égalité extentionnelle entre fonctions. Grâce à notre représentation des applications linéaires par des matrices, nous avons que dans le type `linear_map`

l'égalité extensionnelle est équivalente à l'égalité de Leibniz. Cette propriété est donnée par le lemme suivant :

```
Lemma lmapP : forall f1 f2 : linear_map, f1 =1 f2 <-> f1 = f2.
```

Les définitions et preuves données ci-dessus fournissent une double interprétation des applications linéaires. L'interprétation matricielle nous permet de coder les applications linéaires et avoir une égalité de Leibniz entre elles. L'interprétation fonctionnelle fournit un moyen pratique pour manipuler ces applications.

Noyau et projection

Dans notre développement sur les applications linéaires, nous définissons le sous-espace correspondant au noyau d'une application linéaire. Pour une application linéaire `f : linear_map`, `('ker f)` est le sous-espace des éléments de `V` qui s'envoie sur 0 par `f`. Le lemme suivant prouve la propriété d'appartenance à ce sous-espace :

```
Lemma kerP : forall v, reflect (f v = 0) (v \in ('ker f)).
```

Nous définissons aussi la projection sur un sous-espace donné. Pour un sous-espace `U : (vspace V)`, la fonction `(proj U)` représente la projection linéaire sur `U`. C'est une application linéaire de type `linear_map V V`. Elle a les propriétés suivantes :

```
Lemma proj_mem : forall v, (proj U) v \in U.
Lemma proj_val : forall v, v \in U -> (proj U) v = v.
```

Avec ces définitions et preuves, nous disposons d'une théorie formelle des applications linéaires. Ce sont les types de base pour les morphismes associés aux structures algébriques auxquelles nous nous intéressons. Ce développement sur les applications linéaires, ainsi que les développements sur les structures d'algèbres linéaires et les sous-structures associées, constitue l'infrastructure de base nécessaire pour le travail sur la théorie des représentations des groupes finis. Avec la correspondance entre représentations et modules, les G-modules ou modules sur une algèbre de groupe modélisent les représentations des groupes finis. Ce sont des instances particulières des modules sur algèbre. Dans la prochaine section, nous présentons notre travail de formalisation de la théorie des G-modules.

4.4 Les G-modules

La principale motivation de notre développement sur les modules est de disposer de l'infrastructure nécessaire pour travailler avec les représentations des groupes finis.

Comme nous l'avons déjà vu, ces représentations correspondent à des modules sur l'algèbre d'un groupe G ou G-modules. La définition d'une représentation d'un groupe fini nécessite la définition de l'algèbre du groupe. Cette définition et la formalisation du théorème de Maschke vont nous permettre de tester l'intégration de notre développement sur les modules avec le développement sur les groupes finis [21] de SSReflect. Elles correspondent aussi à l'infrastructure de base pour faire la théorie des représentations des groupes finis.

4.4.1 Algèbre de groupe

Définitions

Dans la Section 4.2.1 nous avons vu que la structure de l'algèbre de groupe est définie par une somme formelle. Pour un corps F et un groupe fini G, nous avons que les éléments de l'algèbre de groupe s'écrivent sous la forme :

$$\sum_{g \in G} a_g g \ ; \ a_g \in F.$$

Les éléments de l'algèbre de groupe sont donc déterminés par les coefficients $\{a_{g \in G}\}$. Ils peuvent être représentés par une fonction de G vers le corps F. Comme G est fini, ces fonctions sont à support fini.

En partant du développement sur les fonctions à support fini et les groupes finis de SSReflect, nous représentons le type des éléments d'une algèbre de groupe par celui des fonctions du groupe vers le corps.

```
Variables (F : fieldType) (G : finGroupType).
Notation FG := {ffun G -> F}.
```

Ici nous ne définissons pas de nouveau type pour représenter les algèbres de groupes. Ceci est dû au fait que la structure qui nous intéresse est l'instance du type des algèbres (`algType`) que nous aurons à définir par la suite. Il n'est donc pas nécessaire de définir un nouveau type. Dans la déclaration ci-dessus, `finGroupType` représente le type des groupes finis. C'est un sous-type de `finType` le type des types finis. Nous avons donc que `G` est un groupe fini et aussi un type fini. Le terme `{ffun G -> F}` correspond au type des fonctions à support fini de `G` dans `F`.

Nous avons vu précédemment que chaque élément du groupe peut être injecté dans l'algèbre de groupe. Nous définissons cette injection par la fonction Coq suivante :

```
Definition injG (g : G) : FG := [ffun k => if k == g then 1 else 0].
```

Dans la définition ci-dessus, nous utilisons le constructeur générique `[ffun g => E]` des fonctions à support fini. Le `0` et le `1` correspondent aux éléments du corps `F`. Avec cette injection, le groupe fini définit une base de l'algèbre. Dans la suite, pour `g : gT` la notation `g %:FG` va correspondre à `(injG g)`. Par exemple, l'injection de l'élément neutre du groupe `G` est notée `(1%g %:FG)`.

Pour définir une instance du type `algType` pour l'algèbre de groupe, nous définissons les opérations d'espaces vectoriels et d'anneaux associées.

```
Definition opprgA (v : FG) : FG := [ffun g => - (v g)].
Definition addrgA (v1 v2 : FG) : FG := [ffun g => v1 g + (v2 g)].
Definition mulvgA a (v : FG) : FG := [ffun g => a * (v g)].
Definition mulrgA (v1 v2 : FG) : FG :=
 [ffun g => \sum_(k : gT) (v1 k) * (v2 ((k^-1) * g)%g) ].
Definition gAbasis : seq FG := map injG (enum G).
```

Après ces définitions et la preuve des propriétés d'algèbre associées, l'algèbre de groupe est définie comme une instance du type des algèbres.

```
Definition GAlgebra_algMixin := @AlgMixin _ _
 (GRing.Ring.mul GAlgebra_ringMixin) mulg_mulvl mulg_mulvr.
Definition GAlgebra := AlgType GAlgebra_algMixin.
```

Nous disposons ainsi d'une instance de `algType` qui représente l'algèbre de groupe. C'est une algèbre et donc un espace vectoriel et un anneau. Le type des éléments de cette algèbre est celui des fonctions à support fini de `G` dans `F`.

Sous-algèbre de sous-groupe

Chaque sous-groupe d'un groupe fini définit une sous-algèbre de l'algèbre du groupe. En effet, étant donné un sous-groupe H du groupe fini G, le sous ensemble $\{\sum_{g \in H} a_g g\}$ définit un sous-espace de l'algèbre du groupe G qui est aussi un F-espace vectoriel. C'est le sous-espace généré par les éléments du sous-groupe H vu comme des éléments de l'algèbre du groupe G. Grâce à la propriété de sous-groupe de H, ce sous-espace est stable par la multiplication interne de l'algèbre. C'est donc une sous-algèbre.

Le sous-espace de l'algèbre de groupe correspondant à un sous-groupe donné va être construit à partir de ses générateurs. Ces derniers sont donnés par l'injection des éléments du sous-groupe dans l'algèbre du groupe. La liste de ces générateurs est définie comme suit :

```
Variable (H : {group G}).
Definition spanFG := map injG (enum (mem H)).
```

Dans la définition ci-dessus, le terme `(enum (mem H))` correspond à une liste qui contient tous les éléments du sous-groupe `H`. La liste des générateurs est donnée par l'application de la fonction `injG` aux éléments de cette liste. Ainsi avec nos notations pour les sous-espaces, nous avons que `<< spanFG >>` représente le sous-espace généré par les éléments du sous-groupe `H`. Nous prouvons que ce sous-espace est une sous-algèbre avec le lemme suivant :

```
Lemma is_subalg_spanFG : is_algebra << spanFG >>.
```

Après cette preuve, la sous-algèbre associée au sous-groupe `H` est alors définie comme suit :

```
Definition groupSAlg := mk_algebra is_subalg_spanFG.
```

Avec cette définition et en partant de la correspondance entre représentations et modules, nous allons pouvoir définir les représentations d'un sous-groupe. En effet, une représentation d'un sous-groupe fini donné correspond à un sous-module sur la sous-algèbre associée. Dans la suite, nous noterons `F[H]` la sous-algèbre associé au sous-groupe `H` et le corps `F`.

4.4.2 Formalisation du théorème de Maschke

Nous avons donné dans la Section 4.2.3 page 78 une preuve du théorème de Maschke. C'est un résultat de base dans la théorie des représentations des groupes finis. Il permet de ramener l'étude de ces représentations à celles des représentations simples. Toujours dans cette section, le théorème de Maschke est prouvé pour des modules sur une algèbre de groupe. Cette preuve peut être généralisée pour les sous-modules sur une sous-algèbre de sous-groupe. Dans notre développement, la formalisation est faite de cette manière. Elle utilise la définition de sous-algèbre de sous groupe de la section précédente. La Figure 4.8 donne le code Coq de l'énoncé du théorème. Notre preuve Coq du théorème de Maschke procède de la même manière que la preuve donnée dans la Section 4.2.3. Elle est longue de 23 lignes, ce qui est le double de la longueur de la preuve papier standard. Ceci est un signe positif de la bonne intégration de notre développement générique sur les modules avec le développement sur les groupes finis. Cela indique aussi la facilité de réutilisation de ce développement.

La formalisation du théorème de Maschke a été une plateforme de test pour notre développement sur les modules. Elle jette aussi les bases pour la formalisation de résultats plus avancés sur les représentations et les caractères des groupes finis.

```
Variables (G : finGroupType) (H : {group G}) (F : fieldType).
Variable (mT : frmodType (GAlgebra F G)).
Notation Local "|H|" := (#|H| %:R : F).
Hypothesis (HcardG : |H| != 0).
Theorem Maschke : forall V : vspace mT, module F[H] V -> semisimple F[H] V.
Proof.
move=> V MV; split=> // W SW MW.
pose p' v := (1/|G|) *: \sum_(g \in G) ((proj W) (v :* (g %:FG))) :* ((g^-1)%g %:FG).
have Lp : linear p'; last set p := (lmap_of_fun Lp).
  apply: linear_mulv_fun; apply: linear_sum_fun => g Gg; apply: linear_mulm_fun.
  by apply: linear_comp_fun; [apply: lmap_linear | apply: linear_mulm].
suff Sp : rlinear_on p F[G] V.
  exists ((’ker p) :&: V)%V; split; first by apply/moduleP; apply: stableKer.
  rewrite subvsIr; split=> //; apply: proj_ds => //; split => [v | v Wv /=].
    rewrite lmapE in_vs_mul in_vs_sum 1?orbC // => ? ?.
    by rewrite (in_mod_mul MW) 1?proj_mem // FG_memG groupV.
  rewrite lmapE /p' (eq_bigr (fun g => v)) 1?sumr_const -1?mulv_nat => [|i *].
    by rewrite mulvA -mulrA mulVf 1?mulr1 1?mulv1.
  by rewrite proj_val 1?(in_mod_mul MW) 1?FG_memG // mulmA -injGM mulgV mulm1.
apply: rlinear_on_genA => /= a v Fa Vv; rewrite !lmapE mulm_mulvl -summ_distrl.
congr (_ *: _); case/mapP: Fa => /= h Gh ->.
rewrite (eq_bigr (fun g => (proj W ((v :* h%:FG) :* g%:FG) :*
 ((h * g)^-1%g)%:FG :* h%:FG))) => [| g Gg]; last first.
  by rewrite !mulmA -!injGM invMg -mulgA mulVg mulg1.
rewrite (reindex (fun g => (h^-1 * g)%g)) /=; last first.
 by exists (fun g => (h * g)%g) => g; rewrite mulgA 1?mulgV 1?mulVg mul1g.
rewrite (eq_bigl (fun g => g \in G)) => [| g /=]; last first.
  by rewrite groupMl 1?groupV -1?mem_enum.
by apply: eq_bigr => g Gg; rewrite !mulmA -injGM mulgA mulgV mul1g.
Qed.
```

FIGURE 4.8 – L’énoncé et la preuve Coq du théorème de Maschke.

4.5 Conclusions

Dans ce chapitre, nous avons présenté un développement formel de la théorie des modules et de celle des G-modules. Ce développement couvre les espaces vectoriels, les algèbres et les modules sur une algèbre. Il contient aussi une formalisation des sous-espaces vectoriels, des sous-algèbres, des sous-modules et des applications linéaires. En plus des structures d’algèbre linéaire le développement comprend une formalisation de la théorie des modules sur une algèbre de groupe fini. Ces développements formels sont combinés avec des travaux indépendants sur les groupes finis pour formaliser le théorème de Maschke. Au total, notre développement contient 3340 lignes de code Coq. Ces lignes de code correspond à 395 objets dont 110 définitions ou structures Coq et 285

lemmes Coq. La composante du développement sur la théorie générique des structures d'algèbre linéaire contient 3010 lignes de code pour 101 définitions et 253 lemmes. C'est la composante principale de ce développement.

4.5.1 Travaux futurs

Les travaux de formalisation présentés dans ce chapitre fournissent une bonne infrastructure pour faire de la théorie des représentations. Une première perspective à ces travaux est la formalisation de résultats plus avancés sur les représentations. Un exemple de résultat nécessaire pour la preuve du théorème de Feit-Thompson, est le théorème d'Artin-Wedderburn [28]. Ce théorème permet de décomposer une algèbre semi-simple en somme d'algèbres de matrices. La formalisation de ce théorème va nécessiter la définition d'une notion de morphismes entre modules. Ceci peut se faire au-dessus de notre formalisation des applications linéaires. En effet, les morphismes entre modules sur une algèbre sont des cas particuliers d'applications linéaires.

Une seconde perspective est de définir la somme directe des sous-espaces comme une opération binaire. Dans la Section 4.3.2, nous avons vu que dans notre développement la somme directe entre sous-espaces est définie par le prédicat booléen `dsum`. Ce prédicat prend en paramètre trois sous-espaces. Il retourne vrai si le premier argument est égal à la somme des deux autres et l'intersection de ces derniers est nulle. Avec cette définition, la manipulation de la somme directe de deux sous-espaces ne pose pas de problèmes majeurs. Par contre, elle devient moins pratique lorsque nous voulons faire la somme directe de plus que deux sous-espaces. En particulier nous ne pouvons pas utiliser avec cette définition la bibliothèque pour les opérations indexées.

La principale motivation du développement que nous avons présenté dans ce chapitre est la formalisation d'une théorie des représentations des groupes finis. Ceci rentre dans le cadre du projet *Mathematical Components* pour formaliser le théorème de Feit-Thompson. Par ailleurs, la théorie générique des structures d'algèbre linéaire que nous avons développée peut être réutilisée de façon indépendante. En particulier, elle peut servir comme base pour formaliser une théorie des espaces euclidiens. Elle pourra ainsi servir dans le cadre d'un autre projet majeur de formalisation mathématique : le projet *Flyspeck* qui vise à formaliser la preuve de la conjoncture de Kepler.

Enfin, une autre perspective intéressante est de connecter ce travail sur les représentations avec les travaux existants dans les systèmes de calculs algébriques. Ces systèmes fournissent des bibliothèques qui implémentent des algorithmes sur les représentations. En particulier, le système GAP [53] fournit un large ensemble large de bibliothèques

pour faire des calculs sur les représentations et les caractères des groupes finis. Dans le long terme, une utilisation de ces systèmes peut être utile. En effet, dans des parties avancées de la preuve du théorème de Feit-Thompson, nous aurons à faire des calculs sur les représentations et caractères de groupes particuliers.

4.5.2 Travaux reliés

Dans la communauté des utilisateurs des assistants à la preuve, il y a plusieurs développements sur l'algèbre linéaire qui couvrent une partie de notre formalisation. En revanche, à notre connaissance, il n'y a pas eu de travaux sur la théorie des représentations.

Certains des travaux [44] dans le système FoCaLize [24] (ancien FoCal) sont proches de notre développement sur la hiérarchie des structures algébriques. Ces travaux de formalisation s'intéressent surtout à la certification des systèmes de calcul algébrique. Par ailleurs, les limitations dans le système pour définir des opérations polymorphiques rendent la définition de structures algébriques abstraites et génériques compliquée.

La bibliothèque de mathématiques formelles de Mizar [36] contient des formalisations de structures algébriques comme les modules ou espaces vectoriels réels. Ces formalisations ont été faites par différents auteurs et dans différents articles disponibles dans le répertoire des preuves du système. La maintenance et l'extension de ces développements posent différents problèmes pratiques comme mentionné dans [47].

Dans l'assistant à la preuve Matita [1], une hiérarchie de structures algébriques qui comprend les espaces vectoriels et les algèbres a été développée [49]. Ce développement utilise les télescopes et le mécanisme de *coercion* de Matita. Il n'inclut pas de développement sur l'algèbre des sous-espaces ou des sous-algèbres. Par ailleurs, il est utilisé pour formaliser des résultats d'analyse constructive [48] dans le système.

Le système Isabelle comprend une formalisation d'une hiérarchie structures algébriques [3]. Cette hiérarchie couvre les groupes, les anneaux et va jusqu'aux modules. Elle est utilisée pour formaliser le théorème de Sylow et des résultats sur les polynômes. Dans HOL-Light, une formalisation des espaces vectoriels réels a été développée [26]. Elle est utilisée pour développer une théorie des espaces euclidiens.

Dans Coq, nous avons essentiellement deux développements sur les structures algébriques [9, 43]. Les structures algébriques définies dans ces développements utilisent les *setoids*. Elles utilisent les *télescopes* mais pas les structures canoniques. Il y a eu une tentative d'étendre la hiérarchie de [43] avec une algèbre des sous-espaces finis [52].

Cette tentative n'est pas allée très loin à cause des difficultés introduites par l'utilisation des *setoids*.

Conclusion

Dans cette thèse, nous avons proposé un ensemble de composants Coq pour la théorie des groupes finis. Ces composants font partie de l'infrastructure mathématique nécessaire à la formalisation du théorème de Feit-Thompson. Nous avons développé la première formalisation du théorème de Cayley-Hamilton dans un assistant à la preuve. Cette formalisation a été construite sur des développements indépendants sur les polynômes et les matrices, qui ont été combinés ensemble pour aboutir à la preuve formelle du théorème de Cayley-Hamilton. Nous avons aussi développé un ensemble de bibliothèques formelles pour la représentation des groupes finis. Ce développement contient une hiérarchie de structures d'algèbre linéaire qui couvre les espaces vectoriels, les algèbres et les modules sur une algèbre. Il contient aussi une formalisation de la théorie des sous-structures et des morphismes associés à ces structures algébriques. En partant du développement sur l'algèbre linéaire, nous avons formalisé le théorème de Maschke qui est un résultat important sur les représentations des groupes finis. Notre développement a été fait dans l'environnement Coq-SSReflect. Il a été construit sur l'ensemble des bibliothèques de base de SSReflect hérité du développement pour la formalisation du théorème des quatre couleurs. De nouvelles bibliothèques, comme celle sur les opérations indexées, ont été ajoutées à cet ensemble. Elles ont permis de répondre à des besoins qui se sont manifestés durant notre processus de développement.

L'objectif de nos travaux n'a pas seulement été la formalisation de résultats comme le théorème de Cayley-Hamilton ou le théorème de Maschke, nécessaires à la formalisation du théorème de Feit-Thompson, mais le développement de bibliothèques de mathématiques formelles. En faisant le parallèle avec le génie logiciel, la richesse des bibliothèques d'un langage de programmation est un facteur important dans la réussite des développements de grande taille. Le développement de bibliothèques mathématiques est une étape importante et nécessaire dans la formalisation de résultats mathématiques avancés comme le théorème de Feit-Thompson. Dans nos travaux de formalisation mathématique, notre approche a aussi été d'adopter des techniques de génie logiciel

comme la séparation entre contenu et présentation, et la définition d'interfaces de programmation. Les *coercions*, les structures canoniques et les types enregistrements de Coq nous ont permis d'appliquer ces techniques dans le contexte d'un environnement de développement de preuves formelles. Ces mécanismes et la richesse de la logique du système Coq (types et propositions dépendants) ont joué un rôle important dans notre développement. Ils ont permis d'organiser les objets et preuves définis dans ce développement. Ils ont aussi permis d'avoir un développement générique et pouvant être facilement étendu. Un exemple est la formalisation de la preuve du théorème de Cayley-Hamilton où les développements sur les polynômes et les structures algébriques ont été étendus et modifiés sans provoquer de changements majeurs dans la structure de la preuve.

La réflexion à petite échelle de SSReflect a joué un rôle important dans notre développement. D'un point de vue génie logiciel, elle peut être vue comme une application du principe de séparation entre contenu et présentation à l'ensemble des propositions décidables. Le choix dans notre développement, et plus généralement dans le projet *Mathematical Components*, est de travailler avec des objets et structures mathématiques décidables. Dans le cadre intuitionniste de Coq, ceci nous permet d'utiliser des propriétés classiques comme le tiers exclus, l'axiome du choix ou l'extentionnalité sans avoir besoin d'ajouter de nouveaux axiomes à la logique par défaut du système. La combinaison de la réflexion à petite échelle et de la programmation par interfaces permet de localiser l'utilisation de ces propriétés mathématiques classiques dans le développement. Les interfaces pour les types munis d'une égalité décidable et d'un opérateur de choix sont des exemples de cette combinaison.

Dans notre développement, les conditions de décidabilité sur les structures mathématiques peuvent être vues comme des limitations. Dans le cadre constructif de Coq, ceci est vrai d'un point de vue théorique puisque ces conditions ne sont pas toujours satisfaites pour toutes les structures mathématiques. Par contre, d'un point de vue pratique ces choix ne posent pas de problèmes étant donné que les structures mathématiques auxquelles nous nous intéressons respectent toujours ces choix. Ces conditions sont aussi respectées par l'ensemble non négligeable des structures mathématiques qui peuvent être représentées sur ordinateur. Plus généralement, ces conditions sont aussi toujours respectées dans la logique classique et dans les assistants à la preuve où cette logique est utilisée par défaut.

Notre expérience de développement mathématique dans Coq confirme qu'il a atteint un degré de maturité où il est envisageable de formaliser des théories mathématiques non triviales. Le système fournit des mécanismes comme les structures canoniques et les types dépendants qui permettent de développer des interfaces et bibliothèques génériques. Cependant et pour faciliter l'utilisation de ces bibliothèques, le système doit remédier à certaines limitations. En particulier, la gestion par Coq de la visibilité des définitions et preuves d'une bibliothèque n'est pas satisfaisante. Dans le travail avec les interfaces de programmation, il est important de disposer d'un mécanisme qui permette de dire au système que certains objets de la bibliothèques sont locaux, c'est à dire qu'ils ne doivent pas être visibles lors de l'importation de cette bibliothèque. Par exemple, un utilisateur qui voudra utiliser la bibliothèque sur les applications linéaires n'a pas besoin des définitions et preuves qui permettent de construire l'instance du type des espaces vectoriels associée à celui des applications linéaires. Il a seulement besoin de cette instance pour pouvoir utiliser les opérations et preuves génériques sur les espaces vectoriels. La gestion de la visibilité des définitions et preuves permet de s'assurer que l'utilisateur va utiliser les bonnes méthodes pour interagir avec la bibliothèque. C'est une partie importante de la programmation par interfaces. Elle permet aussi de limiter l'impact des changements dans les bibliothèques sur les développements utilisateurs.

Perspectives

Les travaux présentés dans cette thèse ouvrent la voie à de nouveaux développement mathématiques formels. Sur le court terme, les travaux sur le théorème de Cayley-Hamilton et en particulier les développements sur les polynômes fournissent une base pour entamer une formalisation de la théorie de Galois. Toujours sur le court terme, les développements sur la théorie des représentations fournissent une infrastructure pour formaliser des résultats plus avancées sur les représentations comme le théorème de Wedderburn sur la décomposition des algèbres simples. La formalisation de ce théorème ouvrira la voie à des développements sur la théorie des caractères. Ces développements permettront de formaliser les parties de la preuve papier du théorème de Feit-Thompson qui font appel à cette théorie.

Dans notre développement, les choix de définitions pour les objets et structures manipulés ne sont pas toujours adaptés au calcul. Le premier critère pris en compte dans ces définitions est la facilité d'utilisation pour faire des preuves. Il serait intéressant de combiner notre développements orienté preuve avec des développements orientés

calcul comme ceux existants dans les systèmes de calculs formels. Par exemple, notre développement sur la théorie des représentations peut être combiné avec des bibliothèques du système GAP pour le calcul sur les représentations. Au début de notre travail de thèse, nous avons effectué un développement pour tester la communication entre GAP et Coq avec le langage XML. Ce développement nous a permis d'échanger entre les deux systèmes des objets comme les entiers ou des groupes finis. L'échange d'objet plus complexe s'est heurté à l'absence dans GAP d'une interface standard pour communiquer avec des systèmes externes. De récents développements, comme ceux effectué dans le projet SCIEnce [45], permettent d'espérer que ces obstacles vont pouvoir être surmontés.

Sur le long terme, les développements sur la théorie des représentations fournissent une infrastructure de base qui permet d'envisager l'utilisation des assistants à la preuve pour la formalisation de travaux de recherche mathématiques contemporain. En particulier, ils peuvent être combinés avec d'autres développements du projet *Mathematical Components* pour commencer la formalisation du programme de Langlands [15].

Références

[1] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the matita proof assistant. *Journal of Automated Reasoning*, 39(2) :109–139, 2007. 110

[2] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Trans. Comput. Logic*, 9(1) :2, 2007. 2

[3] Clemens Ballarin, Florian Kammuller, and Lawrence C. Paulson. The isabelle/hol algebra library, 2009. Available at `http://isabelle.informatik.tu-muenchen.de/library/HOL/HOL-Algebra/document.pdf`. 110

[4] H.P. Barendregt. *The Lambda Calculus : Its Syntax and Semantics*. North-Holland, 1984. 7

[5] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2) :261–293, March 2003. 24

[6] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art : the Calculus of Inductive Constructions*. Springer-Verlag, 2004. 7

[7] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical big operators. In *Theorem Proving in Higher-Order Logics*, volume 5170 of *LNCS*, pages 86–101, 2008. 35, 42, 57

[8] N. G. De Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91 :189–204, 1991. 13

[9] C-CoRN, 2009. `http://c-corn.cs.ru.nl/`. 69, 110

[10] Jacek Chrzaszcz. Implementing modules in the coq system. In *Theorem Proving in Higher-Order Logics*, volume 2758 of *LNCS*, pages 270–286, 2003. 11

[11] Coq development team. *The Coq Proof Assistant Reference Manual, version 8.2*, 2008. 28, 52

[12] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Information and Computation*, 76(2/3) :95–120, 1988. 7

[13] Walter Feit and John G. Thompson. Solvability of groups of odd order. *Pacific Journal of Mathematics*, 13(3) :775–1029, 1963. 2, 74

[14] Francois Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *Theorem Proving in Higher-Order Logics*, volume 5674 of *LNCS*, pages 327–342, 2009. 58, 85, 90

[15] Stephen Gelbart. An Elementary Introduction to the Langlands Program. *Bulletin of the American Mathematical Society*, 10(2), 1984. 74, 116

[16] Herman Geuvers, Randy Pollack, Freek Wiedijk, and Jan Zwanenburg. A constructive algebraic hierarchy in Coq. *Journal of Symbolic Computation*, 34(4) :271–286, 2002. 2, 9, 83, 89

[17] Eduardo Giménez and Pierre Castéran. *A Tutorial on [Co-]Inductive Types in Coq*, 2007. 7

[18] Roger Godement. *Cours d'algèbre*. Hermann, 2005. 46

[19] Georges Gonthier. Formal proof - The Four-Color Theorem. *Notices of the American Mathematical Society*, 55(11), 2008. 2, 17

[20] Georges Gonthier and Assia Mahboubi. *A small scale reflection extension for the Coq system*, 2009. 17, 21

[21] Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry. A modular formalisation of finite group theory. In *Theorem Proving in Higher-Order Logics*, volume 4732 of *LNCS*, pages 86–101, 2007. 34, 57, 105

[22] Georges Gonthier and Stéphane Le Roux. *An Ssreflect Tutorial*, 2009. 17

[23] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in coq. In *Theorem Proving in Higher-Order Logics*, volume 3603 of *LNCS*, pages 98–113, 2005. 52

[24] Thérèse Hardin, Francois Pessaux, Pierre Weis, and Damien Doligez. *FoCaLize Manual, version 1.0*, 2009. 110

[25] John Harrison. HOL Light : A Tutorial Introduction. In *FMCAD*, pages 265–269, 1996. 69

[26] John Harrison. A hol theory of euclidean space. In *Theorem Proving in Higher-Order Logics*, volume 3603 of *LNCS*, pages 114–129, 2005. 110

[27] Gérard Huet and Amokrane Saïbi. Constructive category theory. In *Proof, language, and interaction : essays in honour of Robin Milner*, pages 239–275. MIT Press, 2000. 14

[28] I. Martin Isaacs. *Character Theory of Finite Groups*. American Mathematical Society, 1994. 74, 76, 109

[29] Nathan Jacobson. *Lectures in Abstract Algebra : volume II. Linear Algebra*. Springer-Verlag, 1975. 46

[30] Gordon James and Martin Liebeck. *Representations and Characters of Groups*. Cambridge University Press, 2001. 74, 76

[31] Israel Kleiner. *A history of abstract algebra*. Springer, 2007. 41, 42

[32] T. Y. Lam. Representations of finite groups : A hundred years, part i. *Notices of the American Mathematical Society*, 45(3), 1998. 73, 74

[33] Serge Lang. *Algebra*. Springer-Verlag, 2002. 46, 49

[34] Nicolas Magaud. Programming with dependent types in coq : a study of square matrices. `http://dpt-info.u-strasbg.fr/~magaud/UNSW/Coq/Matrices/`, 2005. 69

[35] Assia Mahboubi. *Contributions à la certification des calculs dans $\mathbb{R}$: théories, preuves, programmation*. PhD thesis, Université de Nice Sophia-Antipolis, 2006. 62

[36] MML, 2009. `http://www.mizar.org/`. 68, 110

[37] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. 69

[38] Sidi Ould Biha. Formalisation des mathématiques : une preuve du théorème de Cayley-Hamilton. In *Journées Francophones des Langages Applicatifs*, pages 1–14, 2008. 42

[39] Sidi Ould Biha. Finite groups representation theory with coq. In *Calculemus/MKM*, volume 5625 of *LNCS*, pages 438–452, 2009. 74

[40] Karol Pak and Andrzej Trybulec. Laplace expansion. *Journal of Formalized Mathematics*, 15(3) :143–150, 2007. 68

[41] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, Décembre 1996. 7

[42] Robert Pollack. Dependently Typed Records for Representing Mathematical Structure. In *Theorem Proving in Higher Order Logics*, pages 462–479. Springer-Verlag, 2000. 9

[43] Loïc Pottier. User contributions in Coq, Algebra, 1999. Available at `http://coq.inria.fr/contribs/Algebra.html`. 83, 89, 110

[44] Virgile Prevosto, Damien Doligez, and Thérèse Hardin. Algebraic structures and dependent records. In *Theorem Proving in Higher-Order Logics*, volume 2410 of *LNCS*, pages 298–313, 2002. 110

[45] projet SCIEnce. `http://www.symbolic-computation.org/The_SCIEnce_Project`. 116

[46] Piotr Rudnicki. Little Bezout theorem (factor theorem). *Journal of Formalized Mathematics*, 12(1) :49–58, 2004. 68

[47] Piotr Rudnicki, Christoph Schwarzweller, and Andrzej Trybulec. Commutative algebra in the Mizar system. *Journal of Symbolic Computation*, 32(1) :143–169, 2001. 110

[48] Claudio Sacerdoti Coen and Enrico Tassi. A constructive and formal proof of Lebesgue Dominated Convergence Theorem in the interactive theorem prover Matita. *Journal of Formalized Reasoning*, 1 :51–89, 2008. 110

[49] Claudio Sacerdoti Coen and Enrico Tassi. Working with Mathematical Structures in Type Theory. In *Proceedings of TYPES 2007 : Conference of the Types Project*, volume 4941 of *LNCS*, pages 157–172, 2008. 110

[50] Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *POPL '97 : Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 292–301. ACM, 1997. 12

[51] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Theorem Proving in Higher-Order Logics*, volume 5170 of *LNCS*, pages 278–293, 2008. 14

[52] Jasper Stein. Coq contribution : Linear algebra, 2001. http://coq.inria.fr/distrib/current/contribs/LinAlg.html. 110

[53] The GAP Group. *The GAP Reference Manual, version 4.4*, 2008. 109

[54] Peter Webb. *Finite Group Representations for the Pure Mathematician*. The manuscript of the book is available on the author web page http://www.math.umn.edu/~webb/RepBook/index.html, 2007. 74, 76

[55] Benjamin Werner. *Une théorie des Constructions Inductives*. PhD thesis, Paris 7, 1994. 7