UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H È S E

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice - Sophia Antipolis
Spécialité : Informatique

Présentée et soutenue par

Maxime Dénès

Étude formelle d'algorithmes efficaces en algèbre linéaire

Thèse dirigée par Yves Bertot

et préparée à l'INRIA Sophia Antipolis, équipe Marelle

soutenue le 20 novembre 2013

Jury :

Yves Bertot	Inria	Directeur
Thierry Coquand	Université de Göteborg	Rapporteur
Jean-Christophe Filliâtre	CNRS	Rapporteur
Herman Geuvers	Université de Nimègue	Examinateur
Georges Gonthier	Microsoft Research	Examinateur
Conor McBride	Université de Strathclyde	Examinateur

MAXIME DÉNÈS

ÉTUDE FORMELLE D'ALGORITHMES EFFICACES EN ALGÈBRE LINÉAIRE

THÈSE DE DOCTORAT

Équipe Marelle

http://www.maximedenes.fr/thesis

Septembre 2013

Maxime Dénès : *Étude formelle d'algorithmes efficaces en algèbre linéaire*, Thèse de doctorat, © Septembre 2013.

SITE WEB :
http://www.maximedenes.fr/thesis

E-MAIL :
mail@maximedenes.fr

La recherche menant à ces résultats a reçu un financement du 7ème projet cadre de l'Union Européenne sous la convention de subvention num. 243847 (ForMath).

À Melania et à mes parents.

ABSTRACT

Formal methods have reached a degree of maturity leading to the design of general-purpose proof systems, enabling both to verify the correctness of complex software systems and to formalize advanced mathematics. However, the ease of reasoning on programs is often emphasized more than their efficient execution. The antagonism between these two aspects is particularly significant for computer algebra algorithms, whose correctness usually relies on elaborate mathematical concepts, but whose practical efficiency is an important matter of concern.

This thesis develops approaches to the formal study and the efficient execution of programs in type theory, and more precisely in the proof assistant Coq. In a first part, we introduce a runtime environment enabling the native code compilation of such programs while retaining the generality and expressiveness of the formalism. Then, we focus on data representations and in particular on the formally verified and automatized link between proof-oriented and computation-oriented representations.

Then, we take advantage of these techniques to study linear algebra algorithms, like Strassen's matrix product, Gaussian elimination or matrix canonical forms, including the Smith normal form for matrices over a Euclidean ring. Finally, we open the field of applications to the formalization and certified computation of homology groups of simplicial complexes arising from digital images.

RÉSUMÉ

Les méthodes formelles ont atteint un degré de maturité conduisant à la conception de systèmes de preuves généralistes, permettant à la fois de vérifier la correction de systèmes logiciels complexes ou de formaliser des mathématiques avancées. Mais souvent, l'accent est mis davantage sur la facilité du raisonnement sur les programmes plutôt que sur leur exécution efficace. L'antagonisme entre ces deux aspects est particulièrement sensible pour les algorithmes de calcul formel, dont la correction repose habituellement sur des concepts mathématiques élaborés, mais dont l'efficacité pratique est une préoccupation importante.

Cette thèse développe des approches à l'étude formelle et l'exécution efficace de programmes en théorie des types, et plus précisément dans l'assistant à la preuve Coq. Dans un premier temps, nous présentons un environnement d'exécution permettant de compiler en code natif de tels programmes tout en conservant la généralité et l'expressivité du formalisme. Puis, nous nous intéressons aux représentations de données et plus particulièrement au lien formellement vérifié et automatisé entre représentations adaptées aux preuves ou au calcul.

Ensuite, nous mettons à profit ces techniques pour l'étude d'algorithmes en algèbre linéaire, comme le produit matriciel de Strassen, le procédé d'élimination de Gauss ou la mise en forme canonique de matrices, dont notamment la forme de Smith pour les matrices sur un anneau euclidien. Enfin, nous ouvrons le champ des applications à la formalisation et au calcul

certifié des groupes d'homologie de complexes simpliciaux issus d'images numériques.

REMERCIEMENTS

Le travail présenté dans cette thèse a été avant tout l'occasion et le fruit de rencontres extrêmement enrichissantes. Je souhaite remercier très chaleureusement les personnes exceptionnelles que j'ai eu la chance de croiser ou d'avoir dans mon entourage et pour qui j'ai une grande estime et une admiration sincère.

Merci donc, Yves, pour ton encadrement sans faille. Et ce, depuis ce jour de début de stage à l'été 2008 où ma réponse négative à la question fatidique « Tu as déjà fait du Coq ? » ne t'a pas découragé et m'a valu une explication particulièrement pédagogique de l'isomorphisme de Curry-Howard ! Puis, plus tard, pour la liberté de travail que tu m'as accordée durant ces trois années, tout en répondant toujours présent aux moments où j'en avais besoin, avec à chaque fois des conseils avisés.

Merci à Thierry Coquand et Jean-Christophe Filliâtre pour l'intérêt porté à mon travail en acceptant d'être rapporteurs de cette thèse, et dont les nombreux retours ont permis d'améliorer sensiblement ce document.

Merci à Georges Gonthier and thank you, Herman Geuvers and Conor McBride, I am deeply honored that you have accepted to be in my jury.

Dire que j'ai bénéficié pendant ces trois années d'un excellent environnement de travail est un euphémisme. L'ambiance familiale, les discussions passionnantes (et passionnées !) et la bonne humeur générale de l'équipe Marelle sont des souvenirs très forts. Merci donc à ses membres, Laurence (avec qui tous les problèmes semblent résolubles, comme les groupes d'ordre impair), Laurent (qui m'a guidé dans ma première preuve Coq !), Benjamin (dont les compétences vont de la théorie des types au dépannage de voiture), Loïc, José, Nathalie, Ioana, Sidi, Sylvain, Guillaume, Erik, Nicolas, Julianna, Michaël.

Merci aux membres de l'équipe Mathematical Components que j'ai eu la chance de rencontrer très tôt dans mon apprentissage de Coq. Quoi de plus motivant que de voir les belles mathématiques que l'on peut formaliser ? Merci donc, Georges, Assia, Enrico, Cyril (et ceux déjà cités) pour votre gentillesse et votre accessibilité, et pour m'avoir fait tomber dans la marmite (SSReflect) étant petit.

Cette thèse a eu lieu au sein du projet européen ForMath regroupant, outre des membres de Marelle et de Mathematical Components, des équipes des universités de Göteborg, Nimègue et La Rioja. Ces horizons différents ont donné lieu à des collaborations et échanges incroyablement enrichissants. Merci à Thierry, Vincent, Arnaud, tack Anders, bedankt Bas, Herman en Robbert, gracias Julio, María y Jónathan.

Merci aux membres de l'équipe Typical où j'ai amorcé le travail qui a abouti au premier chapitre de cette thèse. Merci à Gilles, pour ton soutien et la grande qualité de ton encadrement pendant mon stage de M2. Merci à Mathieu, un troll vaut souvent mieux qu'un long discours, travailler à tes côtés a été un grand plaisir. Merci à Bruno pour ta patience infinie face à mes innombrables questions pendant mes premières tentatives en apnée dans le code source de Coq. Merci également à Benjamin, Jean-Marc et Germain.

Merci aussi aux membres de l'équipe πr^2, qui m'ont toujours chaleureusement accueilli, notamment à l'occasion des GT Coq qui sont toujours très enrichissants. Merci Hugo, Pierre, Matthieu, Yann, Pierre, Pierre-Marie.

Merci à toute l'équipe Plume de l'ENS Lyon pour leur accueil pour un exposé en octobre 2011. Les (très) nombreuses questions et retours que j'ai eu ce jour-là ont eu une grande influence sur la suite du travail présenté dans cette thèse. Merci également à Damien et Thomas qui avaient arrangé cette rencontre et étaient présents.

Merci à tous les gens qui auraient dû être cités sur cette page, mais que j'ai honteusement oubliés. J'en suis sincèrement désolé.

Merci aux amis qui m'ont entouré pendant ces années, me forçant à m'entraîner à répondre à la question terrorisant tout doctorant : « Au fait, tu travailles sur quoi exactement ? ». Merci aussi aux amis que j'ai perdus de vue, mais qui occupent toujours une place.

Merci à ma famille, sans laquelle je n'en serais pas là, et notamment mes parents. Je n'oublie pas ce que je vous dois.

Enfin, je comprends aujourd'hui pourquoi la plupart des doctorants remercient en particulier une personne qui a traversé avec eux l'épreuve que constitue une thèse. Melania, îți mulțumesc pentru sprijinul și dragostea ta.

TABLE DES MATIÈRES

0 INTRODUCTION

0.1 MOTIVATION

Notre société a un rapport complexe à l'outil informatique. Lequel d'entre nous n'a jamais fait l'expérience de systèmes ou logiciels capricieux, peu fiables et d'une complexité difficilement saisissable ? Et pourtant, nous leur confions notre vie régulièrement et sans état d'âme, que ce soit dans les transports (lignes de métro automatisées, aiguillage des trains, circuit de commandes d'avions), en médecine (aide au diagnostic, intervention chirurgicale assistée) ou pour la supervision d'infrastructures critiques comme les centrales nucléaires.

Fort heureusement, cette décontraction est en partie permise par les techniques spécifiques utilisées lors de la conception et la mise au point des systèmes destinés à des domaines critiques, généralement nommées *méthodes formelles*. Elles consistent à définir et appliquer des outils scientifiques et mathématiques pour étudier le comportement et faciliter la conception de systèmes logiciels, sur le modèle de l'ingénierie traditionnelle. Un des principaux objectifs est de pouvoir définir rigoureusement ce qui est attendu d'un programme (sa *spécification*) et justifier mathématiquement qu'une réalisation logicielle satisfait cette attente.

Le plus souvent, la spécification d'un logiciel destiné à un usage grand public est informelle et incomplète. L'adéquation du produit final à ce cahier des charges est vérifiée de manière empirique par des cycles alternant tests et correctifs, jusqu'à ce qu'aucun comportement trop gênant ne soit plus observé. On peut penser que la conception d'un pont suivant les mêmes principes remporterait une adhésion limitée. Au contraire, on exigerait qu'il ait été modélisé, que des plans aient été établis et suivis, que les techniques utilisées aient été éprouvées.

Ainsi, même si les développements récents du génie logiciel conduisent à organiser rigoureusement les phases de conception, les cycles, l'interaction avec l'utilisateur lors de la mise au point, il n'en reste pas moins que les techniques utilisées ne sont que très peu mathématisées. En effet, les méthodes formelles ont actuellement un coût trop élevé pour espérer dépasser la sphère des domaines critiques ou stratégiques, alors même que la sûreté et la rigueur qu'elles apportent devraient les amener à se généraliser.

D'autant que l'empreinte croissante des équipements informatiques dans notre vie quotidienne augmente leur pouvoir de nuisance en cas de dysfonctionnement. Citons, par exemple, les erreurs de programmation des applications réveil et calendrier sur certains iPhone 3G, 3GS et 4, et qui ont valu à des milliers d'utilisateurs, le lendemain d'un changement d'heure, puis d'année, d'arriver en retard au travail, de rater un avion, avec parfois des conséquences très douloureuses [N. Bilton, 2011]. Il y a peu, un tel problème aurait touché tout au plus les quelques détenteurs d'un même modèle de réveil. Mais l'industrialisation et la généralisation de ces systèmes fait qu'aujourd'hui une part importante de la population active d'un pays peut être affectée simultanément, avec un impact économique significatif le jour où cela se produit.

Mais pourquoi le coût de la rigueur serait plus élevé ici que dans l'ingénierie traditionnelle ? Une réponse possible est que les logiciels actuels sont parmi les constructions les plus complexes que l'esprit humain ait produites et surtout industrialisées, à ce jour. Mais une autre cause de cet usage limité est que les sciences permettant de mathématiser le génie logiciel ont été moins développées, par exemple, que celles applicables au génie civil (mécanique du solide et des fluides, thermodynamique,...) qui ont hérité de plusieurs siècles de recherche et d'intérêt.

Cette relative jeunesse est compensée par le dynamisme de ce champ de recherche. Des connexions importantes ont été établies avec la sémantique des langages de programmation, car il s'agit de modéliser mathématiquement le comportement des programmes, et avec les fondements des mathématiques car il faut également définir le cadre logique du raisonnement déductif. Plus généralement, la preuve de correction de programmes a été rapprochée de la vérification de preuves de théorèmes mathématiques généraux.

En pratique, de nombreuses applications à grande échelle des méthodes formelles voient le jour. Parmi les plus spectaculaires, citons le dépôt de mathématiques constructives associé à la preuve du théorème fondamental de l'algèbre [H. Geuvers et al., 2000], celle du théorème des quatre couleurs [G. Gonthier, 2007], la vérification du micro-noyau seL4 [G. Klein et al., 2009], la preuve du théorème de Feit-Thompson [G. Gonthier, A. Asperti et al., 2013], celle de la conjecture de Keppler [T. C. Hales, 2005], ou encore le développement du compilateur formellement vérifié CompCert [X. Leroy, 2009].

Outre leur intérêt propre, de tels travaux permettent d'évaluer le passage à l'échelle des techniques existantes, et d'en mettre au point de nouvelles. Ce qui est remarquable, c'est que des outils généraux permettent de traiter à la fois la vérification de programmes complexes et la formalisation de théorèmes reposant sur des théories mathématiques avancées.

La thèse présentée ici se situe à la frontière entre ces deux aspects, car elle explore les possibilités d'appliquer les méthodes formelles à des programmes manipulant des objets mathématiques. Nous poursuivons un objectif double : de tels programmes, comme les logiciels de calcul formel ou numérique, sont déjà très utilisés, notamment pour l'ingénierie et la conception. Les étudier formellement permet d'augmenter la confiance qui peut leur être accordée. Mais réciproquement, avoir formellement vérifié un tel programme permet de l'utiliser par la suite comme outil de preuve sans étendre la base de confiance.

Notre étude cible plus précisément les algorithmes utilisés en algèbre linéaire, dont l'omniprésence en mathématiques les rend représentatifs. Mais les outils et approches que nous développons sont applicables, et parfois appliqués, à d'autres théories mathématiques.

0.2 PRÉREQUIS ET OUTILS UTILISÉS

0.2.1 Programmation fonctionnelle et théorie des types

L'évolution globale des langages de programmation implémentés sur ordinateurs tend à remplacer les langages machine ou d'assemblage, spécifiques à une architecture cible, par des langages portables et de haut niveau, c'est-à-dire favorisant la conceptualisation des objets manipulés.

Plus précisément, des compilateurs font le pont entre ces deux niveaux d'abstraction, permettant au programmeur d'avoir une vue idéalisée de l'algorithme qu'il implémente, tout en générant in fine un code machine ou assembleur efficace. Ces langages de bas niveau font donc toujours partie de la chaîne de compilation, mais sont rarement manipulés directement par le programmeur.

Cette augmentation du niveau d'abstraction pose la question de la modélisation des concepts du langage. Ils peuvent y être décrits de manière très opérationnelle dans des langages relativement bas niveau mais portables, comme C, ou encore vus comme des objets dans les langages dits « orientés objets ».

La programmation fonctionnelle est un paradigme où les programmes sont vus comme des fonctions au sens mathématique. En particulier, la notion de fonction devient centrale et manipulable dans le langage comme n'importe quelle valeur (passage en argument, retour comme résultat, application partielle, curryfication,...). Dans son interprétation dite « pure », cette approche signifie aussi que deux évaluations d'un programme appliqué aux mêmes entrées doivent renvoyer le même résultat en sortie. Sont ainsi exclus tous les *effets de bords* présents dans les langages impératifs habituels.

Les exemples de langages fonctionnels, purs ou non, sont nombreux. LISP [J. McCarthy, 1960] est probablement précurseur dans ce domaine, mais citons également Haskell [P. Hudak et al., 2007], Erlang [J. Armstrong, 2010] et la famille ML : Standard ML [R. Milner et al., 1990], OCaml [X. Leroy et al., 2012]. Nous utiliserons en particulier ce dernier langage.

Les algorithmes que nous étudierons seront matérialisés par des programmes purement fonctionnels. Cette approche impose des contraintes, notamment en termes de performances, mais facilite l'étude formelle. En effet, un des intérêts des langages fonctionnels est qu'ils ont une sémantique relativement facile à exprimer formellement, leur modèle théorique étant le λ-calcul de Church.

À ce sujet, il est intéressant de noter que le λ-calcul a été originellement introduit comme système formel en réponse aux problèmes de fondements des mathématiques. Malheureusement, la logique initialement décrite par ce système s'est avérée incohérente. C'est comme modèle de calculabilité que le λ-calcul va décrocher ses premiers succès, dont la solution négative à l'important problème de la décision posé par Hilbert (« Entscheidungsproblem »). Il est une alternative, de même expressivité, aux machines de Turing.

Ce n'est qu'en ajoutant une notion de *types* au formalisme que Church parviendra à décrire un système logique consistant basé sur le λ-calcul. Les types catégorisent les termes et restreignent les opérations permises, évitant ainsi les paradoxes logiques, au prix d'une perte d'expressivité.

Cette notion trouve un écho dans certains langages de programmation qui catégorisent les valeurs selon leur type, permettant ainsi de s'assurer que différentes parties d'un programme respectent la même abstraction sur un objet en mémoire. En effet, au niveau matériel rien ne distingue une suite de bits codant un entier d'une autre représentant une chaîne de caractères. Des informations de typage permettent donc notamment d'éviter des erreurs de programmation.

L'isomorphisme de Curry-De Bruijn-Howard concrétise cette analogie, en identifiant les preuves d'une proposition à des programmes d'un certain type. Certaines théories des types, au rang desquelles la théorie intui-

tionniste des types [P. Martin-Lof et G. Sambin, 1984] ou le Calcul des Constructions [T. Coquand et G. P. Huet, 1988], se basent sur cette identification pour donner aux preuves un statut de première classe. Elles sont donc tout autant un langage de preuve qu'un langage de programmation. Ce double point de vue est utilisé de manière centrale dans notre travail.

Le formalisme dans lequel seront exprimés les preuves et les programmes que nous écrirons est basé sur le Calcul des Constructions (Co)Inductives, qui hérite du Calcul des Constructions le polymorphisme et les types dépendants, et l'étend avec une hiérarchie d'univers, des types de données inductifs et coinductifs. D'autres fonctionnalités sont présentes dans le formalisme que nous utilisons réellement, comme un système de modules, mais par souci de simplicité nous l'appellerons toujours dans la suite « Calcul des Constructions Inductives » (C.C.I.).

Notons que le cadre logique associé à ce formalisme est constructif, ce qui signifie que la proposition $\forall P, P \vee \neg P$ n'est pas prouvable. La notion de preuve d'existence est par contre renforcée : si la proposition $\exists x, P(x)$ est prouvable, où $P(x)$ a pour seule variable libre x, alors un algorithme permet de trouver un t tel que $P(t)$ est prouvable. Nous n'utiliserons dans cette thèse aucun axiome logique supplémentaire, et nous limiterons donc à des preuves constructives. En règle générale, le développement de l'algèbre constructive est une discipline à part entière [H. Lombardi et C. Quitté, 2011]. Cependant, la plupart des objets que nous traitons sont finis, ce qui simplifiera considérablement notre étude. Lorsque nous adopterons des définitions spécifiques pour rester dans un cadre constructif, nous le signalerons.

0.2.2 Assistants de preuve

Nous avons déjà mentionné l'unité conceptuelle entre la vérification de programmes et la formalisation de preuve en mathématiques. Il existe cependant des différences de forme. L'étape la plus complexe pour prouver la correction d'un programme est généralement de deviner les invariants, c'est-à-dire les propriétés logiques suffisamment faibles pour rester vraies au cours de l'exécution du programme, mais suffisamment fortes pour impliquer sa correction. Une fois ces invariants définis, il reste à montrer qu'ils sont bien préservés, le plus souvent grâce à l'application d'arguments relativement simples, répétée un grand nombre de fois pour couvrir l'ensemble des constructions du programme. Le caractère fastidieux et répétitif de cette vérification conduit à développer des outils pour l'automatiser, on parle alors de *démonstration automatique*.

A contrario, la formalisation de preuves mathématiques réclame souvent un travail approfondi sur les définitions, c'est-à-dire l'encodage choisi dans le langage formel pour représenter un concept mathématique. La structure du raisonnement déductif proprement dit est relativement linéaire, ce qui permet d'envisager un développement manuel de la preuve. Seule la *vérification* de la correction du raisonnement logique est alors automatisée. La recherche automatique de preuve dans ce contexte ne peut être que partielle, soit à cause de résultats d'indécidabilité pour des logiques expressives, soit par une trop grande complexité algorithmique. Dès lors, raisonner manuellement, même sur certaines étapes élémentaires, présente l'avantage de mettre à l'épreuve l'encodage choisi pour les concepts, en vue des étapes plus complexes où la démonstration automatique n'aboutit plus.

Dans la suite, nous nous placerons dans un contexte hybride, où la vérification des preuves est automatique, et où certaines étapes de raisonnement peuvent être automatisées, à la discrétion de l'utilisateur. On parle alors de preuve semi-interactive, et les outils implémentant cette approche sont appelés *assistants à la preuve*.

Nous utilisons en particulier l'assistant à la preuve Coq [H. Herbelin et al., 2013], qui implémente le Calcul des Constructions Inductives déjà évoqué. Outre les considérations pratiques comme la réutilisation de développements déjà existants, deux raisons motivent ce choix. D'une part, l'expressivité du formalisme, et notamment la présence de types dépendants, nous permet d'encoder confortablement les notions mathématiques qui nous intéressent. D'autre part, les performances de calcul nous importent, or Coq implémente des technologies évoluées dans ce domaine, que nous tenterons d'ailleurs d'améliorer encore.

0.2.3 Calcul formel

Le terme de calcul formel désigne habituellement des techniques de manipulation d'expressions mathématiques. Les logiciels de calcul formels sont couramment utilisés pour résoudre symboliquement des problèmes comme la différentiation, la factorisation ou certains types d'équations. À première vue donc, les points communs sont nombreux avec le monde des assistants à la preuve.

Pourtant, les différences sont plus larges qu'il n'y paraît. En effet, les logiciels de calcul formel les plus courants sont orientés vers la simplicité d'utilisation, et l'obtention d'un résultat, même s'il n'est que partiellement vérifié. Typiquement, l'utilisateur saisit une expression, lance le calcul, et reçoit une nouvelle expression. La sémantique de ce processus, c'est-à-dire le lien entre l'expression originale et celle récupérée en sortie, n'est pas explicite, et parfois difficile à exprimer précisément.

Nous avons déjà évoqué le double intérêt que présente le rapprochement des logiciels de calcul formel et des assistants à la preuve : d'une part, le lien renforcé entre preuve de correction et implémentation (toutes deux réalisées sur machine dans un formalisme unique, par opposition à une preuve sur papier d'une implémentation machine) offre plus de garanties. D'autre part, l'intégration d'outils de calcul formel dans un assistant à la preuve permet d'aider l'utilisateur à mener à bien une preuve en automatisant certaines étapes calculatoires.

Plusieurs méthodes ont déjà été développées pour un tel rapprochement. On les regroupe habituellement en trois familles [H. Barendregt et E. Barendsen, 2002] :

- L'approche *crédule*, qui consiste à faire confiance aux résultats fourni par un logiciel de calcul formel. La complexité de ces logiciels et leur taille rend cette approche peu satisfaisante scientifiquement.

- L'approche *sceptique* qui consiste à instrumenter un logiciel de calcul formel pour qu'il fournisse une trace ou un certificat permettant de vérifier la validité du résultat à moindre frais [J. Harrison et L. Théry, 1998 ; C. Kaliszyk et F. Wiedijk, 2007]. Le vérifieur est prouvé formellement, ce qui permet de ne pas étendre la base de confiance.

- L'approche *autarcique* qui consiste à prouver directement la correction d'une routine de calcul formel. Bien sûr, cette preuve demande un tra-

vail plus conséquent que dans les deux autres approches mais élimine le besoin d'une trace ou d'un certificat, qui peuvent avoir un coût.

Puisque nous nous intéressons directement à la vérification de la correction d'algorithmes de calcul formel, nous nous placerons le plus souvent dans l'approche autarcique.

0.2.4 Homologie

En mathématiques, la topologie algébrique est un domaine qui étudie des espaces topologiques en leur associant une structure algébrique, de telle manière qu'à deux espaces homéomorphes (c'est-à-dire admettant une bijection continue d'inverse continue) soient associées des structures isomorphes.

La relation d'homéomorphisme est fine mais peu pratique pour des applications calculatoires. L'homologie est une théorie qui relâche un peu cette relation, en étudiant les objets selon la structure de leur trous dans une dimension donnée. Plus précisément, pour un espace topologique X et un entier naturel k, on construit un groupe abélien $H_k(X)$ qui décrit la structure des trous de X de dimension k ($H_0(X)$ décrit les composantes connexes de X).

L'homologie peut être étudiée à des niveaux d'abstractions très variés, que ce soit à partir d'une réalisation géométrique « concrète » ou en faisant appel à la théorie des catégories. Dans cette thèse, nous nous intéressons à l'homologie dite *simpliciale*, qui étudie des structures abstraites d'un point de vue purement combinatoire. Très peu de prérequis sont donc nécessaires et les définitions essentielles sont données au chapitre 6, dont celle de la notion centrale de *complexe simplicial*.

La motivation pour l'étude formelle de l'homologie nous est apportée par le projet européen ForMath [T. Coquand et al., 2010] dans le cadre duquel a été développée cette thèse. En effet, ce projet vise à formaliser des mathématiques utiles à la vérification de logiciels interagissant avec le monde physique. L'algèbre linéaire et la topologie algébrique font naturellement partie de ces quelques domaines d'étude.

0.3 État de l'art de l'algèbre linéaire formalisée

L'étude formelle d'algorithmes d'algèbre commutative a déjà été entreprise, par exemple pour l'algorithme de Buchberger pour le calcul de bases de Gröbner [T. Coquand et H. Persson, 1998 ; L. Théry, 2001]. De nombreuses bibliothèques d'algèbre linéaire existent, que ce soit dans le système HOL Light [J. Harrison, 2005], Isabelle [S. Obua, 2005], ACL2 [R. Gamboa et al., 2003 ; J. Hendrix, 2003] ou Coq [L. Pottier, 1999 ; J. Stein, 2001 ; N. Magaud, 2005 ; G. Gonthier, 2011]. Cependant, à notre connaissance, aucune ne propose à la fois une description de haut niveau des concepts en algèbre linéaires (notamment d'espaces vectoriels) et une possibilité d'exécution efficace des algorithmes sous-jacents (multiplication matricielle, calcul du rang,...).

Concilier ces deux aspects est le fil conducteur du travail présenté dans cette thèse. Afin de réutiliser au maximum l'existant, nous nous basons sur la bibliothèque SSReflect [G. Gonthier, A. Mahboubi et al., 2008]. Celle-ci intègre un développement très complet d'algèbre linéaire formalisée [G.

Gonthier, 2011], traitant les concepts de vecteur, de matrice ou d'espace vectoriel d'une manière uniforme en les construisant comme une abstraction au-dessus d'un encodage matriciel. Ce point de vue permet d'obtenir une description de haut niveau, tout en ayant accès à un encodage plus concret quand c'est nécessaire. Cependant, aucun calcul pratique n'est possible sur ces objets concrets du fait de problèmes d'efficacité, qui sont liés à des choix de conception orientés vers la facilité des preuves et non l'effectivité des calculs.

Ces questions d'effectivité des calculs, ainsi que les notations et définitions fournies par SSReflect que nous utilisons seront expliquées au fur et à mesure de leur introduction dans nos développements. L'idée générale à retenir est que nous utilisons SSReflect pour *spécifier* et *prouver* la correction de nos algorithmes, et que nous développerons une couche permettant le calcul effectif de ces algorithmes, que nous pensons être un complément utile à SSReflect.

0.4 ORGANISATION DE LA THÈSE

Cette thèse est organisée en trois parties : la première présente des techniques que nous avons mises au point pour permettre l'exécution efficace de programmes au sein d'un assistant à la preuve (Coq, dans notre cas) tout en conciliant ce souci d'efficacité avec la faisabilité de la preuve de correction de ces mêmes programmes.

Ensuite, dans la partie II nous appliquons ces techniques à l'étude de quelques algorithmes fondamentaux en algèbre linéaire. Nous décrivons à la fois les algorithmes eux-mêmes, la formalisation de leur preuve de correction et les performances de leur implémentation certifiée.

Enfin, la partie III réutilise les outils mis en place dans les parties précédentes à un sujet d'ouverture, le calcul de groupes d'homologie. Plus spécifiquement, nous nous intéressons aux groupes d'homologie de complexes simpliciaux issus d'images numériques.

0.5 FICHIERS SOURCES

L'ensemble des formalisations et développements décrits dans cette thèse est disponible en ligne. Notamment, une version de Coq incluant toutes les fonctionnalités décrites au chapitre 1 est disponible à l'adresse :

`https://github.com/maximedenes/native-coq`.

Ces fonctionnalités ont été portées à la version de développement de Coq, à l'exception des entiers machines et des tableaux persistants. L'ensemble du développement (compilateur, entiers, tableaux) représente 4600 lignes de code OCaml.

Toutes les formalisations du chapitre 2 et de la partie II sont intégrées à notre bibliothèque CoqEAL, qui est disponible à l'adresse :

`http://www.maximedenes.fr/coqeal`

Au moment de l'écriture de ces lignes, notre bibliothèque contient environ 690 définitions et 830 lemmes pour un total de 12630 lignes de code. Parmi

cette base de code issue d'une collaboration au sein du projet ForMath, environ 6670 lignes correspondent directement au matériel décrit dans le chapitre 2 et la partie II de cette thèse.

Enfin, la page `http://www.maximedenes.fr/thesis` récapitule le matériel accompagnant cette thèse, les versions de Coq et SSReflect requises pour le compiler, et fournit les développements décrits dans la partie III de cette thèse. Ceux-ci représentent environ 110 définitions et 100 preuves pour 2560 lignes de code Coq. Nos évaluations de performances sur les images numériques sont implémentées par 190 lignes supplémentaires.

Seuls quelques jeux de tests requièrent notre propre version de Coq. Par souci de reproductibilité, précisons que toutes les études de performances décrites dans cette thèse ont été effectuées sur une machine disposant d'un processeur Intel® Xeon® X5482 cadencé à 3.2 GHz, dont nous n'utilisons qu'un seul coeur, et de 32 Go de mémoire vive.

Première partie

PROGRAMMES EFFICACES EN THÉORIE DES TYPES

1 COMPILATION NATIVE DE LA RÉDUCTION FORTE

La plupart des assistants de preuve modernes reposent sur un formalisme incorporant un langage de programmation, souvent basé sur le λ-calcul. S'inspirant de la logique d'ordre supérieur [A. Church, 1940], ils utilisent ce langage à la fois pour décrire les *objets* du raisonnement et les *propositions*.

Les systèmes à types dépendants, dans la lignée du précurseur Automath [N. G. De Bruijn, 1970], poussent cette approche encore plus loin, en unifiant les objets du raisonnement, les propositions et les preuves. C'est l'isomorphisme de Curry-De Bruijn-Howard, que nous notons ici ϕ. Une preuve p d'une proposition P est vue comme un programme $\phi(p)$ de type $\phi(P)$. Les identifier, c'est omettre ϕ et noter $p : P$ dans les deux cas. Sur le plan théorique, cet isomorphisme permet de réutiliser des techniques de sémantique des langages de programmation pour établir des résultats en théorie de la preuve. Mais ses conséquences sont également pratiques, car il permet d'uniformiser l'implémentation du formalisme et de réduire la vérification d'une preuve à celle du bon typage d'un programme.

Programmer dans une telle théorie des types peut signifier deux choses. La première possibilité est de construire une preuve d'un énoncé de la forme $\forall x\, \exists y\, P(x, y)$ et d'en extraire [C. Paulin-Mohring, 1989a,b ; P. Letouzey, 2008] le contenu calculatoire sous forme d'un programme fonctionnel prenant un x en entrée et renvoyant un y tel que $P(x, y)$. C'est la méthodologie suivie, par exemple, dans le développement du compilateur certifié CompCert, auquel nous avons déjà fait référence. L'avantage de cette approche est qu'elle permet de ne compiler et de n'évaluer que le contenu calculatoire. En particulier, dans notre exemple, la preuve de $P(x, y)$ n'apparaît pas dans le programme extrait. Le revers de la médaille est que le résultat de l'évaluation est externe, et ne peut donc être utilisé par la suite dans d'autres preuves.

Or les programmes que nous certifions dans cette thèse manipulent des objets mathématiques, et le résultat de leur exécution est souvent intéressant à réutiliser dans des preuves. Heureusement, une seconde possibilité est ouverte en théorie des types par la règle de conversion :

$$\frac{\Gamma \vdash M : A \qquad A \equiv B}{\Gamma \vdash M : B}\ (\text{conv})$$

qui identifie deux types ou propositions A et B s'ils sont reliés par la relation $\equiv$. Le choix de $\equiv$ divise les théories des types en deux grandes familles. Dans une théorie des types intentionnelle comme celle implémentée par Coq, $\equiv$ représente le calcul. C'est-à-dire que deux types A et B seront identifiés (*convertibles*) si, vus comme des programmes, ils s'évaluent tous deux de la même manière. Dans le fragment restreint au λ-calcul, $\equiv$ est la congruence induite par la β-réduction. Pour étendre au formalisme complet, sont rajoutées des règles de réduction correspondant aux différentes constructions du langage.

Mais il est également possible de renforcer la règle de conversion. En théorie des types extensionnelle, $A \equiv B$ exprime que les types A et B sont

prouvablement égaux. L'avantage est alors que le système de types est plus souple (les conditions pour pouvoir remplacer un type par un autre sont plus faibles), mais la relation $\equiv$, et par conséquent la vérification de types, deviennent indécidables. Pour vérifier qu'un terme `t` a le type `T`, il est alors nécessaire de stocker une dérivation de typage.

Des approches intermédiaires existent également, se basant sur des systèmes de types dont la relation de convertibilité est enrichie par des règles de réécriture [G. Dowek et al., 2003], ou des procédures de décision pour des théories du premier ordre [B. Barras, J.-P. Jouannaud et al., 2011], de manière à gagner en souplesse tout en gardant une vérification de types décidable. Dans la suite de ce chapitre et de cette thèse, nous nous placerons dans une théorie des types intentionnelle habituelle, telle que celle de Coq.

La règle de conversion ouvre des possibilités intéressantes du point de vue de l'automatisation des preuves. Ainsi, la notion de « preuve par réflexion » est une méthodologie qui a émergé au fil des années [D. J. Howe, 1988 ; G. Barthe, M. Ruys et al., 1995 ; J. Harrison, 1995 ; S. Boutin, 1997 ; K. N. Verma et al., 2000 ; M. Oostdijk et H. Geuvers, 2002 ; B. Grégoire et A. Mahboubi, 2005 ; G. Gonthier, 2007] et qui exploite le principe de remplacement d'étapes de preuve par des étapes de calcul. Prenons par exemple la relation d'ordre $\leq$ sur les entiers naturels qui est définie dans la librairie standard de Coq par le prédicat inductif suivant :

```
Inductive le (n:nat) : nat -> Prop :=
  | le_n : n <= n
  | le_S : forall m:nat, n <= m -> n <= S m
```

Avec cette définition, une preuve d'une instance close de `n <= m` a une taille qui dépend des valeurs de `n` et `m`. Par exemple, une preuve de `0 <= 1` s'écrit `le_S 0 0 (le_n 0)` tandis qu'une preuve de `0 <= 2` ressemble au terme `le_S 0 1 (le_S 0 0 (le_n 0))`. Mais on peut remarquer qu'en définissant une fonction booléenne `leq` comme dans la librairie SSReflect :

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

où la soustraction est tronquée et == désigne un test d'égalité booléen, alors pour toutes valeurs connues de `m` et `n` telles que $m \leq n$, on peut prouver `leq m n = true` par réflexivité de l'égalité : cette proposition est équivalente à `true = true` grâce à la règle (conv) ci-dessus. Si maintenant on dispose d'une preuve `p : forall m n, leq m n = true -> m <= n`, alors une preuve de `m <= n` est fournie par le terme `p m n (eq_refl true)`. Ainsi, `p` est une preuve générique qui peut être appliquée à toutes les inégalités closes, et dont la taille est indépendante de `m` et `n`. Cette approche est portée à grande échelle notamment dans la preuve du théorème des quatre couleurs et adoptée à petite échelle dans la librairie SSReflect.

Les tests de convertibilités mis en jeu dans le cadre des preuves par réflexion nécessitent la plupart du temps d'évaluer les termes jusqu'à leur forme normale. Les environnements d'exécution des langages de programmation fonctionnels seraient donc tout à fait appropriés pour mener cette évaluation. Cependant, cette stratégie rencontre deux problèmes :

- Dans la plupart des langages de programmation conventionnels, les valeurs ne peuvent être analysées et comparées que pour les types de base (listes, entiers,...). Les habitants des types fonctionnels se comportent comme des boîtes noires qui ne peuvent être comparées.

- Les programmes d'un langage fonctionnel sont toujours des termes clos, alors que dans notre contexte, nous pouvons être amenés à comparer (et donc évaluer) des termes ouverts (avec des variables libres faisant référence à des hypothèses du contexte de typage).

Ces contraintes permettent aux environnements d'exécution de prendre en charge seulement des substitutions de termes clos dans des termes clos, ouvrant la porte à des stratégies d'implémentation efficaces et évitant tout problème de capture de nom. Mais dans notre contexte, les formes normales ne peuvent pas toujours être atteintes uniquement par des substitutions closes. Seules les formes normales de tête faibles sont calculées, autrement dit elles sont seulement *faiblement réduites*. Par exemple, le terme $\lambda x.(\lambda y.y)x$ est en forme normale de tête faible, mais pour les règles de réduction du C.C.I., sa forme normale est $\lambda x.x$.

Notre objectif est d'implémenter la *réduction forte* en forme normale de termes potentiellement ouverts, à types dépendants et à l'ordre supérieur, en nous rapprochant des performances et du niveau d'optimisation d'un compilateur optimisant pour un langage de programmation ordinaire. Nous évitons pour ce faire de modifier un compilateur existant, et *a fortiori* d'en écrire un nous-mêmes, en réutilisant tel quel le compilateur OCaml.

Ce choix de conception suit une longue tradition d'approches à la normalisation utilisant des composants génériques dont la *Normalisation par Évaluation* (N.p.E.) fait partie. L'idée est d'obtenir les formes normales voulues non pas par l'itération usuelle d'une relation de réduction à un pas, mais plutôt en construisant un modèle *résidualisant* D de l'ensemble des termes Λ, donné par une dénotation $[\![\cdot]\!] : \Lambda \to D$ admettant un inverse $\downarrow : D \to \Lambda$ (appelé *réification*) tel que :

1. si $t \to t'$ alors $[\![t]\!] = [\![t']\!]$ (correction) ;
2. si t est un terme en forme normale, alors $\downarrow[\![t]\!] = t$ (reproduction).

Nous notons $\to^*$ pour la fermeture réflexive et transitive de la relation de réduction $\to$. Il est facile de voir que si $t \to^* t'$ où t' est en forme normale alors $\downarrow[\![t]\!] = \downarrow[\![t']\!] = t'$, donc la composée de l'interprétation et de la réification donne une fonction de normalisation pour les termes normalisants. Dans la N.p.E. dite typée et l'évaluation partielle dirigée par les types [O. Danvy, 1996], la réification est en fait guidée par les types. Mais de telles approches doivent être adaptées à chaque système de types et les étendre aux systèmes expressifs comme le Calcul des Constructions Inductives est non-trivial. Des variantes non-typées de N.p.E. ont été proposées [A. Filinski et H. K. Rohde, 2004 ; S. Lindley, 2005 ; K. Aehlig et al., 2008 ; M. Boespflug, 2010], mais comme nous le verrons, la généralité des approches non-typées avait jusqu'à maintenant un coût du fait de l'ajout d'étiquettes à la syntaxe cible de l'interprétation afin de permettre son plongement profond dans le langage hôte. Même si l'impact de cet étiquetage sur les performances peut être réduit dans de nombreux cas [M. Boespflug, 2010], une partie de ce surcoût demeure. L'allocation mémoire et la localité du code sont impactées négativement et des optimisations habituelles des compilateurs (comme la décurrification) doivent être refaites au niveau de la syntaxe cible de l'interprétation.

L'implémentation de la réduction forte que nous décrivons ici allie la généralité de la N.p.E. non-typée, car elle s'applique à tous les termes (ouverts) du λ-calcul, et la performance de la N.p.E. typée, puisque l'interprétation des termes introduit un surcoût quasi-nul. Il n'est donc plus besoin

de choisir entre généralité et performance. Notre approche s'inspire de [B. Grégoire et X. Leroy, 2002], mais contrairement à ce travail antérieur qui repose sur une version modifiée du compilateur et de la machine virtuelle OCaml, nous parvenons à une généricité totale vis-à-vis de l'évaluateur sous-jacent. Nous n'avons pas besoin de maintenir une version spécifique de cet évaluateur, ce qui signifie une meilleure séparation des problèmes entre le développement d'assistants à la preuve et celui de compilateurs.

Ce chapitre est structuré comme suit. Dans la section 1.1, nous proposons une vue unifiée de la normalisation par évaluation non typée et de l'algorithme de normalisation de [B. Grégoire et X. Leroy, 2002], mettant en évidence que le second est une instance de la première. Nous montrons également comment implémenter cet algorithme par traduction du langage source vers un langage fonctionnel. Puis, nous généralisons cet algorithme à la réduction forte de termes du C.C.I. (section 1.2). Nous continuons par l'ajout de types de données co-inductifs, d'entiers machines et de tableaux persistants (section 1.5). Ces encodages sont, à notre connaissance, des fonctionnalités nouvelles dans un algorithme de N.p.E.. Dans la section 1.6, nous montrons à travers un certain nombre de cas d'utilisation pratiques que notre solution se compare très favorablement aux meilleures implémentations antérieures de la règle de conversion dans Coq, atteignant typiquement une accélération d'un facteur cinq.

1.1 APPLICATION AU λ-CALCUL

1.1.1 Le calcul de réduction symbolique

Trouver la forme normale d'un terme t par évaluation repose sur la capacité à distinguer la tête de la valeur de t et à continuer la réduction dans le corps de l'abstraction si la valeur de t est de la forme $\lambda x.\, t'$. Cependant, t' n'est pas, en général, un terme clos (x peut apparaître libre), et n'est donc pas pris en charge par les évaluateurs des langages fonctionnels.

Le calcul symbolique de [B. Grégoire et X. Leroy, 2002] introduit une nouvelle classe de valeurs pour représenter les variables libres et les termes dont l'évaluation est « gelée » à cause d'une variable libre en position de tête, ainsi qu'une nouvelle règle de réduction. L'idée clé est que la réduction faible des termes ouverts peut être simulée par réduction faible de termes symboliques clos. La syntaxe du calcul symbolique est la suivante :

$$\begin{aligned} \textbf{Term} \ni t &::= x \mid t_1\ t_2 \mid v \\ \textbf{Val} \ni v &::= \lambda x.\, t \mid [\tilde{x}\ v_1 \dots v_n] \end{aligned}$$

Nous faisons le choix de l'appel par valeur dans le reste de ce chapitre, mais nous aurions aussi bien pu choisir n'importe quelle stratégie d'évaluation standard, comme l'appel par nécessité. De toute façon, comme le C.C.I. est fortement normalisant et confluant, le résultat est indépendant de la stratégie d'évaluation.

où $[\tilde{x}\ v_1 \dots v_n]$ est une valeur, appelée *accumulateur*, représentant la variable libre x appliquée aux arguments $v_1, \dots, v_n$. Les règles de réduction du calcul sont :

$$\begin{aligned} (\lambda x.\, t)\ v &\to t\{x \leftarrow v\} && (\beta_v) \\ [\tilde{x}\ v_1 \dots v_n]\ v &\to [\tilde{x}\ v_1 \dots v_n\ v] && (\beta_s) \\ \Gamma(t) &\to \Gamma(t') \quad \text{si } t \to t' \quad (\text{avec } \Gamma ::= t\ [\,] \mid [\,]\ v) && \text{contexte} \end{aligned}$$

La règle β_v est la règle standard de β-réduction en appel par valeur, la règle de contexte autorise la réduction dans tout sous-terme qui n'est pas une abstraction (c'est la réduction faible). Enfin, la règle β_s exprime le fait

que les variables libres se comportent comme des boîtes accumulant leurs arguments lorsqu'elles sont appliquées.

Nous définissons la valeur $\mathcal{V}(t)$ d'un terme symbolique clos t comme la forme normale de t pour la relation $\to^*$. Comme des suites infinies de réductions sont possibles, cette forme normale n'existe pas nécessairement. Cependant, si la forme normale existe alors elle doit être une valeur parce que la réduction $\to$ ne peut pas être gelée sur des termes symboliques clos [B. Grégoire et X. Leroy, 2002].

Étant donnée une traduction $\ulcorner\cdot\urcorner$ qui à un terme t associe un terme symbolique $\ulcorner t\urcorner$, nous pouvons exprimer précisément comment obtenir la forme normale de t par rapport à la β-réduction, par itération de phases successives de réduction symbolique faible et de « relecture » : d'abord, on calcule $\mathcal{V}(\ulcorner t\urcorner)$ par réduction symbolique faible (équation 1.1) ; ensuite, on inspecte la valeur résultat et on normalise récursivement les sous-termes (*relecture*).

$$\mathcal{N}(t) = \mathcal{R}(\mathcal{V}(\ulcorner t\urcorner)) \tag{1.1}$$

$$\mathcal{R}(\lambda x.\, t) = \lambda y.\, \mathcal{R}(\mathcal{V}((\lambda x.t)\ [\tilde{y}])) \text{ où } y \text{ est fraîche} \tag{1.2}$$

$$\mathcal{R}[\tilde{x}\ v_1 \dots v_n] = x\ \mathcal{R}(v_1) \dots \mathcal{R}(v_n) \tag{1.3}$$

L'algorithme de normalisation prend un terme symbolique clos et renvoie un λ-terme en forme normale. Si la valeur est un accumulateur, la fonction de relecture injecte la variable symbolique $\tilde{x}$ vers la variable source x et applique celle-ci à la relecture des arguments accumulés (équation 1.3).

Si la valeur est une abstraction $\lambda x.\, t$, la fonction de relecture normalise l'application de la fonction à une variable symbolique fraîche[1] (équation 1.2). On note que l'application se réduit en une étape vers $t\{x \leftarrow [\tilde{y}]\}$, donc $\mathcal{R}(\mathcal{V}((\lambda x.\, t)\ [\tilde{y}]))$ renverra la forme normale de t modulo le renommage de x en y. Cette astuce est centrale dans la réutilisation des évaluateurs conventionnels pour la réduction symbolique : elle implique qu'il n'est pas nécessaire de pouvoir examiner le corps des fonctions. Celles-ci sont, comme d'habitude, des boîtes noires dont le seul comportement observable est déclenché par leur application à une valeur.

[1] *La condition de fraîcheur peut être rendue précise par l'utilisation de* niveaux *de De Bruijn pour les variables symboliques.*

Exemple 1.1. *Considérons le terme* $t = \lambda x.\, (\lambda y.\, y)\ x$. *Par abus de notation, nous confondons le terme d'origine* t *et sa traduction symbolique* $\ulcorner t\urcorner$. *Pour calculer sa forme normale* $\mathcal{N}(t)$, *une première évaluation symbolique faible est effectuée, mais le terme étant une valeur fonctionnelle, il ne peut être réduit. Une phase de relecture intervient alors :* $\mathcal{N}(t) = \mathcal{R}(\lambda x.\, (\lambda y.\, y)\ x) = \lambda u.\, \mathcal{R}(\mathcal{V}((\lambda x.\, (\lambda y.\, y)\ x)\ [\tilde{u}]))$, *ce qui déclenche une réduction symbolique :* $(\lambda x.\, (\lambda y.\, y)\ x)\ [\tilde{u}] \to \lambda y.\, y\ [\tilde{u}] \to [\tilde{u}]$. *On obtient donc finalement* $\mathcal{N}(t) = \lambda u.\, \mathcal{R}([\tilde{u}]) = \lambda u.\, u$, *qui est bien la forme normale attendue, modulo renommage des variables liées.*

Il est facile de faire le lien avec le cadre de la Normalisation par Évaluation que nous avons présenté en introduction de ce chapitre : notre dénotation est définie par $[\![t]\!] = \mathcal{V}(\ulcorner t\urcorner)$ et notre réification n'est autre que la fonction de relecture $\mathcal{R}(\cdot)$.

Dans [B. Grégoire et X. Leroy, 2002], les auteurs ont recours à une modification de la machine virtuelle OCaml pour obtenir un évaluateur efficace pour le calcul symbolique. Ceci a un inconvénient : tandis que l'effort d'implémentation est réduit, la nouvelle machine abstraite et le compilateur associé doivent être maintenus séparément, *ad infinitum*. En outre, l'efficacité est limitée à ce que peuvent apporter les machines abstraites, c'est-à-dire souvent beaucoup moins que la compilation en code natif. Nous aurions pu tenter de développer un nouveau compilateur « ahead-of-time » ou « just-in-time » pour cette machine virtuelle, mais le bénéfice par rapport à la

réutilisation d'un compilateur existant vers du code natif, pour un langage existant, aurait probablement été faible pour un coût d'implémentation bien plus élevé. Dans la prochaine section, nous présentons une interface modulaire que nous instancions de deux façons, donnant lieu à deux implémentations du calcul symbolique décrit ci-dessus, toutes les deux comme programmes OCaml standard. La première est inspirée par la syntaxe abstraite d'ordre supérieur et la deuxième utilise la possibilité de manipuler en OCaml la représentation mémoire des valeurs.

1.1.2 Cadre abstrait

Pour effectuer la normalisation d'un terme source (ici un λ-terme), nous le traduisons d'abord vers un programme de notre langage cible (ici OCaml) qui calcule une valeur. Ensuite, par inspection de la tête de la valeur obtenue, nous relisons celle-ci vers un terme source en forme normale. Pour ce faire, nous supposons avoir un module pour les valeurs avec l'interface suivante :

Le type `var` *est abstrait et représente les variables libres de notre calcul. Son implémentation importe peu ; dans le cas de* Coq*, c'est essentiellement le type* `string`*.*

```
module type Values = sig
  type t
  val app : t -> t -> t
  type atom =
    | Var of var
  type head =
    | Lam of t -> t
    | Accu of atom * t list
  val head : t -> head
  val mkLam : (t -> t) -> t
  val mkAccu : atom -> t
end
```

La première composante de la signature est le type représentant les valeurs. Nous supposons qu'étant données deux valeurs, nous sommes capables de calculer la valeur correspondant à l'application de l'une à l'autre (fonction `app`). Deuxièmement, nous supposons pouvoir distinguer la tête de n'importe quelle valeur (fonction `head`). Dans le cas du λ-calcul, une valeur est soit une λ-abstraction (constructeur `Lam`) ou un accumulateur qui est un atome appliqué à ses arguments[2] (constructeur `Accu`). Enfin, nous postulons l'existence d'une fonction injectant les atomes dans les termes. Nous supposons que les abstractions sur les termes sont représentées comme des fonctions OCaml, avec `mkLam` qui injecte les fonctions dans les termes.

[2] *Les arguments sont stockés dans l'ordre inverse pour permettre d'étendre efficacement la liste des arguments lorsqu'un accumulateur est appliqué.*

Il peut sembler redondant d'avoir les deux types `t` et `head` et de distinguer les constructeurs `Lam` et `Accu` des fonctions `mkLam` et `mkAccu`. Il est en effet parfois possible de confondre les deux, comme nous le verrons à la section 1.1.3. Cependant, ce ne sera plus le cas à la section 1.1.4, où nous manipulerons directement la représentation des valeurs du type `t`, en prenant quelques libertés avec le système de types d'OCaml. Le type `head` permet alors de maintenir une *vue* algébrique sur le type `t`. Les lois suivantes doivent être vérifiées :

```
head (mkAccu a) = Accu(a,[])
 head (mkLam f) = Lam f
```

La compilation d'un λ-terme vers un programme OCaml est définie de la façon suivante :

$$
\begin{aligned}
\ulcorner x \urcorner^{B} &= \begin{cases} x & \text{if } x \in B \\ \texttt{mkAccu(Var } x\texttt{)} & \text{sinon} \end{cases} \\
\ulcorner \lambda x.\, t \urcorner^{B} &= \texttt{mkLam (fun } x \texttt{ -> } \ulcorner t \urcorner^{B \cup \{x\}}\texttt{)} \\
\ulcorner t_1\ t_2 \urcorner^{B} &= \texttt{app } \ulcorner t_1 \urcorner^{B}\ \ulcorner t_2 \urcorner^{B}
\end{aligned}
$$

Le compilateur prend en entrée un λ-terme t et un ensemble de variables liées B, et renvoie un programme OCaml calculant la forme normale de tête faible de t, vue comme valeur symbolique. Les variables liées dans t sont compilées vers des variables OCaml. Dans le cas d'une variable libre x, le code construit l'accumulateur $[\tilde{x}]$ correspondant à la variable symbolique $\tilde{x}$. La compilation d'une abstraction construit une fonction OCaml (l'ensemble B de variables liées est étendu avec la variable liée x). Pour l'application, nous utilisons la fonction `app` pour appliquer la partie fonctionnelle à l'argument.

L'algorithme de normalisation est ainsi une traduction directe de l'algorithme de normalisation présenté à la section précédente. Pour être précis, nous scindons la fonction de relecture en trois fonctions $\mathcal{R}_V$, $\mathcal{R}$ et $\mathcal{R}_A$ opérant respectivement sur les types `Values.t`, `head` et `atom`. Elles sont définies comme suit :

$$
\begin{aligned}
\mathcal{N}(t) &= \mathcal{R}_V(\ulcorner t \urcorner^{\emptyset}) \\
\mathcal{R}_V(v) &= \mathcal{R}(\texttt{head } v) \\
\mathcal{R}(\texttt{Lam f}) &= \lambda y.\, \mathcal{R}_V(\texttt{f (mkAccu (Var } y\texttt{))}) \text{ où } y \text{ est fraîche} \\
\mathcal{R}(\texttt{Accu(}a\texttt{,[}v_n\texttt{;...;}v_1\texttt{])}) &= \mathcal{R}_A(a)\ \mathcal{R}_V(v_1) \ldots \mathcal{R}_V(v_n) \\
\mathcal{R}_A(\texttt{Var } x) &= x
\end{aligned}
$$

1.1.3 Normalisation avec étiquettes

Une implémentation naturelle pour le type `t` du module `Values`, suggérée dans [K. Aehlig et al., 2008 ; M. Boespflug, 2010], consiste à utiliser le type `head` directement :

```
type t = head
let head v = v
let app t v = match t with
  | Lam f -> f v
  | Accu(a, args) -> Accu(a,v::args)
let mkLam f = Lam f
let mkAccu a = Accu(a,[])
```

Dans ce cas, l'implémentation découle en grande partie des lois énoncées plus haut. Si le premier argument modélise une abstraction, la fonction `app` doit extraire la fonction représentée et effectuer la substitution (modélisée par une application de valeurs OCaml). Sinon, le premier argument est un accumulateur. Le fait qu'un accumulateur a été appliqué à `v` est mémorisé en étendant la liste des arguments de l'accumulateur avec ce nouvel argument.

La représentation que nous avons décrite se distingue par une implémentation succincte. Cependant, l'étiquetage explicite de toutes les valeurs pour signaler la forme de leur tête a un coût mesurable en termes de performances. Plusieurs optimisations sont suggérées dans [M. Boespflug, 2010]

pour atténuer l'impact de cette représentation coûteuse de l'application, comme la décurrification ou la spécialisation des constructeurs. Bien que l'amélioration soit significative, nous préférons réduire à néant le coût de l'étiquetage en éliminant les étiquettes. Nous montrons dans la section suivante comment encoder les accumulateurs comme des fonctions spéciales d'arité infinie. Les applications ne doivent plus inspecter les têtes, étant donnée la convention d'appel uniforme à la fois pour les fonctions ordinaires et les accumulateurs.

1.1.4 Normalisation sans étiquettes

Dans [B. Grégoire et X. Leroy, 2002], les auteurs remarquent déjà qu'un accumulateur peut être vu comme une fonction primitive gardant trace de tous les arguments qui lui sont fournis. Nous montrons comment écrire une telle fonction de manière interne en OCaml, sans étendre le langage avec une nouvelle primitive.

L'environnement d'exécution OCaml manipule un assortiment de différents types de valeurs : entiers, flottants, pointeurs vers des tableaux, valeurs construites d'un type de données défini par l'utilisateur, clôtures, etc. À un niveau plus bas, les entiers sont distingués des pointeurs vers des blocs alloués sur le tas par la valeur de leur bit de poids faible, qui vaut toujours 1 dans le cas des entiers.[3] Un bloc mémoire, écrit $[T : v_0; \ldots; v_n]$, est composé de son étiquette T (un petit entier) et de ses champs $v_0, \ldots, v_n$. Les clôtures sont encodées par un bloc $[T_\lambda : C; v_1; \ldots; v_n]$ où le premier champ C est un pointeur de code et $v_1, \ldots, v_n$ sont les valeurs associées aux variables libres de la fonction (c'est-à-dire l'environnement de la clôture).

[3] *Cette information est utilisée par le glaneur de cellules, et est la raison de la limitation des entiers* OCaml *à 31 (resp. 63) bits sur une architecture 32 (resp. 64) bits.*

Dans [B. Grégoire et X. Leroy, 2002], les accumulateurs sont représentés en utilisant la même architecture que les clôtures : $[0 : \texttt{ACCU}; k]$ où k est la représentation mémoire de l'accumulateur et `ACCU` est un pointeur de code vers une seule instruction. Lorsqu'elle est appliquée à un argument, cette instruction construit un bloc accumulateur frais contenant la représentation de k appliquée au nouvel argument. L'avantage majeur de cette technique est que le schéma de compilation de l'application est inchangé (les accumulateurs peuvent être vus comme des clôtures), donc aucun coût ne pèse sur l'évaluation d'une application. En particulier, il n'y aucun impact sur l'évaluation de termes clos. Un second avantage est que l'étiquette utilisée pour les blocs accumulateurs est 0, ce qui permet de distinguer le type de bloc obtenu par une simple inspection de l'étiquette (l'étiquette T_λ utilisée pour les clôtures n'est pas 0).

Notre idée est d'utiliser la même astuce mais directement en OCaml. Il faut se souvenir qu'un accumulateur est une fonction attendant un argument, accumulant cet argument et renvoyant récursivement un accumulateur. Notre module `Values` de la section 1.1.2 peut être instancié en définissant une telle fonction comme suit :

L'option `-rectypes` *est nécessaire pour que le type récursif* `t` *soit autorisé.*

```
type t = t -> t
let rec accu atom args = fun v -> accu atom (v::args)
let mkAccu atom = accu atom []
```

Étant donné un atome `a`, la valeur de `mkAccu a` est une fonction attendant un argument `v`. Cet argument est stocké dans la liste `args`, et le résultat est lui-même un accumulateur. Cette définition est naturelle, mais l'étiquette de l'objet renvoyé est T_λ et non 0. Heureusement, l'étiquette des objets peut

être changée en utilisant la fonction `set_tag` du module `Obj` d'OCaml. Cela nous amène au code suivant pour `accu` :

```
let rec accu atom args =
  let res = fun v -> accu atom (v::args) in
  Obj.set_tag (Obj.repr res) 0; (res : t)
```

Le résultat est un objet de type `t` et son étiquette est maintenant `0`. Enfin, nous devons écrire la fonction `head`. Pour cela, nous inspectons l'étiquette de la valeur : si l'étiquette n'est pas `0` alors il s'agit d'une clôture donc nous renvoyons la valeur elle-même ; si l'étiquette est `0`, nous devons obtenir l'atome et les arguments accumulés. L'atome est stocké à la position 3 dans la clôture et la liste des arguments à la position 4. La fonction `Obj.field` permet de les récupérer. Cela aboutit au code suivant :

```
type t = t -> t
let app f v = f v
let mkLam f = f
let getAtom o = (Obj.magic (Obj.field o 3)) : atom
let getArgs o = (Obj.magic (Obj.field o 4)) : t list
let head (v:t) =
  let o = Obj.repr v in
  if Obj.tag o = 0 then Accu(getAtom o, getArgs o) else Lam(v)
```

Il faut remarquer que la fonction `app` effectue simplement l'application (sans filtrage sur la partie fonctionnelle) et que la fonction `mkLam` est l'identité. En pratique, ces opérateurs sont dépliés par le compilateur et disparaissent donc du code produit en sortie. Les étiquettes de l'implémentation précédente jouaient deux rôles : elles permettaient à `App f t` d'avoir le bon comportement selon que `f` était une fonction ou un accumulateur, et guidaient la relecture. Avec notre implémentation sans étiquettes, nous avons rendue uniforme la convention d'appel pour les fonctions et les accumulateurs, et reposons sur l'environnement d'exécution du langage cible pour informer la relecture. Nous n'avons plus besoin des étiquettes pendant la relecture car l'environnement d'exécution peut déjà distinguer les différents types de valeurs. Enfin, la présence d'opérations non sûres, comme `Obj.repr` et `Obj.magic` (qui sont des fonctions identité non typées), n'est pas source d'inquiétude dans la mesure où notre langage *source* est typé, donc nous n'avons pas de besoin particulier de sûreté dans notre langage *cible*.

1.2 EXTENSION AU CALCUL DES CONSTRUCTIONS INDUCTIVES

Dans cette section, nous étendons notre approche au Calcul des Constructions Inductives. La décidabilité de la vérification de types est vérifiée seulement pour des termes *avec domaine*, où les variables sont toutes explicitement annotées avec leur type au niveau du lieur. Cependant, il est toujours possible d'effacer les annotations sur les termes du C.C.I. pour tester la convertibilité [B. Barras et B. Grégoire, 2005]. De plus, étant donné un terme (bien typé) en forme normale,[4] avec des annotations effacées, il est possible de retrouver ces annotations à partir de son type. Nous considérons donc seulement une variante *sans domaine* du C.C.I..

[4] *Cette hypothèse est nécessaire, il n'est pas possible par exemple de retrouver les annotations dans le terme* $(\lambda \texttt{f.true})(\lambda x.x)$ *de type* `bool`.

1.2.1 Le C.C.I. symbolique

La syntaxe du calcul symbolique est étendue avec des sortes, des produits dépendants, des types inductifs, des constructeurs, du filtrage et des points fixes :

$$\begin{aligned}
\textbf{Term} \ni t, T, P ::= {} & x \mid t_1\ t_2 \mid v \mid C_i(\vec{t}) \mid \textbf{match}_{\langle P\rangle}\ t\ \textbf{with}\ (C_i(\vec{x_i}) \to t_i)_{i\in I} \\
& \mid \textbf{fix}_m\ (f : T := t) \\
\textbf{Val} \ni v ::= {} & \lambda x.t \mid [k\,\vec{v}] \mid C_i(\vec{v}) \\
\textbf{Atom} \ni k ::= {} & \tilde{x} \mid s \mid I \mid \Pi x : t.t \mid \textbf{match}_{\langle P\rangle}\ k\ \textbf{with}\ (C_i(\vec{x_i}) \to t_i)_{i\in I} \\
& \mid \textbf{fix}_m\ (f : T := t)
\end{aligned}$$

Les seules informations de types que nous conservons sont le prédicat de retour du filtrage (noté P dans $\textbf{match}_{\langle P\rangle}\ t\ \textbf{with}\ (C_i(\vec{x_i}) \to t_i)_{i\in I}$) et le type du corps des points fixes. Ces annotations ne peuvent être enlevées sans changer la notion de convertibilité [B. Barras et B. Grégoire, 2005].

Il est important de noter que nous ne représentons ici que les constructeurs totalement appliqués, bien que le formalisme de Coq permette l'application partielle de constructeurs. En effet, puisque la règle η est admissible pour le Calcul des Constructions Inductives (et implémentée dans Coq depuis la version 8.4), nous pouvons effectuer des η-expansions où c'est nécessaire pour préserver cet invariant. Les règles de réduction du calcul symbolique sont étendues par :

$$\begin{aligned}
&\textbf{match}_{\langle P\rangle}\ C_i(\vec{v})\ \textbf{with}\ (C_i(\vec{x_i}) \to t_i))_{i\in I} \to_\iota t_i\{\vec{x_i} \leftarrow \vec{v}\} && (\iota_v^1) \\
&\quad \textbf{match}_{\langle P\rangle}\ [k]\ \textbf{with}\ (C_i(\vec{x_i}) \to t_i)_{i\in I} \to_\iota && (\iota_s^1) \\
&\qquad\qquad [\textbf{match}_{\langle P\rangle}\ k\ \textbf{with}\ (C_i(\vec{x_i}) \to t_i)_{i\in I}] \\
&\quad \textbf{fix}_m\ (f : T := t)v_1 \dots v_{m-1} C_i(\vec{v}) \to_\iota && (\iota_v^2) \\
&\qquad\qquad t\{f \leftarrow \textbf{fix}_m\ (f : T := t)\}\, v_1 \dots v_{m-1}\ C_i(\vec{v}) \\
&\quad \textbf{fix}_m\ (f : T := t)v_1 \dots v_{m-1}\ [k] \to_\iota && (\iota_s^2) \\
&\qquad\qquad [\textbf{fix}_m\ (f : T := t)v_1 \dots v_{m-1}\ k]
\end{aligned}$$

Les règles ι_v^1 et ι_v^2 sont les règles usuelles de réduction pour l'analyse par cas et les points fixes dans le C.C.I.. Les points fixes se réduisent seulement si leur argument récursif (dont l'index est dénoté par m) est un constructeur. Ceci empêche le dépliage infini de points fixes pendant la normalisation, ce qui ne permettrait plus d'assurer la décidabilité de la vérification de types. Les règles ι_s^1 et ι_s^2 sont les pendants symboliques, et prennent en charge le cas où l'argument est un accumulateur. Un nouvel accumulateur est alors créé pour représenter l'application d'une analyse par cas ou d'un point fixe qui ne peut plus être réduite.

$$\begin{aligned}
&\mathcal{R}_A(\textbf{match}_{\langle P\rangle}\ k\ \textbf{with}\ (C_i(\vec{x_i}) \to t_i)_{i\in I}) = \\
&\qquad \textbf{match}_{\langle P\rangle}\ \mathcal{R}(k)\ \textbf{with}\ (C_i(\vec{x_i}) \to \mathcal{R}(f(C_i(\overrightarrow{[x_i]}))))_{i\in I} \\
&\qquad \text{où } f = \lambda x.\textbf{match}_{\langle P\rangle}\ x\ \textbf{with}\ (C_i(\vec{x_i}) \to t_i)_{i\in I} \\
&\mathcal{R}_A(\textbf{fix}_m\ (f : T := t)) = \textbf{fix}_m\ (f : T := \mathcal{R}((\lambda f.t)\,[\tilde{f}]))
\end{aligned}$$

Figure 1.1 – Algorithme de relecture pour le filtrage et les points fixes

La figure 1.1 décrit l'algorithme de relecture. Relire un accumulateur représentant une analyse par cas requiert de normaliser les branches. Comme dans le cas des abstractions, les corps ne peuvent pas être accédés directement, d'où la nécessité d'appliquer l'expression pour déclencher la réduction. Plus précisément, si $\textbf{match}_{\langle P \rangle}\ t\ \textbf{with}\ (C_i(\vec{x_i}) \to t_i)_{i\in I}$ est en forme normale de tête faible (t est un accumulateur), nous relisons successivement l'application du terme $\lambda x.\textbf{match}_{\langle P \rangle}\ x\ \textbf{with}\ (C_i(\vec{x_i}) \to t_i)_{i\in I}$ à chaque constructeur $C_i(\vec{x_i})$ où les $\vec{x_i}$ sont des accumulateurs représentant des variables libres, en nombre égal à l'arité de C_i.

1.2.2 Traduction des termes

La signature de notre module `Values` doit être étendue en conséquence pour représenter toutes les têtes possibles et toutes les formes des accumulateurs.

Les types `var`, `sort` *et* `inductive` *sont abstraits et représentent respectivement des noms de variable libre, des niveaux d'univers et des noms de types inductifs. Nous utilisons l'implémentation de ces types fournie par le noyau de* Coq.

```
module type Values = sig
  type t
  val app : t -> t -> t
  type atom =
    | Var of var
    | Sort of sort
    | Ind of inductive
    | Prod of t * t
    | Match of annot * t * t * (t -> t)
    | Fix of (t -> t) * t * int
  type head =
    | Accu of atom * t list
    | Lam of t -> t
    | Construct of int * t array
  val head : t -> head
  val mkLam : (t -> t) -> t
  val mkAccu : atom -> t
  val mkConstruct : int -> t array -> t
end
```

Ici encore, nous ne stockons que les informations pertinentes pour le test de conversion. Un constructeur est identifié par un index dans la liste des constructeurs du type inductif auquel il appartient, et porte un vecteur d'arguments. Le filtrage (constructeur `Match`) est caractérisé par une annotation contenant notamment le nombre de branches, par le terme analysé, le prédicat qui exprime le type de retour, et enfin les branches.

Le schéma de compilation est étendu en conséquence, comme montré dans la figure 1.2, où `is_accu` est une simple fonction auxiliaire définie par :

```
let is_accu v = match head v with
  | Accu _ -> true
  | _ -> false
```

La compilation du filtrage construit une clôture récursive qui peut se réduire vers une branche si elle est appliquée à une constructeur ou sinon vers un accumulateur stockant toutes les informations nécessaires à la réification. En particulier, la clôture récursive `case` est stockée dans l'accumulateur. Cette fonction joue le rôle de $\lambda x.\textbf{match}_{\langle P \rangle}\ x\ \textbf{with}\ (C_i(\vec{x_i}) \to t_i)_{i\in I}$ dans l'algorithme de relecture (c.f. figure 1.1). L'utilisation d'une telle clôture ré-

$$
\begin{array}{rcl}
\ulcorner s \urcorner^B & = & \texttt{mkAccu (Sort s)} \\
\ulcorner \Pi x : T.U \urcorner^B & = & \texttt{mkAccu (Prod(}\ulcorner T \urcorner^B, \ulcorner \lambda x.U \urcorner^B\texttt{))} \\
\ulcorner C_i(\vec{t}) \urcorner^B & = & \texttt{mkConstruct i [|}\ulcorner \vec{t} \urcorner^B\texttt{|]}
\end{array}
$$

$\ulcorner$**match**$_{\langle P \rangle}$ t **with** $(C_i(\vec{x_i}) \to t_i))_{i\in I}\urcorner^B$ =

```
let rec case c =
  match c with
  | ⌜C1(x1)⌝^{B∪{x1}} -> ⌜t1⌝^{B∪{x1}}
  | ...
  | ⌜Cn(xn)⌝^{B∪{xn}} -> ⌜tn⌝^{B∪{xn}}
  | _ -> mkAccu Match(Ĩ,c,⌜P⌝^B,case)
in case ⌜t⌝^B
```

$\ulcorner$**fix**$_m$ $(f : T := t)\urcorner^B$ =

```
let fnorm f = ⌜t⌝^{B∪{f}} in
let rec f =
  mkLam (fun x1 -> ... -> mkLam (fun xm ->
  if is_accu xm then
    mkAccu (Fix (fnorm,⌜T⌝^B,m)) x1 ... xm
  else fnorm f x1 ... xm) ... )
in f
```

FIGURE 1.2 – Schéma de compilation du calcul symbolique étendu

cursive évite une duplication exponentielle du code en présence de filtrages de motifs imbriqués.

Une astuce similaire est utilisée pour les points fixes : la fonction `fnorm` paramétrée par `f` encapsule le corps et le point fixe lui-même est représenté par une clôture récursive attendant m arguments. Si le m-ième argument est un accumulateur, alors la règle ι_s^2 s'applique, sinon le point fixe est réduit comme dans ι_v^2.

Suivant notre première approche (avec étiquetage explicite), une implémentation concrète des constructeurs pourrait être :

```
mkConstruct i args = Construct(i,args)
```

À la place, nous faisons correspondre les types et constructeurs du C.C.I. aux types de données algébriques et constructeurs du langage hôte, évitant ainsi un surcoût d'allocation, des indirections superflues et bénéficiant de l'implémentation efficace du filtrage du langage hôte. Ainsi, le type inductif suivant :

```
Inductive I := C1 : T1 | ... | Cn : Tn
```

est traduit par :

```
type I = Accu_I of t | C1 of t * ⋯ * t | ... | Cn of t * ⋯ * t
```

où les signatures font correspondre les arités de chaque constructeur. Cela nous permet d'interpréter un constructeur dans le terme source par un constructeur dans le langage hôte :

```
mkConstruct i v⃗ = Obj.magic Ci(v⃗) : t
```

OCAML représente les constructeurs non-constants par un bloc mémoire et les distingue selon leur étiquette. Puisque `Accu_I` est le premier constructeur non constant du type généré, l'étiquette 0 lui sera attribuée. Ceci est

compatible avec notre fonction `head`, dont l'implémentation repose sur le fait que nous avons réservé l'étiquette 0 pour les accumulateurs.

1.3 TYPES CO-INDUCTIFS

Le Calcul des Constructions (Co)Inductives supporte la définition de données co-récursives, qui peuvent être construites en utilisant des constructeurs de types de données co-inductifs ou des co-points-fixes. Les données co-récursives peuvent être des objets infinis comme les éléments du type `stream` :

```
CoInductive stream := Cons : nat -> stream -> stream.
CoFixpoint ones := Cons 1 ones.
CoFixpoint nats x := Cons x (nats (1+x)).
```

Le co-point-fixe `ones` représente la liste infinie de `1` et `nats x` la liste infinie d'entiers naturels commençant à `x`. Pour empêcher un dépliage infini, l'évaluation des co-points-fixes est paresseuse. Cela signifie que `ones` est en forme normale, comme l'est `nats 0`. Seul le filtrage peut forcer l'évaluation d'un co-point-fixe, la règle de réduction étant la suivante :

La condition de garde assure que la réduction d'un co-point-fixe produit toujours un constructeur.

$$\textbf{match}_{\langle P\rangle}\ \textbf{cofix}\ (f := t)\vec{a}\ \textbf{with}\ (C_i(\vec{x_i}) \to t_i)_{i\in I} \to \\ \textbf{match}_{\langle P\rangle}\ (t\{f \leftarrow \textbf{cofix}\ (f := t)\})\vec{a}\ \textbf{with}\ (C_i(\vec{x_i}) \to t_i)_{i\in I}$$

Une implémentation directe de la règle de réduction conduirait à une stratégie d'évaluation inefficace, puisqu'il n'y a pas de partage entre des évaluations multiples de $c\,\vec{a}$. Pour obtenir une stratégie efficace pour les co-points-fixes, nous utilisons la même idée qu'OCaml, qui consiste grossièrement à représenter un terme de type `'a Lazy.t` par une référence soit vers une valeur de type `'a` (lorsque le terme a été forcé), soit vers une fonction de type `unit -> 'a`. Lorsqu'une valeur paresseuse est forcée, deux cas apparaissent : la référence pointe vers le résultat d'une évaluation précédente, qui peut être directement renvoyé, ou vers une fonction, auquel cas elle est évaluée et le résultat est stocké dans la référence.

Cependant, les règles de réduction des co-points-fixes requièrent que nous gardions trace du terme original qui a été forcé.[5] Ceci conduit à l'implémentation suivante :

[5] *À cause du partage des calculs, il faut faire attention à ne pas changer la forme normale d'un co-point-fixe s'il a déjà été forcé par ailleurs.*

```
type atom =
  | ...
  | Acofix_e of t * (t -> t) * t
  | Acofix of t * (t -> t) * (unit -> t)

let update_atom v a =
   Obj.set_field (Obj.magic v) 3 (Obj.magic a)

let force v =
  if is_accu v then match get_atom v with
     | Acofix_e(_,_,v') -> v'
     | Acofix (t,norm, f)  ->
        let v' = app_list f (args_accu v) () in
        update_atom v (Acofix_e(t,norm v')); v'
     | _ -> v
  else v
```

Pour forcer une valeur, nous vérifions d'abord s'il s'agit d'un accumulateur. Si ce n'est pas le cas, nous avons affaire à un constructeur d'un type co-inductif, que nous renvoyons inchangé. En revanche, dans le cas d'un accumulateur, si l'atome est un co-point-fixe déjà évalué, nous renvoyons le résultat stocké. Si c'est un co-point-fixe qui n'a pas été évalué, la fonction est appliquée à ses arguments accumulés (à travers la routine `app_list`) et l'accumulateur est mis à jour avec un nouvel atome. Dans le dernier cas, l'accumulateur est un terme neutre.

Les types co-inductifs ont strictement le même schéma de compilation que les types inductifs (c.f. section 1.2). Pour les co-points-fixes, nous utilisons le schéma suivant :

$\ulcorner$**cofix** f : T := t$\urcorner^{B}$ =

```
let fnorm f = ⌜t⌝^{B∪{f}} in
let f = mk_accu dummy_atom in
update_atom f (Acofix(⌜T⌝^{B}, fnorm, fun _ -> fnorm f));
f
```

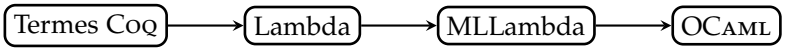
Ceci est directement adapté de la compilation des points fixes (c.f. section 1.2). Il faut noter que l'accumulateur `f` est créé au départ avec un atome factice et ensuite mis à jour avec un atome réel dont la définition dépend de `f` (sous une lambda abstraction). Nous utilisons cette construction pour contourner une limitation sur le membre droit du `let rec` en OCaml. Enfin, la compilation du filtrage ajoute un `force` si le terme filtré a un type co-inductif.

1.4 IMPLÉMENTATION

Nous décrivons maintenant l'implémentation que nous avons réalisée pour Coq des techniques présentées jusqu'ici, permettant de normaliser un terme ou de tester la convertibilité par compilation en code natif, à travers le compilateur optimisant d'OCaml. Cette implémentation est accessible dans les versions récentes de Coq par la commande `Eval native_compute in t`, la tactique `native_compute` ou encore l'insertion d'un cast explicite `t <: T`.

1.4.1 Architecture du compilateur

Notre compilateur des termes de Coq vers les programmes OCaml procède en trois passes :



Les langages intermédiaires mis en jeu diffèrent notamment par leur représentation des variables liées : les termes Coq et la représentation Lambda utilisent des indices de De Bruijn tandis que le MLLambda et le code source OCaml généré en sortie utilisent des variables nommées.

La première phase consiste grossièrement à enlever des termes toutes les informations qui ne sont pas utiles à la normalisation ou au test de conversion.

Le langage intermédiaire Lambda est un λ-calcul enrichi notamment avec des constantes et variables globales, des définitions locales, des conditionnelles et des opérations primitives. Il est inspiré du langage intermédiaire du même nom utilisé par le compilateur OCaml [X. Leroy, 1990] et, comme dans ce dernier, nous permet de traiter le dépliage (*inlining*) et la gestion des

opérateurs primitifs liés aux entiers et tableaux que nous décrirons dans la section 1.5, ainsi que la réduction de β-redex apparents.

Le langage MLLambda, quant à lui, est très proche de la syntaxe abstraite d'un fragment du langage OCaml. Ainsi, la dernière phase est essentiellement une mise en forme immédiate d'un programme source représenté en syntaxe abstraite. L'essentiel de travail de compilation, que nous avons noté $\ulcorner\cdot\urcorner$ précédemment, a donc lieu lors de la deuxième phase (de Lambda vers MLLambda).

À l'issue de la dernière phase, un programme source OCaml est généré, puis compilé. Le résultat de la normalisation ou du test de conversion est récupéré grâce au chargement dynamique, disponible en code natif depuis la version 3.11 d'OCaml. Ce procédé nous permet de ne pas réévaluer les constantes globales qui ont déjà été évaluées lors d'une normalisation ou d'un test de conversion précédent. Nous veillons enfin à respecter la modularité du code en compilant chaque librairie Coq (c'est-à-dire chaque fichier d'extension `.v`) vers un plug-in dynamique, qui sera chargé en même temps que la librairie compilée (fichier `.vo`).

Plus précisément, pour un terme `t`, un appel à `Eval native_compute in t` va compiler et lier dynamiquement le programme suivant :

```
let t1 = ⌜t⌝

let _ =
  Nativelib.rt1 := t1
```

où `Nativelib.rt1` est une référence exposée par le programme hôte (Coq) qui peut alors appeler la fonction de relecture sur la valeur contenue dans cette référence. La libération de la mémoire est permise en réinitialisant `Nativelib.rt1` à une valeur par défaut. Remarquons que la libération de la mémoire occupé par le *code* des programmes liés dynamiquement n'est, elle, pas permise par le module `Dynlink` d'OCaml. Cependant, la taille de ce code en mémoire est en pratique négligeable par rapport à la taille des objets manipulés.

1.4.2 Optimisations

Afin de rendre le schéma de compilation présenté dans la section 1.2 plus explicite, nous considérons maintenant quelques exemples concrets, avec les optimisations spécifiques que nous avons implémentées.

FILTRAGE Notre schéma de compilation du filtrage représente celui-ci par une clôture récursive, qui est stockée dans l'accumulateur construit si le filtrage est gelé. Bien que ce procédé permette d'avoir toutes les informations nécessaires à la réification, il a l'inconvénient de ne plus faire apparaître explicitement le filtrage à l'endroit où il a lieu dans le terme d'origine, et de déclencher la construction locale d'une clôture, ce qui impacte négativement les performances. Nous choisissons donc de déplier l'appel à la clôture récursive, en encapsulant celle-ci dans une définition globale auxiliaire. Considérons la fonction Coq suivante, qui teste si un entier naturel est nul :

```
Definition is_zero (m : nat) :=
  match m with
  | O => true
  | _ => false
```

```
  end.
```

Une application stricte de notre schéma de compilation donnerait lieu à :

```
let const_is_zero m =
  let rec case c = match c with
  | Accu_nat _ ->
      mk_sw_accu [..] (cast_accu m) (fun _ -> const_bool) case
  | Construct_nat_0 -> true
  | Construct_nat_1 _ -> false
  in
  case m
```

Mais dans l'implémentation réelle, nous définissons deux fonctions auxiliaires `case_is_zero` et `pred_is_zero` contenant respectivement la clôture récursive locale `case` du code ci-dessus et le prédicat de retour du filtrage. Ainsi, ces deux constructions ne seront pas réallouées à chaque fois que le filtrage est exécuté. Après optimisation, le code devient donc :

Dans cet exemple, les fonctions `case_is_zero` *et* `const_is_zero` *coïncident car notre terme de départ est réduit à un filtrage. Le prochain paragraphe montre un exemple plus évolué. Les constructeurs* `Construct_nat_0` *et* `Construct_nat_1` *représentent respectivement* `O` *(zéro) et* `S` *(successeur).*

```
let pred_is_zero = fun _ -> const_bool

let rec case_is_zero m = match c with
  | Accu_nat _ ->
      mk_sw_accu [..] (cast_accu m) pred_is_zero case_is_zero
  | Construct_nat_0 -> true
  | Construct_nat_1 _ -> false

let const_is_zero m = match c with
  | Accu_nat _ ->
      mk_sw_accu [..] (cast_accu m) pred_is_zero case_is_zero
  | Construct_nat_0 -> true
  | Construct_nat_1 _ -> false
```

POINTS FIXES Dans le cas des points fixes, nous remarquons que leur corps commence souvent par un filtrage sur l'argument récursif. Le test déterminant si cet argument commence par un constructeur ou non peut alors être fusionné avec le filtrage. Prenons comme exemple l'addition sur les entiers naturels définie comme :

```
Fixpoint add (m n : nat) :=
  match m with
  | O => n
  | S p => S (add p n)
  end.
```

Avec l'optimisation du filtrage, cette fonction est compilée vers le programme suivant :

```
let pred_add = fun _ _ -> const_nat

let rec case_add f n m = match m with
  | Accu_nat _ ->
      mk_sw_accu [...] (cast_accu m) pred_add (case_add f n)
  | Construct_nat_0 -> n
  | Construct_nat_1 p -> Construct_nat_1 (f p n)
```

```
let norm_add f m n = match m with
  | Accu_nat _ ->
      mk_sw_accu [...] (cast_accu m) pred_add (case_add f n)
  | Construct_nat_0 -> n
  | Construct_nat_1 p -> Construct_nat_1 (f p n)
in
let rec const_add m n = if is_accu m then
    mk_fix_accu [...] fixtype_add norm_add m n
  else norm_add const_add m n
```

Dans le code ci-dessus, on remarque que le test `is_accu` et la branche `Accu_nat` de la fonction `norm_add` font double emploi. On peut donc les fusionner pour obtenir un code généré final ressemblant à :

```
let pred_add = fun _ _ -> const_nat

let rec case_add f n m = match m with
  | Accu_nat _ ->
      mk_sw_accu [...] (cast_accu m) pred_add (case_add f n)
  | Construct_nat_0 -> n
  | Construct_nat_1 p -> Construct_nat_1 (f p n)

let norm_add f m n = case_add f n m

let rec const_add m n = match m with
  | Accu_nat _ -> mk_fix_accu [...] fixtype_add norm_add m n
  | Construct_nat_0 -> n
  | Construct_nat_1 p -> Construct_nat_1 (const_add p n)
```

VALEURS CLOSES ET PARTAGE DU CODE Lorsqu'une valeur close est connue statiquement, il est inefficace de la compiler vers sa représentation sous forme de constructeurs OCaml, pour deux raisons. D'une part le code produit sera moins compact et entraînera un temps de compilation plus long. De nombreuses constructions en Coq ayant des représentations gourmandes en mémoire (par exemple, les entiers de Peano), nous avons observé des cas où ce temps de compilation devient très significatif. D'autre part, si cette valeur a plusieurs occurrences au cours de tests de conversion successifs, elle sera allouée en mémoire plusieurs fois.

Pour contourner ce problème, nous générons à la compilation les représentations mémoire des valeurs closes, et les stockons dans une table. Ainsi, le code compilé remplace la valeur par un accès dans cette table.

Nous assurons de plus, à l'échelle de chaque librairie Coq, un partage des valeurs et des définitions globales. Ceci est particulièrement utile pour les valeurs, mais aussi pour les prédicats de retour des filtrages, qui sont souvent redondants.

1.5 ENTIERS MACHINES ET TABLEAUX

Un des avantages de l'approche présentée dans ce chapitre est que notre compilateur dispose de l'ensemble du langage OCaml comme cible. En particulier, des extensions de la machine virtuelle de Coq comme des entiers

machine [A. Spiwack, 2011] ou des tableaux persistants [M. Armand, B. Grégoire et al., 2010] peuvent être portées à notre contexte à moindre frais.

Cependant, nous choisissons une approche légèrement différente de celle des deux travaux que nous avons cités. En effet, ceux-ci reposent sur un mécanisme appelé « retroknowledge » qui consiste à définir un type inductif reflétant un type de données exploitable à plus bas niveau, par exemple des mots de 31 bits pour accéder à l'arithmétique processeur. Tous les opérateurs sont définis et leur propriétés prouvées sur ce type inductif, mais au moment d'évaluer un terme dans ce type inductif, si sa valeur est connue, sa représentation est substituée par son équivalent en machine. De même, les opérateurs appliqués à des entiers se réduisant vers des valeurs closes sont substitués par les opérateurs correspondant de l'arithmétique processeur (plus précisément, leur implémentation utilisée par l'environnement d'exécution d'OCaml).

L'avantage du *retroknowledge* est qu'il est transparent pour l'utilisateur, et ne demande pas de modification du formalisme en lui-même. De plus, aucun axiome n'est rajouté explicitement, la seule hypothèse méta-théorique étant que la substitution des valeurs closes et des opérateurs par leurs équivalents en machine est correcte. Tous les termes du formalisme gardent donc un comportement calculatoire.

Mais le prix à payer est que les représentations compactes des valeurs ne sont utilisées que localement, lors de l'évaluation ou du test de conversion. Dans tout le reste du système, un encodage dans le type inductif associé, gourmand en mémoire, est utilisé. Dans des cas extrêmes, par exemple la vérification de traces de preuves issues de solveurs SMT (*satisfaisabilité modulo théories*) [M. Armand, G. Faure et al., 2011 ; F. Besson et al., 2011 ; P.-E. Cornilleau, 2013 ; C. Keller, 2013], il peut même être impossible d'allouer de la mémoire pour le terme contenant l'encodage inductif des données.

Partant de ce constat, Benjamin Grégoire a initié une approche alternative [M. Armand, B. Grégoire et al., 2010], que nous suivons ici. Celle-ci consiste à étendre directement le formalisme avec des objets représentés efficacement en machine. Les opérateurs sur ces objets sont des primitives dont la théorie équationnelle est axiomatisée. Ainsi l'occupation mémoire est réduite dans l'ensemble du système, que ce soit à la création du terme, lors de la vérification du typage ou pour stocker les termes de preuve.

ENTIERS MACHINE Un des points délicats de l'implémentation d'arithmétique machine est la question de la portabilité. En OCaml par exemple, la taille des entiers dépend de l'architecture : 31 bits sur une architecture 32 bits, 63 sur une architecture 64 bits. Mais dans le contexte d'un système de preuve, où la reproductibilité est une préoccupation centrale, il est raisonnable de vouloir s'abstraire d'une architecture particulière et d'avoir un comportement uniforme dans tous les cas. Dans la version 8.4 de Coq, le choix est fait de donner accès à une arithmétique sur 31 bits. Si l'architecture sous-jacente s'avère être 32 bits, c'est donc l'arithmétique native, telle qu'implémentée par OCaml, qui est utilisée. Dans le cas d'une architecture 64 bits en revanche, des opérations de masquage sont utilisées pour simuler cette arithmétique modulo 2^{31}.

Nous pensons qu'il est dommage de ne pas profiter des 63 bits lorsqu'ils sont disponibles, d'autant que les machines et systèmes 64 bits sont de plus en plus courants. Nous implémentons donc un module `Uint63` d'arithmétique non signée modulo 2^{63} portable, qui sur les architectures 32 bits est émulée grâce à la bibliothèque `Int64` fournie dans la distribution OCaml.

Les performances sur ces architectures seront donc dégradées, mais en retour des résultats précédemment inaccessibles pourront être obtenus sur des machines 64 bits. Le seul inconvénient est que la portabilité de la représentation interne des termes de preuves (que l'on retrouve dans les fichiers d'extension `.vo`) n'est pas assurée. Les garanties de portabilité sont néanmoins préservées au niveau source (fichiers d'extension `.v`).

Le schéma de compilation des valeurs littérales entières dépend donc de l'architecture. Pour en tenir compte, nous définissons un type abstrait `Uint63.t` qui sera instancié à `int` (le type primitif des entiers OCaml) dans le cas 64 bits et au type `Int64.t` de la librairie standard OCaml dans le cas 32 bits. Dans le premier cas, on peut donc injecter le type `int` dans `Uint63.t` à l'aide d'une fonction `Uint63.of_int` qui est l'identité :

$$\ulcorner \texttt{i} \urcorner^{B} = \texttt{Uint63.of\_int i}$$

Si par contre notre compilateur est utilisé sur une architecture 32 bits, c'est le type `Int64.t` que nous injectons dans `Uint63.t` :

$$\ulcorner \texttt{i} \urcorner^{B} = \texttt{Uint63.of\_int64 iL}$$

Le suffixe `L` indique à OCaml d'interpréter une valeur littérale dans `Int64.t`.

Pour traduire les opérations arithmétiques, nous introduisons une fonction `is_int : Values.t -> bool` qui renvoie **`true`** si la valeur passée en argument est close et entière (du type `int` ou `Int64.t`. Chaque opération est implémentée par deux fonctions, l'une testant les arguments à l'aide de `is_int` et renvoyant un accumulateur si l'opération est gelée, l'autre effectuant l'opération proprement dite. Pour l'addition par exemple, nous définissons :

```
let no_check_add x y =
  mk_uint (Uint63.add (to_uint x) (to_uint y))

let add accu x y =
  if is_int x && is_int y then no_check_add x y
  else accu x y
```

Les fonctions `mk_uint` et `to_uint` sont des fonctions identités de conversion entre les types `Uint63.t` et `Values.t` implémentées par `Obj.magic`.

L'argument `accu` de la fonction `add` est destiné à représenter l'application de l'addition à des expressions entières non closes. Le schéma de compilation de l'addition est alors simplement :

$$\ulcorner t_1 + t_2 \urcorner^{B} = \texttt{add const\_add} \ \ulcorner t_1 \urcorner^{B} \ulcorner t_2 \urcorner^{B}$$

La constante `const_add` définit un accumulateur représentant l'opération primitive d'addition sur les entiers.

Dans l'implémentation réelle, lors de la compilation des expressions arithmétiques nous traitons séparément le cas où toutes les variables mises en jeu s'évaluent vers des valeurs closes, auquel cas toutes les opérations sont remplacées par leur équivalent traitant uniquement le cas clos (`no_check_add` dans le cas de l'addition). De plus, si une variable a plusieurs occurrences dans une expression, le test `is_int` n'est effectué qu'une seule fois. Enfin, ce test n'est jamais appliqué à des valeurs littérales connues à la compilation. La philosophie derrière ces optimisations est de minimiser, dans le cas de termes clos, le coût des tests liés à la prise en charge des termes ouverts.

TABLEAUX PERSISTANTS Nous fournissons également des primitives pour manipuler une autre structure de données : les tableaux persistants [H. G. Baker, 1978, 1991]. Ces tableaux offrent une interface fonctionnelle (par exemple, la fonction d'ajout d'un élément prend en argument un tableau

et en renvoie un nouveau), tout en utilisant en interne une structure impérative. La cohérence de l'interface fonctionnelle est assurée en maintenant l'historique des modifications apportées au tableau. L'implémentation est telle que si les accès se font toujours à la version la plus récente du tableau, le comportement est le même qu'en utilisant directement une structure impérative.

L'intérêt dans notre contexte est que l'interface fonctionnelle est compatible avec la logique de Coq, tout en apportant l'efficacité des tableaux destructifs, c'est-à-dire notamment un accès à un élément en temps constant, dans le cas où on accède toujours à la dernière version du tableaux. Ainsi, un prototype [M. Armand, B. Grégoire et al., 2010] implémente cette structure de données dans la machine virtuelle de Coq. Ici encore, le fait de disposer de tout le langage OCaml comme cible de compilation nous permet de reprendre directement l'implémentation décrite dans [S. Conchon et J.-C. Filliâtre, 2007].[6]

[6] *Notre implémentation diffère légèrement : nous n'appelons pas systématiquement l'opération* `reroot` *lors d'un accès à un tableau ancien ou d'une mise à jour, mais l'exposons à l'utilisateur.*

Nous étendons donc la syntaxe des termes du C.C.I. avec la construction $[|x_1;\ldots;x_n\,|\,a\,|]$ qui désigne un tableau dont les éléments sont $x_1,\ldots,x_n$ et dont l'élément par défaut (renvoyé en cas d'accès en dehors des bornes) est a. Le schéma de compilation des tableaux construits explicitement est le suivant :

$$\ulcorner[|x_1;\ldots;x_n\,|\,a\,|]\urcorner^B \quad = \quad \texttt{mk\_parray (ref (Array [|}\ulcorner x_1\urcorner^B\texttt{;...;}\ulcorner x_n\urcorner^B\texttt{;}\ulcorner a\urcorner^B\texttt{|]))}$$

Les opérateurs sur ces tableaux persistants (notamment `get`, `set`, `make`, `init`, `map` et `length`) sont compilés de manière similaire aux opérations sur les entiers. Ainsi, nous introduisons une fonction `is_parray` de type `Values.t -> bool` qui renvoie **`true`** si la valeur en argument est un tableau persistant. Pour détecter ce cas, nous utilisons le fait qu'un tableau persistant est représenté par une référence et qu'aucun autre type de valeur de notre calcul ne peut se réduire vers une référence. Pour l'opérateur `get` d'accès à un élément, nous définissons les deux fonctions suivantes :

La fonction `Parray.get` *vérifie si l'indice est dans les bornes, et renvoie l'élément par défaut associé au tableau dans le cas contraire.*

```
let no_check_arrayget t n =
   Parray.get (to_parray t) (to_uint n)

let arrayget accu vA t n =
  if is_parray t && is_int n then no_check_arrayget t n
  else accu vA t n
```

Ici, `t` *désigne un terme de type* `array` *T, c'est-à-dire représentant un tableau d'éléments de type* `T`.

Le schéma de compilation de cet opérateur est alors :

$$\ulcorner t.[i]\urcorner^B = \texttt{arrayget const\_arrayget }\ulcorner T\urcorner^B\ulcorner t\urcorner^B\ulcorner i\urcorner^B$$

Les autres opérations primitives sur les tableaux persistants sont compilées de la même façon.

1.6 PERFORMANCES

Pour évaluer les performances de notre approche sans étiquette, nous avons comparé notre implémentation (**Native**) à la conversion utilisant une machine virtuelle dédiée (**Bytecode**) [B. Grégoire et X. Leroy, 2002], ainsi qu'à du code extrait (**Extracted**) grâce au mécanisme d'extraction de Coq puis compilé au moyen du compilateur optimisant d'OCaml. Cette dernière comparaison permet de mesurer le surcoût actuel de la prise en charge des

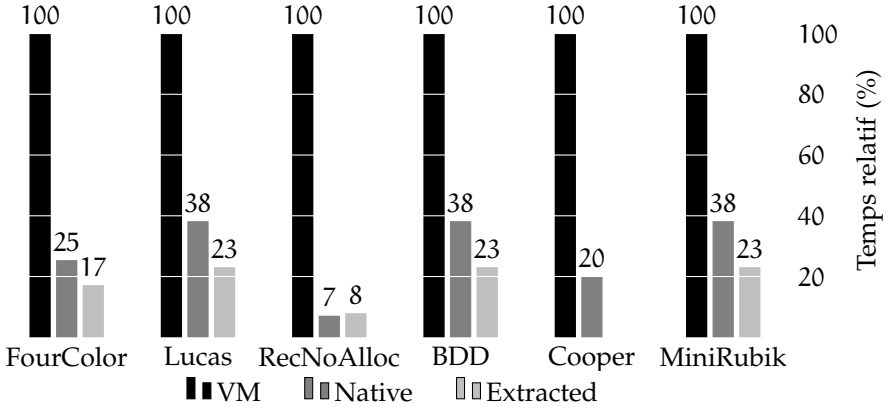


FIGURE 1.3 – Temps relatifs d'exécution de la machine virtuelle de Coq, de la normalisation par compilation native et du code extrait compilé vers du code natif, sur des exemples variés.

termes ouverts (l'approche par extraction et compilation ne fonctionnant que sur des termes clos).

Afin d'assurer une comparaison significative, nous avons extrait la plupart de nos tests de cas d'utilisations réels, issus de contextes variés :
BDD est une implémentation des diagrammes de décision binaire [K. N. VERMA et al., 2000], qui vérifie si une proposition donnée est une tautologie. Dans notre exemple, nous l'avons utilisée sur une expression du principe des tiroirs : si n chaussettes sont rangées dans $n-1$ tiroirs, il ne peut y avoir une seule chaussette par tiroir.
Four colour est le test de réductibilité des configurations issu de la preuve formelle du théorème des quatre couleurs [G. GONTHIER, 2007], ce qui représente la part la plus importante du temps de calcul de l'ensemble de la preuve.
Lucas est une implémentation du critère de primalité de Lucas-Lehmer qui décide si un nombre de Mersenne donné est premier ou non.
MiniRubik vérifie que toute position d'une variante de taille $2 \times 2 \times 2$ du Rubik's cube est résoluble en au plus 11 coups, en utilisant les entiers machines et les tableaux. La formalisation d'origine est décrite dans [L. THÉRY, 2008].
Cooper implémente l'élimination des quantificateurs de Cooper sur une formule à 5 variables.
RecNoAlloc déclenche 2^{29} appels récursifs triviaux (c'est-à-dire sans allocation mémoire pour stocker le résultat). Ce test vise à mesurer l'impact de performance pure quand le glaneur de cellule n'est pas sollicité significativement.

	Temps (s)			
	FourColor	RecNoAlloc	Lucas	Cooper
vm	4 667,23	56,39	22,88	44,21
native	1 185,33	4,01	8,75	8,83
extracted	801,61	4,47	5,28	NaN

TABLE 1.1 – Évaluation des performances du compilateur natif

La plupart des résultats mettent en évidence un gain en temps d'un facteur 2 à 5 par rapport à la machine virtuelle de Coq, ce qui est proche des rapports que l'on retrouve typiquement entre l'interprétation de bytecode et la compilation en code natif. Il est à noter que l'amélioration des performances est particulièrement significative sur les exemples sollicitant moins le glaneur de cellules. Ce constat est mis en évidence par **RecNoAlloc**, où le facteur de gain se situe autour de 7. Nous n'avons pas pu évaluer les performances du code extrait pour **Cooper**, car l'extraction de Coq échoue sur cet exemple.

Nous avons également utilisé **Cooper** et **RecNoAlloc** pour évaluer l'écart entre notre implémentation actuelle et un prototype préliminaire basé sur la version de normalisation avec étiquettes, donnant un facteur de gain en temps entre 1.5 et 2.5 en faveur de la première.

L'écart de performance restant entre le code extrait et la normalisation par compilation native peut être attribué au surcoût du test de l'argument structurel pour la condition de garde à chaque appel récursif.

1.7 CONCLUSION

L'amélioration de l'automatisation et la réduction de la taille des preuves formelles est une tendance forte, qui permet de réduire l'effort de l'utilisateur, et ainsi de mener des développements formels de taille auparavant inenvisageable. Le revers de la médaille est que l'implémentation des composants vérifiant la preuve est soumise à une plus forte pression. La preuve du théorème des quatre couleurs en est un exemple édifiant, avec un temps de calcul qui est de l'ordre d'une heure et demie sur une machine performante, en utilisant la compilation en bytecode et qui passe sous les vingt minutes en utilisant notre compilateur. Nous espérons que l'amélioration des performances permise par le travail présenté dans ce chapitre permettra de repousser encore plus loin les limites des développements réalisés en mathématiques formelles et en vérification de programmes.

Notre approche est suffisamment légère pour être implémentée dans la plupart des environnements de preuve interactive où le calcul dans les preuves joue un rôle important, sans nécessiter d'expertise particulière sur les technologies de compilation.

La correction de notre approche repose en partie sur la correction du compilateur, qui rentre entièrement dans la base de confiance. Cependant, si le langage cible de la traduction des termes que nous avons présentée coïncide avec le langage d'implémentation de l'assistant à la preuve, alors le compilateur fait déjà partie de la base de confiance, même si c'est de manière moins critique. Dans ce contexte, la certification d'un compilateur pour le langage cible serait tout à fait bienvenue.

Puisque le langage cible est accessible dans son ensemble à notre routine de compilation, nous avons pu implémenter à moindre frais de nouvelles fonctionnalités comme les entiers machines ou les tableaux persistants, alors que les approches basées sur des compilateurs et environnements d'exécution ad-hoc demandent un travail supplémentaire à chaque fois qu'une telle extension est réalisée.

Notre implémentation avec étiquettes est portable vers n'importe quel langage fonctionnel, tandis que la version sans étiquettes utilise des extensions spécifiques d'OCaml. Cependant, des fonctionnalités similaires sont déjà partiellement implémentées dans d'autres langages. En particulier, la primi-

tive `unpackClosure#` du compilateur GHC pour Haskell pourrait suffire à nos besoins. Une étude plus poussée devra être menée pour déterminer la faisabilité de cette approche pour d'autres langages.

Pour résumer, nos contributions présentées dans ce chapitre sont :

- Un cadre unifiant la normalisation par évaluation non typée et les approches compilées de la réduction forte des termes d'un langage fonctionnel.
- Une implémentation en OCaml de l'interface définie par ce cadre réutilisant au maximum les fonctionnalités de l'évaluateur ambiant, garantissant ainsi de bonnes performances.
- L'intégration dans la version de développement de Coq de cette implémentation.

Précisons que ce travail a été réalisé à partir d'un prototype développé par Mathieu Boespflug implémentant, grâce à la normalisation non typée, la réduction forte du λ-calcul pur avec constantes, étendu par des types algébriques et du filtrage [M. Boespflug, 2010]. Ce prototype était basée sur l'approche « avec étiquettes ». Nous l'avons dans un premier temps étendu à l'ensemble du Calcul des Constructions Inductives. Puis, en collaboration avec Benjamin Grégoire, nous l'avons porté à l'approche « sans étiquettes ». En outre, ce travail à fait l'objet d'une publication [M. Boespflug et al., 2011].

Enfin, signalons qu'une part significative de notre effort de développement a consisté à passer d'un prototype à une implémentation finale intégrée à la version de développement de Coq, et compatible avec l'ensemble des fonctionnalités du système.

Cet effort a permis de mettre à disposition notre développement et de constater qu'il répond à un réel besoin, puisqu'il a été utilisé dans divers contextes. Citons notamment, outre les exemples que nous développerons nous-mêmes dans cette thèse, une implémentation efficace (et formellement vérifiée) des nombres réels exacts [R. Krebbers et B. Spitters, 2011], l'importation de preuves générées par des prouveurs SMT [M. Armand, G. Faure et al., 2011 ; F. Besson et al., 2011 ; P.-E. Cornilleau, 2013 ; C. Keller, 2013] et des approximations polynomiales certifiées [N. Brisebarre et al., 2012 ; E. Martin-Dorel, 2012].

Nous envisageons deux lignes directrices principales pour prolonger et améliorer ce travail. La première, nous l'avons déjà évoquée, consiste à réduire la base de confiance en vérifiant formellement le compilateur que nous utilisons et l'environnement d'exécution associé, notamment. Un travail qui pourrait être réutilisé dans ce contexte est [Z. Dargaye, 2009], où est présenté un compilateur optimisant (vérifié formellement) pour un mini langage fonctionnel. Cependant, pour pouvoir servir de cible à notre traduction des termes Coq, ce langage devra être étendu.

Une seconde perspective d'amélioration est l'utilisation de compilation « juste à temps » (JIT). En effet, un des inconvénients de notre approche actuelle est que le temps de compilation est souvent significatif car nous faisons appel au compilateur optimisant OCaml dont l'allocation de registres notamment est assez coûteuse (mais produit en retour un code affichant de bonnes performances à l'exécution). En pratique, cela signifie que l'utilisation de notre compilateur n'est intéressante que lorsque les calculs mis en jeu sont suffisamment importants. Des outils de compilation plus flexibles permettraient d'adapter le temps passé à générer le code au temps de calcul

estimé. Par exemple, disposer de plusieurs algorithmes d'allocation de registres et d'une heuristique de choix permettant d'optimiser dans la plupart des cas le temps global passé à compiler et évaluer. Une autre possibilité serait de lancer la machine d'évaluation standard de Coq en parallèle avec la compilation, quitte à l'interrompre si la compilation termine la première.

Dans tous les cas, la mise au point d'un mécanisme d'évaluation efficace pour les calculs coûteux et se comportant raisonnablement bien sur des calculs plus petits est un objectif intéressant car il permettrait d'enlever la machine virtuelle et le compilateur en bytecode présents dans le noyau de Coq, réduisant ainsi la base de confiance.

2 RAFFINEMENTS

Les algorithmes que nous allons étudier dans cette thèse ont la particularité d'avoir des spécifications faisant intervenir des objets mathématiques relativement évolués. Comparativement à un algorithme de tri par exemple, les calculs de groupes d'homologie décrits en partie III nécessitent un développement théorique important. Nous attachons donc une grande importance à la réutilisation de librairies mathématiques existantes, lorsque les théories qui y sont développées permettent d'établir nos résultats de correction.

En particulier, nous utiliserons la librairie SSReflect, qui préconise l'utilisation de la réflection à petite échelle pour atteindre un niveau de détail comparable à celui des mathématiques usuelles sur papier, même pour des théories avancées comme la preuve du théorème de Feit-Thompson. Dans cette approche, l'utilisateur se concentre sur les étapes significatives du raisonnement tandis que les détails de bas niveau sont traités par des petites étapes calculatoires, du moins lorsque les propriétés sont décidables. Une telle méthode rend le style de preuve plus proche des mathématiques usuelles.

Une des particularités de ces librairies est qu'elles reposent sur des types dépendants riches, ce qui donne l'opportunité d'encoder une grande quantité d'informations directement dans le type des objets : par exemple, le type des matrices contient leur taille, ce qui rend les opérations comme la multiplication plus faciles à spécifier. De la même façon, les algorithmes sur ces objets sont assez simples pour que leur correction découle facilement de leur définition.

Cependant, en pratique ces descriptions adaptées aux preuves sont éloignées des implémentations efficaces que l'on trouve dans les logiciels de calcul formel qui, de leur côté, ne reposent pas sur des types dépendants et ne fournissent pas de preuves de correction sur machine. La performance étant une problématique importante dans la suite de notre travail, nous développons dans ce chapitre une méthodologie pour réconcilier ces deux aspects.

Il est bien connu que les programmes et structures de données bien adaptés au calcul sont habituellement plus difficiles à étudier formellement que lorsqu'ils sont naïfs. Les formalismes riches comme le Calcul des Constructions Inductives offrent la possibilité d'exprimer plusieurs représentations d'un même objet mathématique, de façon à laisser l'utilisateur choisir celle qui est la mieux adaptée à ses besoins.

Même des objets aussi simples que les entiers naturels peuvent avoir plusieurs représentations. Ils peuvent être unaires (entiers de Peano), dotés alors d'un schéma inductif simple, ou binaires, ce qui les rend exponentiellement plus compacts mais entraîne habituellement des preuves plus complexes. Leurs incarnations respectives dans la librairie standard de Coq sont les deux types inductifs `nat` et `N`.

Le problème est alors d'être capable d'abstraire un algorithme exprimé sur l'un ou l'autre de ces deux types, s'il est suffisamment générique. Ainsi, sa preuve de correction peut être factorisée, aboutissant à un développement plus concis et plus facile à maintenir. Par exemple, le système de modules de Coq [J. Chrzaszcz, 2003 ; E. Soubiran, 2010] constitue un mécanisme

d'abstraction par axiomatisation : une interface contient la signature des opérateurs et les axiomes qu'ils doivent vérifier. Dans le cas des entiers naturels, cette interface est instanciée notamment par deux modules définissant les opérateurs pour `nat` d'une part et `N` d'autre part, et prouvant leurs propriétés.

Cette approche a au moins deux inconvénients dans notre contexte : tout d'abord, il n'est pas toujours facile de définir une interface adaptée, et il n'est même pas certain qu'il en existe toujours une. Ensuite, cette méthode d'abstraction va à l'encontre de la vision, commune en théorie des types, des objets dotés d'un comportement calculatoire. En effet, les axiomes décrits dans les interfaces sont des objets inertes, ce qui interdit en pratique l'utilisation de techniques comme la réflection à petite échelle que nous avons déjà mentionnée.

Pour résoudre ce problème, il suffit de considérer une implémentation de référence des opérateurs, sur des structures adaptées aux preuves, puis de spécifier la correction des versions plus efficaces par rapport à cette référence. Dans le cadre des entiers naturels, on pourrait utiliser les isomorphismes `N.of_nat : nat -> N` et `N.to_nat : N -> nat` pour exprimer la correction de l'addition `N.add` sur les entiers binaires par rapport à celle sur les entiers unaires (notée `+`) :

```
Lemma addP (m n : nat) :
  N.add (N.of_nat m) (N.of_nat n) = N.of_nat (m + n)
```

L'intérêt est que cette manière de spécifier la correction est *compositionnelle* : si nous définissons un algorithme qui utilise génériquement les opérateurs de base sur les entiers, nous pouvons prouver sa correction sur les entiers unaires dans un premier temps, puis transporter cette preuve aux entiers binaires en composant les lemmes de correction des opérateurs de base.

Cette idée est souvent utilisée de manière ad-hoc. L'objectif de ce chapitre est d'en faire une présentation systématique à travers la notion de raffinement, qui est couramment utilisée pour décrire les étapes successives de la vérification d'un programme. Les raffinements sont une idée ancienne dans l'étude de la correction des programmes [C. A. R. Hoare, 1972]. De nombreux cadres théoriques l'étudient, comme le calcul des raffinements [R.-J. Back, 1978 ; R.-J. Back et J. von Wright, 1999], et des environnements de développements de programmes certifiés en font également usage, comme le projet FoCaLiZe [S. Boulmé, 2000] ou la méthode B [M.-L. Potet et Y. Rouzaud, 1998 ; J.-R. Abrial, 2005].

Typiquement, en pratique une spécification est exprimée dans une logique de Hoare, puis le programme est décrit dans un langage de haut niveau, et enfin implémenté en C. Chaque étape est prouvée correcte par rapport à la précédente. L'utilisation de plusieurs formalismes demande de faire confiance à chaque étape de traduction ou de les prouver correctes dans un formalisme supplémentaire.

Notre approche est similaire. Nous raffinons la définition d'un concept vers un algorithme efficace décrit sur des structures de données de haut niveau : c'est un *raffinement de programme*. La preuve de correction peut alors être complexe, mais elle est effectuée sur des objets adaptés. Ensuite, nous implémentons l'algorithme sur des structures de données plus proches des représentations machine, une fois que la théorie riche n'est plus nécessaire pour prouver la correction : c'est un *raffinement de données*. Ainsi, l'implémentation finale est une traduction immédiate de l'algorithme.

Cependant, dans notre contexte, toutes ces couches peuvent êtres exprimées dans le même formalisme (le Calcul des Constructions Inductives), bien qu'elles n'en utilisent pas exactement les mêmes fonctionnalités. D'un côté, les couches de haut niveau utilisent des types dépendants riches qui sont très utiles pour décrire les théories car ils permettent la surcharge de notations et des énoncés concis, ce qui devient vite nécessaire lorsque l'on travaille sur des mathématiques avancées. De l'autre côté, les implémentations efficaces utilisent des types simples, qui sont plus proches des implémentations standard dans les langages de programmation traditionnels. L'avantage principal de cette approche est que la correction des traductions peut facilement être exprimée dans le formalisme lui-même, et nous ne reposons sur aucune preuve externe supplémentaire.

De multiples exemples de raffinements de programmes seront décrits dans la partie II de cette thèse. Dans la suite de ce chapitre, nous montrons un traitement systématique et automatisé des raffinements de données. Plus particulièrement, nous présentons les différents types de tels raffinements (section 2.1) puis nous expliquons comment programmer génériquement des algorithmes dans ce contexte (section 2.2) et automatiser le transport des preuves de corrections aux différentes instances de ces programmes génériques (section 2.3). Enfin, la section 2.4 présentera une vue d'ensemble des travaux connexes au nôtre.

2.1 RAFFINEMENTS DE DONNÉES

Dans cette section, nous allons étudier différents exemples de raffinements de données. Tous ceux-ci rentreront dans un cadre général de raffinements basé sur des relations hétérogènes reliant des types orientés preuve à des types orientés calcul. Le cas le plus basique que nous considérons est lorsque ces deux types sont isomorphes.

2.1.1 Types isomorphes

Nous avons déjà évoqué en introduction les types `nat` et `N` représentant respectivement les entiers naturels unaires et binaires. Ceux-ci sont isomorphes, comme en témoignent les deux fonctions fournies par la bibliothèque standard de Coq : `N.of_nat : nat -> N` et `N.to_nat : N -> nat`. Le type `nat` est ici orienté preuve, car il dispose d'un schéma d'induction très simple, mais le type `N` permet, lui, d'avoir une taille de représentation logarithmique par rapport au nombre représenté. La relation d'isomorphisme qui les lie est représentée par la figure 2.1.

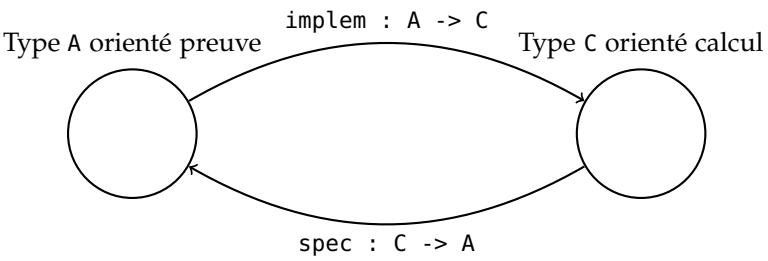


FIGURE 2.1 – Types isomorphes

À tout objet orienté preuve (de type `A`), la fonction `implem` associe élément du type `C` (que nous appelons son implémentation). Réciproquement, la fonction `spec` associe à tout objet orienté calcul (de type `C`) une spécification, c'est-à-dire un élément de type `A`. Ces deux fonctions sont inverses l'une de l'autre.

Un autre exemple qui rentre dans ce cadre est la représentation binaire `Z` des entiers relatifs dans la bibliothèque standard de Coq qui peut être déclarée comme un raffinement de la représentation unaire `int` de ces mêmes entiers dans SSReflect, qui est elle-même basée sur `nat`. On peut aussi mentionner le type `pos := {n : nat | 0 < nat}` défini dans SSReflect, qui peut être vu comme une version orientée preuve du type `positive` des entiers binaires strictement positifs de la librairie standard de Coq.

Cependant, de nombreux exemples de raffinements ne rentrent pas dans ce cadre. En particulier, il est souvent souhaitable d'autoriser plusieurs représentations orientées calcul d'un même objet. On construit alors une structure de quotient.

2.1.2 Quotients

Un exemple important nécessitant plusieurs représentations orientées calcul d'un même objet orienté preuve est le type des polynômes. Ceux-ci sont représentés dans SSReflect par un type enregistrement contenant une liste de coefficients et une preuve que le dernier élément de cette liste est non-nul :

La fonction `last` appliquée à un élément `a : T` et une séquence `s : seq T` renvoie le dernier élément de `s` ou `a` si `s` est vide

```
Record polynomial (R : ringType) :=
  Polynomial {polyseq :> seq R; _ : last 1 polyseq != 0}.
```

Cependant, cette composante de preuve n'est intéressante que pour développer la théorie des polynômes mais n'apporte rien lors du calcul. L'extraction de code l'effacerait, car il s'agit d'un énoncé vivant dans la sorte `Prop`, mais nous souhaitons également effectuer les calculs de manière interne, c'est-à-dire via la règle de conversion, auquel cas rien n'est effacé. Nous définissons donc un type orienté calcul qui représente simplement un polynôme par sa liste de coefficients. Mais alors, plusieurs éléments de ce type peuvent représenter le même polynôme car il n'y a plus de garantie sur le nombre de zéros à la fin de la liste. Ceci est résolu par la définition d'une fonction de normalisation qui enlève les zéros superflus. Cette fonction n'est pas utilisée lors du calcul, mais les lemmes de correction des opérations peuvent alors s'énoncer par égalité modulo normalisation.

Dans ce contexte, le type orienté preuve peut être vu comme un quotient du type orienté calcul par la relation d'équivalence induite par la fonction de normalisation. Ainsi, deux objets orientés calcul sont en relation s'ils ont la même forme normalisée. Ce type de relations est décrit graphiquement par la figure 2.2.

Il faut remarquer que la fonction d'implémentation n'atteint ici qu'une partie du type `C` tandis que la fonction de spécification est totale et surjective. La fonction `spec` est inverse à gauche de `implem`, ce qui s'écrit formellement : `forall x, spec (implem x) = x`. Ceci implique en particulier que `implem` est injective. En fait, cette façon d'exprimer un quotient, à l'aide d'une section (ici `implem`) et d'une rétraction (ici `spec`) est fréquemment utilisée de manière générale pour représenter les quotients en théorie des types [C. Cohen, 2013].

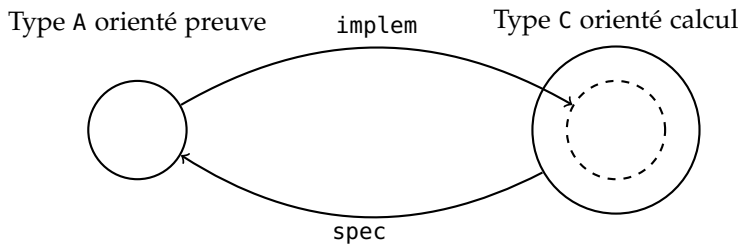


FIGURE 2.2 – Types quotient

2.1.3 Quotients partiels

Les relations de raffinements basées sur des quotients couvrent une plus large variété de types que celles définies par isomorphisme, mais il existe des exemples intéressants qui échappent également à ce cadre. Notamment, la fonction de spécification peut être partielle. Pour l'illustrer, considérons les nombres rationnels, définis dans SSReflect comme des couples d'entiers premiers entre eux, avec un dénominateur strictement positif :

```
Record rat : Set := Rat {
  valq : int * int;
  _ : (0 < valq.2) && coprime `|valq.1| `|valq.2|
}.
```

La comparaison `<` *et le prédicat* `coprime` *renvoient des booléens. Le symbole* `&&` *représente la conjonction booléenne et la notation* `` `|valq.1| `` *désigne la valeur absolue de la première projection du couple d'entiers* `valq`.

La définition ci-dessus est bien adaptée aux preuves, notamment parce que les éléments de type `rat` peuvent être comparés au moyen de l'égalité de Leibniz puisqu'ils sont normalisés. Mais maintenir cet invariant pendant des calculs concrets serait trop coûteux, car cela nécessite des calculs de pgcd. D'autre part, la structure contient aussi une preuve qui n'est utile que pour développer la théorie des nombres rationnels, mais ne présente encore une fois pas d'intérêt pour les calculs.

Afin d'être capable de calculer efficacement, nous aimerions raffiner ces nombres rationnels vers des paires d'entiers (`int * int`) qui ne soient pas nécessairement normalisées et effectuer toutes les opérations sur le sous-ensemble des paires dont la seconde composante est non nulle. Le lien entre les deux représentations est décrit par la figure 2.3.

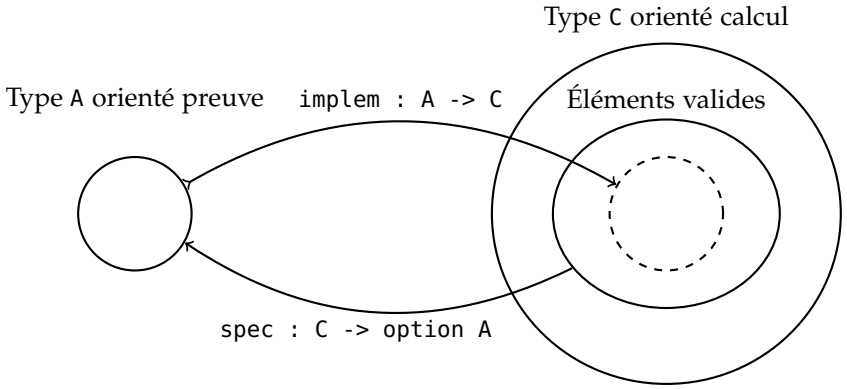


FIGURE 2.3 – Quotient partiel

La fonction de spécification a été rendue totale en ayant recours à un type option. Cette fonction `spec` est ici encore inverse à gauche de `implem`, c'est-à-dire formellement : `forall x, spec (implem x) = Some x`. Ainsi, le type

A peut être vu comme un quotient du sous-ensemble des éléments orientés calcul valides, c'est-à-dire les éléments de C qui ne sont pas envoyés sur None par spec. Dans le cas des nombres rationnels, les fonctions implem et spec et leur lemme de correction s'expriment comme suit :

Si x *est de type* int, x%:Q *permet de l'injecter dans le type* rat.

```
Definition rat_to_Qint (r : rat) : int * int := (numq r, denq r).
Definition Qint_to_rat (r : int * int) : option rat :=
  if r.2 != 0 then Some (r.1%:Q / r.2%:Q) else None.

Lemma Qrat_to_intK :
  forall (x : rat), Qint_to_rat (rat_to_Qint x) = Some x.
```

Un autre exemple de raffinement de donnés entrant dans ce cadre est celui des matrices représentées par des listes de listes. Dans ce cas, les éléments valides sont les listes contenant des listes ayant toutes la même taille. Cet exemple est étudié en détail au chapitre 3.

2.1.4 Relations de raffinement

Afin de définir un cadre unifiant tous les raffinements de données évoqués plus haut, nous nous inspirons de la réécriture généralisée [M. Sozeau, 2010]. En particulier, nous définissons une généralisation hétérogène de la classe Proper, que nous appelons param :

```
Class param (R : A -> B -> Prop) (m : A) (n : B) :=
  param_rel : R m n.
```

La classe param porte ainsi une relation entre deux types A et B, deux éléments et une preuve qu'ils sont en relation. Le fait que param soit définie comme une classe de type Coq [M. Sozeau et N. Oury, 2008] nous permettra, lorsque c'est intéressant, de rechercher automatiquement par résolution d'instance une relation R et deux éléments en relation. En particulier, nous pourrons ainsi retrouver des instances de raffinements. Dans les exemples que nous avons vus précédemment dans ce chapitre, les raffinements étaient exprimés au moyen d'une fonction de spécification spec : C -> A et non d'une relation R : A -> C -> Prop. Cependant, il est bien entendu facile de passer de la première formulation à la seconde au moyen des définitions suivantes :

```
Definition fun_hrel A B (f : B -> A) : A -> B -> Prop :=
  fun a b => f b = a.

Definition ofun_hrel A B (f : B -> option A) : A -> B -> Prop :=
  fun a b => f b = Some a.
```

Ainsi, la relation de raffinement entre rat et int peut s'écrire :

```
Definition Rrat : rat -> int * int -> Prop :=
  ofun_hrel Qint_to_rat.
```

Cette définition permet de relier des éléments de types de base. Pour exprimer des relations entre fonctions, étant données des relations entre leurs types de départ et d'arrivée, nous posons que deux fonctions sont reliées si elles envoient des éléments en relation vers des images en relation. Formellement, pour deux relations R : A -> B -> Prop et S : C -> D -> Prop, on construit une relation R ==> S : (A -> C) -> (B -> D) -> Prop sur l'espace des fonctions, de la façon suivante :

```
Definition hrespectful {A B C D : Type} (R : A -> B -> Prop) (S ↙
    ↳ : C -> D -> Prop) f g :=
  forall x y, R x y -> S (f x) (g y).

Notation " R ==> S " := (hrespectful R S).
```

Nous pouvons maintenant utiliser cette définition pour énoncer la correction de l'addition sur les nombres rationnels :

```
Instance param_addq : param (Rrat ==> Rrat ==> Rrat) +%R +%C.
```

Ici `+%C` *désigne l'addition sur* `int * int` *et* `+%R` *est l'addition sur* `rat`.

Cependant, nous avons laissé un problème de côté : nous avons raffiné le type `rat` vers `int * int`, mais le type `int` étant lui-même orienté preuve, nous n'avons pas obtenu une implémentation adaptée au calcul. Ce que nous souhaitons faire, c'est raffiner `rat` vers `Q C := C * C` pour n'importe quel type `C` raffinant lui-même `int`. La section suivante explique comment programmer génériquement des algorithmes dans le contexte de tels raffinements paramétrés. Par la suite, dans la section 2.3, nous montrerons que la correction peut être prouvée dans le cas particulier où `C` est `int`, puis transportée vers n'importe quel autre raffinement en tirant parti de la parametricité.

2.2 PROGRAMMATION GÉNÉRIQUE

Le principal problème pour raffiner `rat` vers `Q C` pour tout `C` raffinant `int` est que pour définir par exemple l'addition sur `Q C`, il est nécessaire d'avoir accès à l'addition et la multiplication sur `C`. Ainsi, supposer seulement l'existence d'une relation de raffinement entre `int` et `C` n'est pas suffisant, il nous faut un mécanisme de *surcharge* pour désigner génériquement toute implémentation de l'addition sur une instance de `C`. Nous souhaitons également attacher des notations à ces opérations génériques. Pour ce faire, nous définissons des classes de type opérationnelles [B. Spitters et E. van der Weegen, 2011].

Par exemple, pour définir l'addition sur `Q C`, nous utilisons les classes `add` et `mul` suivantes :

```
Class add B := add_op : B -> B -> B.
Class mul B := mul_op : B -> B -> B.

Notation "+%C" := add_op.
Notation "x + y" := (add_op x y).
Notation "*%C" := mul_op.
Notation "x * y" := (mul_op x y).
```

Nous utiliserons les notations ainsi définies dans toute la suite. La fonction `addQ` peut maintenant s'écrire :

```
Instance addQ C `{add C, mul C} : add (Q C) :=
  fun x y => (x.1 *C y.2 +C y.1 *C x.2, x.2 *C y.2).
```

La notation `` `{add C, mul C} `` *réalise l'abstraction sur les classes de type représentant l'addition et la multiplication sur* `C`.

Nous ajoutons ici le type en indice des notations `+` et `*` pour éviter toute confusion. Afin de prouver la correction de l'addition définie ci-dessus, les opérations peuvent être instanciées par leur implémentation orientée preuve :

```
Instance add_int : add int := +int.
Instance mul_int : mul int := *int.

Instance param_addQ : param (Rrat ==> Rrat ==> Rrat) +rat +Qint.
```

D'abord, l'addition et la multiplication sur le type `int` sont déclarées comme instances. Ensuite, le lemme `param_addQ` peut être prouvé en utilisant toute la théorie fournie par SSReflect sur les entiers et les nombres rationnels. Pour effectuer des calculs plus efficaces, il suffit d'instancier ensuite ce schéma avec un type qui raffine `int`, par exemple `Z`, et les opérations associées. Ainsi, l'utilisation de classes de type opérationnelles simplifie le développement et la maintenance du code car une seule description de l'algorithme est requise, grâce à l'abstraction à la fois sur le type des objets et sur les opérations mises en jeu. Le même code peut être utilisé pour des preuves de correction et des calculs efficaces, en changeant simplement l'instanciation.

Une remarque importante est que les classes de types opérationnelles n'ayant qu'un seul champ, elles sont représentées en interne par de simples définitions [B. Spitters et E. van der Weegen, 2011]. En particulier, contrairement aux classes de types à plusieurs champs, leur utilisation n'introduit pas de construction de structures ni de projections, ce qui aurait pu avoir un impact négatif sur la performance des instances orientées calcul. Le seul inconvénient de ce point de vue est la multiplication du nombre d'arguments des fonctions, mais des tests de performance nous ont montré que l'incidence n'était que marginale.

2.3 PARAMÉTRICITÉ ET AUTOMATISATION

L'approche présentée dans la section précédente, permettant l'écriture d'algorithmes génériques est très pratique, mais pose un nouveau problème : comment garantir la correction des instanciations de ces algorithmes sur des structures de données orientées calcul ? Faire manuellement une preuve pour chaque instance serait coûteux et préjudiciable à l'extensibilité du système : si un algorithme est exprimé génériquement sur des nombres entiers et qu'une nouvelle représentation des entiers est développée par la suite, il est légitime d'espérer obtenir à moindre frais une preuve de correction de l'algorithme instancié à cette nouvelle représentation et aux opérateurs associés.

Nous allons présenter ici une solution à ce problème, basée sur le théorème de paramétricité. Lorsqu'un algorithme générique a été vérifié en l'instanciant à des structures adaptées à la preuve, cette technique permettra de générer automatiquement la preuve de raffinement de toute autre instanciation, notamment à des structures orientées calcul. Dans le cas des nombres rationnels, cela nous permettra de prouver automatiquement la correction de `addQ` sur `Q C` dès lors que le type `C` raffine `int` et que l'addition et la multiplication sur `C` raffinent celles sur `int`.

2.3.1 Décomposition des relations de raffinement

Nous allons illustrer le processus mis en place en poursuivant l'exemple de l'addition `addQ` sur les nombres rationnels, introduit dans la section précédente.

Rappelons qu'une instance de `addQ` sur un type `C` dépend de deux opérateurs $+_C$ et $*_C$. Supposons que ces deux opérateurs raffinent respectivement l'addition et la multiplication sur `int` pour une relation `Rint` :

```
Context '{!param (Rint ==> Rint ==> Rint) +int +C}.
Context '{!param (Rint ==> Rint ==> Rint) *int *C}.
```

La commande `Context` *permet d'abstraire le contexte par rapport à des instances de classes de type.*

Sous ces hypothèses, nous voulons montrer qu'à partir de la preuve de correction `param_addQ` de la section précédente, nous pouvons automatiquement établir la correction de `addQ` instanciée sur `Q C`. En notant `\o` la composition de relations (`R \o R' x z := exists y, R x y /\ R y z`) et $*$ le produit de relations (`R * R' x y := R x.1 y.1 /\ R' x.2 y.2`), nous définissons la relation suivante :

```
Definition RratC := Rrat \o (Rint * Rint).
```

Cette relation `RratC : rat -> Q C -> Prop` est la composition des relations de raffinement entre `rat` et `Q int` d'une part et entre `Q int` et `Q C` d'autre part. Le lemme que nous souhaitons prouver s'écrit alors :

```
Lemma refines_addQ :
  param (RratC ==> RratC ==> RratC) +rat +QC.
```

Pour le prouver, nous découpons en deux étapes :

```
param (Rrat ==> Rrat ==> Rrat) +rat +Qint
getparam (Rint * Rint ==> Rint * Rint ==> Rint * Rint) +Qint +QC
```

La classe de type `getparam` *est un alias pour* `param`*, dont nous verrons l'utilité dans la section 2.3.2.*

Nous souhaitons réaliser ce découpage de manière automatique, selon la façon dont la relation de raffinement (ici, `RratC`) est composée (ici, à partir de `Rrat` et `Rint * Rint`). Pour ce faire, nous utilisons la possibilité de simuler des petits programmes logiques grâce aux classes de types [G. Gonthier, B. Ziliani et al., 2011 ; B. Spitters et E. van der Weegen, 2011]. Ici, la recherche du découpage sera réalisée par la résolution d'instances de la classe `composable` :

```
Class composable {A B C} (rAB : A -> B -> Prop) (rBC : B -> C ↙
    ↳ -> Prop) (rAC : A -> C -> Prop) :=
  Composable : rAB \o rBC <= rAC.
```

Le symbole `<=` *dénote l'inclusion des relations.*

Le déclenchement de la recherche est assuré par l'application du lemme suivant :

```
Lemma param_trans A B C (rAB : A -> B -> Prop) (rBC : B -> C -> ↙
    ↳ Prop) (rAC : A -> C -> Prop) (a : A) (b : B) (c : C) :
  composable rAB rBC rAC -> param rAB a b ->
  getparam rBC b c -> param rAC a c.
```

Nous définissons deux instances de la classe `composable` de manière à implémenter le programme logique suivant :

$$\frac{\texttt{composable R1 R2 R3}}{\texttt{composable (rAB ==> R1) (rBC ==> R2) (rAB \o rBC ==> R3)}}$$

$$\frac{}{\texttt{composable rAB rBC (rAB \o rBC)}}$$

2.3.2 Paramétricité pour les raffinements

À l'issue du découpage de la relation de raffinement, le premier sous-but que nous obtenons est `param (Rrat ==> Rrat ==> Rrat)` $+_{\text{rat}}$ $+_{\text{Qint}}$, c'est-à-dire exactement le lemme de correction `refines_addQ` de la section 2.2. Il nous reste donc à prouver l'énoncé suivant[7] :

[7] *Nous l'écrivons ici comme un lemme pour pouvoir y faire référence dans la suite, mais dans le développement réel, cet énoncé n'apparaît que temporairement au cours de la recherche de preuve*

```
Lemma getparam_addQ :
getparam (Rint * Rint ==> Rint * Rint ==> Rint * Rint) +Qint +QC
```

En étudiant la sémantique du polymorphisme dans le système F, Reynolds a fourni une interprétation relationnelle des types [J. C. Reynolds, 1983]. Le théorème de paramétricité [P. Wadler, 1989] est une reformulation basée sur le fait que si un type n'a pas de variable libre, l'interprétation relationnelle exprime une propriété partagée par tous les termes de ce type. Ce résultat s'étend aux systèmes de types purs [J.-P. Bernardy et al., 2012] et fournit, de manière remarquablement uniforme, une transformation notée $[\![\cdot]\!]$ qui s'applique aussi bien à un contexte Γ, à un type T pour construire l'interprétation relationnelle $[\![T]\!]$, qu'à un terme t de type T pour construire une preuve $[\![t]\!]$.

Nous allons montrer que cette idée nous donne un moyen de prouver de manière systématique l'énoncé `getparam_addQ`. Rappelons que nous avons décrit un algorithme de manière générique, en abstrayant le type des objets mis en oeuvre. Puis, nous avons instancié ces paramètres à des types orientés preuve et vérifié leur correction. Par paramétricité, cette propriété de correction est en faite partagée par toutes les instances du programme générique (pour peu que les instances des opérateurs soient elles-mêmes correctes).

Plus précisément, dans le cadre d'un système de types pur, en reprenant les notations de [C. Keller et M. Lasson, 2012], la transformation $[\![\cdot]\!]$ est définie de la façon suivante :

La notation A' désigne le terme A où chaque variable libre x a été remplacée par une variable fraîche x'. Le symbole s dénote une sorte.

$$
\begin{aligned}
[\![\varnothing]\!] &= \varnothing \\
[\![\Gamma, x : A]\!] &= [\![\Gamma]\!], x : A, x' : A', x_R : [\![A]\!]\, x\, x' \\
\\
[\![s]\!] &= \lambda(x : s)(x' : s).\, x \to x' \to s \\
[\![x]\!] &= x_R \\
[\![\forall x : A.\, B]\!] &= \lambda(f : \forall x : A.\, B)(f' : \forall x' : A'.\, B'). \\
&\qquad \forall(x : A)(x' : A')(x_R : [\![A]\!]\, x\, x').\, [\![B]\!]\, (f\, x)\, (f'\, x') \\
[\![\lambda x : A.\, B]\!] &= \lambda(x : A)(x' : A')(x_R : [\![A]\!]\, x\, x').\, [\![B]\!] \\
[\![(A\ B)]\!] &= ([\![A]\!]\ B\ B'\ [\![B]\!])
\end{aligned}
$$

Théorème 2.1. *Le théorème d'abstraction, dans sa forme générale, assure que si* $\Gamma \vdash A : B$ *alors* $[\![\Gamma]\!] \vdash A : B$, $[\![\Gamma]\!] \vdash A' : B'$ *et* $[\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]\, A\, A'$.

Ce résultat n'est valable que dans une certaine classe de systèmes de types purs, dits réflexifs [J.-P. Bernardy et al., 2012]. Cependant, nous ne rentrons pas ici dans les détails car il s'agit juste de s'inspirer de la méthode de génération des preuves.

Dans notre contexte, nous n'aurons besoin que du théorème de paramétricité qui est une spécialisation du résultat précédent aux termes clos :

Corollaire 2.1. *Si* $\vdash A : B$ *alors* $\vdash [\![A]\!] : [\![B]\!]\, A\, A$.

Exemple 2.1. *Grâce à ce corollaire, on peut par exemple montrer que toute fonction* $f : \forall A.\, A \to A$ *se comporte comme l'identité. En effet, on sait que* $[\![f]\!]$ *est alors une preuve de* $[\![\forall A.\, A \to A]\!]\, f\, f$*. Or d'après la définition de* $[\![\cdot]\!]$ *:*

$$\begin{aligned}[\![\forall A.\, A \to A]\!]\, f\, f' &= \forall (A : \mathit{Type})(A' : \mathit{Type})(A_R : A \to A' \to \mathit{Type}).\\ &\qquad [\![A \to A]\!]\, (f\, A\, x)\, (f'\, A'\, x')\\ &= \forall (A : \mathit{Type})(A' : \mathit{Type})(A_R : A \to A' \to \mathit{Type}).\\ &\qquad \forall (x : A)(x' : A').\, A_R\, x\, x' \to A_R\, (f\, A\, x)\, (f'\, A'\, x')\end{aligned}$$

Soient maintenant un type A *et une fonction* $g : A \to A$ *arbitraires. Il est facile de représenter* g *par une relation* $h : A \to A \to \mathit{Type}$*.*

Le terme $[\![f]\!]\, A\, A\, h$ *est alors une preuve que si deux éléments* x *et* x' *de type* A *sont en relation par* h*,* $f\, A\, x$ *et* $f\, A x'$ *sont en relation par* h*. C'est-à-dire que* f *commute avec* g *:* $f \circ g = g \circ f$*. Comme* g *est une fonction arbitraire, cela implique que* f *est l'identité.*

Nous allons suivre la même méthodologie pour construire une preuve de `getparam_addQ`. Plus formellement, nous prouvons le lemme de paramétricité suivant :

$$\begin{aligned}&[\![\forall Z, (Z \to Z \to Z) \to (Z \to Z \to Z) \to Q\ Z \to Q\ Z \to Q\ Z]\!]\ \mathtt{addQ}\ \mathtt{addQ} =\\ &\forall Z : \mathtt{Type}, \forall Z' : \mathtt{Type}, \forall Z_R : Z \to Z' \to \mathtt{Prop},\\ &\forall \mathtt{addZ} : Z \to Z \to Z, \forall \mathtt{addZ}' : Z' \to Z' \to Z', [\![Z \to Z \to Z]\!]\ \mathtt{addZ}\ \mathtt{addZ}' \to\\ &\forall \mathtt{mulZ} : Z \to Z \to Z, \forall \mathtt{mulZ}' : Z' \to Z' \to Z', [\![Z \to Z \to Z]\!]\ \mathtt{mulZ}\ \mathtt{mulZ}' \to\\ &[\![Q\ Z \to Q\ Z \to Q\ Z]\!]\ (\mathtt{addQ}\ \mathtt{addZ}\ \mathtt{mulZ})\ (\mathtt{addQ}\ \mathtt{addZ}'\ \mathtt{mulZ}')\end{aligned}$$

En effet, en instanciant ensuite Z à `int`, Z' à `C` et Z_R à `Rint`, cette preuve permettra de construire, à partir des preuves de correction de l'addition et de la multiplication sur `C`, une preuve de la correction de l'addition sur `Q C`.

Cependant, le théorème de paramétricité n'est pas prouvable de manière interne dans le formalisme de Coq. La transformation $[\![\cdot]\!]$ fournit néanmoins une procédure pour construire, de manière externe, les preuves des énoncés de paramétricité. Une extension de Coq a ainsi été développée [C. Keller et M. Lasson, 2012] mais au moment de l'écriture de ces lignes, elle ne prend pas en charge les types inductifs, qui nous sont indispensables.

Nous faisons donc le choix d'implémenter notre propre procédure de recherche de preuve pour ces énoncés de paramétricité. Ici encore, nous utilisons le système de classe de type pour exprimer un programme logique réalisant cette recherche. Les règles en sont les suivantes :

La classe de type `getparam` *sert à guider l'application de ces règles. Autrement, c'est un synonyme de* `param`*.*

$$\frac{\texttt{forall a b, param R a b -> getparam R' (f a) (g b)}}{\texttt{getparam (R ==> R') f g}}\ \texttt{pop\_param}$$

$$\frac{\texttt{param (R ==> R') f g} \qquad \texttt{param R a b}}{\texttt{param R' (f a) (g b)}}\ \texttt{push\_param}$$

$$\frac{\texttt{param R x y}}{\texttt{getparam R x y}}\ \texttt{set\_param}$$

$$\frac{\texttt{R x y}}{\texttt{param R x y}}\ \texttt{trivial\_param}$$

2.3.3 Exemple

Afin d'illustrer le fonctionnement de la recherche de preuve des énoncés de paramétricité, nous allons montrer l'exécution du programme logique défini à la section précédente sur l'énoncé `getparam_addQ` :

```
getparam (Rint * Rint ==> Rint * Rint ==> Rint * Rint) +Qint +QC
```

Du fait des priorités[8] que nous avons données aux règles, c'est d'abord `pop_param` qui est appliquée. Le but suivant est obtenu :

[8] Le système de classes de types de Coq permet de définir des priorités pour certaines instances. Nous avons choisi ces priorités principalement en fonction de contraintes d'efficacité.

```
forall a b, param (Rint * Rint) a b ->
  getparam (Rint * Rint ==> Rint * Rint) (addQ int a) (addQ C b)
```

Notre procédure introduit alors automatiquement `a`, `b` ainsi que l'hypothèse `param (Rint * Rint) a b` puis applique à nouveau la règle `pop_param` ce qui amène à prouver, sous les hypothèses `param (Rint * Rint) a b` et `param (Rint * Rint) c d` :

```
getparam (Rint * Rint) (a +Qint c) (b +QC d)
```

À ce stade, le mécanisme de recherche d'instance va déplier la définition de `addQ`, et obtenir des couples d'entiers :

```
getparam (Rint * Rint) (a.1 *C c.2 +C c.1 *C a.2, a.2 *C c.2)
  (b.1 *C d.2 +C d.1 *C b.2, b.2 *C d.2)
```

Pour conclure, nous devons fournir des instances du théorème de paramétricité pour le constructeur de couples `pair` et les projections `_.1` et `_.2`. Dans notre contexte, nous construisons manuellement les preuves de ces énoncés, mais leur nombre reste restreint comparé au nombre de leurs occurrences dans nos algorithmes génériques. Dans le cas des couples, nous obtenons :

Pour des raisons techniques, ces lemmes sont introduits dans le système de recherche de preuve par un mécanisme de « hints », plutôt que par des instances de classes de types, mais l'idée est la même.

```
Lemma param_fst RA RB : param (RA * RB ==> RA) _.1 _.1
Lemma param_snd RA RB : param (RA * RB ==> RB) _.2 _.2
Lemma param_pair RA RB : param (RA ==> RB ==> RA * RB) pair pair
```

En dépliant les définitions du lemme `param_pair`, on obtient :

```
forall (RA : A -> A' -> Prop) (RB : B -> B' -> Prop)
forall (a : A) (a' : A') (b : B) (b' : B'),
  RA a a' → RB b b' → (RA * RB) (a, b) (a', b')
```

Ainsi, ce lemme s'applique à notre but courant et nous donne deux nouveaux sous-buts :

```
getparam Rint (a.1 *int c.2 +int c.1 *int a.2)
  (b.1 *C d.2 +C d.1 *C b.2)
getparam Rint (a.2 *int c.2) (b.2 *C d.2)
```

La notation ? désigne une variable existentielle, car la relation de raffinement n'est pas connue à ce stade de la recherche de preuve. Elle sera trouvée quelques étapes après, par unification.

La règle `set_param` change la classe `getparam` en `param` dans ces deux sous-buts, ce qui permet d'appliquer alors la règle `push_param`. Dans le cas du deuxième sous-but (le premier est analogue), cette dernière règle ouvre à son tour deux sous-buts :

```
param (? ==> Rint) (mul_op int a.2) (mul_op C b.2)
param Rint c.2 d.2
```

Le second est une conséquence immédiate du lemme `param_snd`. Quant au premier, il déclenche une nouvelle application de `push_param` qui donne :

```
param (? ==> ? ==> Rint) *int *C
param Rint a.2 b.2
```

À ce moment, les deux sous-buts sont résolus : le premier s'unifie avec l'énoncé de correction de la multiplication sur `C`, qui est une hypothèse, tandis que le second découle comme précédemment de `param_snd`.

2.3.4 Méthodologie

Les concepts que nous avons introduits conduisent naturellement à l'organisation suivante des bibliothèques de notre développement :

1. Nous définissons notre structure raffinée (`Q`) et les opérations associées (`addQ`), paramétrée par les opérations abstraites ($+_C$, $*_C$) sur les types de base (`C`).

2. Nous prouvons que l'instanciation de ces opérations abstraites dans un contexte orienté preuve (`addQ int` $+_{int}$ $*_{int}$) est correcte :
   `param (Rrat ==> Rrat ==> Rrat)` $+_{rat}$ `(addQ int` $+_{int}$ $*_{int}$`)`

3. Nous supposons que les opérations sur le type de base orienté preuve raffinent celles sur le type de base abstrait correspondant :
   `param (Rint ==> Rint)` $+_{int}$ $+_C$
   Sous cette hypothèse, nous dérivons que les opérations sur le type orienté preuve cible se raffinent vers nos nouveaux opérateurs :
   `param (Rrat ==> Rrat ==> Rrat)` $+_{rat}$ `(addQ C` $+_{int}$ $*_{int}$`)`

2.4 TRAVAUX CONNEXES

Le travail présenté dans ce chapitre répond à un problème fondamental : comment changer des représentations de données d'une manière compositionnelle. Il n'est donc pas surprenant que de nombreux autres travaux partagent les mêmes motivations. Nous avons déjà fait référence aux modules et foncteurs à la ML, qui sont disponibles dans Coq, mais ne permettent pas d'utiliser des méthodes de preuves reposant sur le contenu calculatoire des objets.

L'exemple le plus général de relations de raffinement que nous considérons sont des quotients partiels, qui sont le plus souvent représentés en théorie des types par des setoïdes sur des relations d'équivalences partielles [G. Barthe, V. Capretta et al., 2003] et manipulés grâce à la réécriture généralisée [M. Sozeau, 2010]. Les techniques que nous utilisons sont très proches d'une version hétérogène de cette dernière approche. En effet, la réécriture généralisée met en jeu une relation `R : A -> A -> Prop` pour un type `A` donné, tandis que nos relations de raffinement sont de la forme `R : A -> B -> Prop` où `A` et `B` peuvent être deux types différents.

Une extension pour Coq a également été développée pour permettre le changement de représentation de données et de convertir les preuves associées de l'une à l'autre [N. Magaud et Y. Bertot, 2001]. Cependant, ce travail était limité à des types isomorphes, et ne permettait pas la programmation générique (seules les preuves étaient portées). Notre approche est donc plus

générale, et nous ne reposons pas sur une extension externe à Coq qui peut être coûteuse à maintenir.

Dans [Z. Luo, 1989], une méthodologie pour la spécification et le développement modulaires de programmes en théorie des types est présentée. L'idée clé est d'exprimer des spécifications algébriques en utilisant des sigma-types qui peuvent être *raffinés* au moyen de fonctions de raffinement et *réalisés* par des programmes concrets. Cette idée se rapproche des modules à la ML, puisque les objets sont abstraits et leur comportement est représenté par un ensemble de propriétés équationnelles. Une différence importante par rapport au travail présenté dans ce chapitre est que ces propriétés équationnelles sont exprimées en utilisant une relation de congruence abstraite, alors qu'un des objectifs que nous poursuivons est de prouver la correction de nos algorithmes sur des objets pouvant être comparés par l'égalité de Leibniz, ce qui rend le raisonnement plus pratique. Ceci est rendu possible par la relation que nous utilisons entre représentations orientées preuves ou calcul, qui est plus souple que la relation de *réalisation* sur les spécifications à base de sigma-types.

Un autre moyen de réconcilier l'abstraction de données et le contenu calculatoire est l'usage de *vues* [P. Wadler, 1987 ; C. McBride et J. McKinna, 2004]. En particulier, celles-ci permettent de dériver des schémas d'induction indépendamment des représentations concrètes des données. Cela peut être utile dans notre contexte pour écrire des programmes génériques, en ayant recourt à ces schémas pour définir des programmes récursifs et prouver des propriétés associées. Nous en verrons un exemple au chapitre 3.

La programmation générique a également déjà été étudiée en théorie des types, par exemple au moyen d'un encodage dans un univers [T. Altenkirch et al., 2006].

Le travail indépendant le plus proche de celui du présent chapitre est probablement l'outil de raffinement de données automatique Autoref [P. Lammich, 2013] implémenté pour Isabelle. Tandis que de nombreuses idées, comme l'utilisation de la paramétricité, sont proches des nôtres, le parti pris est de s'appuyer sur un outil externe pour synthétiser des instances exécutables d'algorithmes génériques, ainsi que des preuves de raffinement. Le formalisme plus riche que nous avons à notre disposition, en particulier les types dépendants, nous a permis au contraire d'internaliser l'instanciation des programmes génériques.

2.5 CONCLUSION

Dans ce chapitre, nous avons présenté une approche aux raffinements de données permettant à l'utilisateur de fournir une seule description générique d'un programme et de le porter automatiquement à de nouvelles représentations de données, en transportant également les preuves de correction. Nous pensons que ceci répond à un besoin très général qui est souvent traité de manière ad-hoc. Nos trois principales contributions sont :

- Une interface de raffinements légère et générale supportant toute relation hétérogène entre deux types.

- Un mécanisme d'implémentation générique d'algorithmes supportant de nombreuses constructions de la librairie SSReflect utilisant des classes de types opérationnelles

- Le transport automatique des preuves de correction d'une instance à une autre inspirée du théorème de paramétricité.

Une première version de ce travail, réalisée en collaboration avec Anders Mörtberg et Vincent Siles a fait l'objet d'une publication [M. Dénès et al., 2012]. Cette première version prenait en charge des représentations isomorphes ou partiellement isomorphes (c'est-à-dire un type dont un sous-domaine est isomorphe à un autre type). En revanche, le cas des types quotients n'était pas traité. En outre, plusieurs représentations des algorithmes étaient nécessaire (ce qui est rendu superflu par la programmation générique) et le transport des preuves de correction était fait de manière ad-hoc (partiellement automatisée).

Nous avons ensuite amélioré tous ces points comme décrit dans ce chapitre, en collaboration avec Anders Mörtberg et Cyril Cohen. Ceci a donné lieu à une deuxième publication [C. Cohen et al., 2013]. Signalons que l'idée d'utiliser le théorème de paramétricité comme support pour le transport des preuves de correction est de Cyril Cohen.

L'utilisation des classes de types opérationnelles est très pratique pour la programmation générique. Mais sur des programmes complexes, le nombre d'arguments implicites augmentant, il se peut que nous rencontrions des problèmes de performance. Soit lors de la vérification de types des programmes génériques, soit lors de l'évaluation. En effet, pour ce second aspect, il faut préciser que l'évaluateur OCaml que nous utilisons optimise les appels de fonctions en tirant profit des registres du processeur pour stocker des arguments. Si le nombre d'arguments augmente et dépasse celui des registres, des arguments seront stockées sur la pile et les performances peuvent être impactées. Cependant, nous n'avons jusqu'ici rien mesuré de significatif. Le nombre d'arguments peut être réduit en les empaquetant dans des structures, mais des opérations de projections sont alors nécessaires, et peuvent également avoir un impact.

La prise en charge de la paramétricité est actuellement réalisée par métaprogrammation mais nécessite une intervention de l'utilisateur pour écrire les énoncés de paramétricité, même si l'essentiel des preuves est automatisé. De plus, notre procédure ne traite que les constructions polymorphes. Des améliorations pourraient être apportées en fournissant une construction systématique des lemmes de paramétricité pour les types inductifs [J.-P. Bernardy et al., 2012] et en étendant les constructions de relations de raffinements avec des types dépendants. Un rapprochement avec l'implémentation décrite dans [C. Keller et M. Lasson, 2012] serait également souhaitable.

Toutes les formalisations décrites dans ce chapitre ont été réalisées dans une version standard de Coq, mais il serait intéressant de voir comment les avancées récentes liées à la théorie des types homotopiques pourraient être utilisées pour simplifier notre traitement des raffinements de données. En particulier, dans un contexte où l'axiome d'univalence serait disponible, les structures isomorphes seraient égales [B. Ahrens et al., 2013 ; N. A. Danielsson et T. Coquand, 2013] ce qui devrait être utile pour le cas des raffinements entre deux types isomorphes. De plus, la théorie homotopique des types fournit des outils pour représenter les types quotients (voir par exemple [E. Rijke et B. Spitters, 2013]) qui pourraient être mis à profit pour le cas des raffinements de types représentant un quotient ou même un quotient partiel.

Dans la suite de cette thèse, et plus particulièrement dans la partie II, nous appliquerons les outils du présent chapitre à des algorithmes variés en algèbre linéaire. Outre l'intérêt propre de ces algorithmes, cela nous permettra

d'éprouver les techniques que nous avons discutées et en particulier de voir si elles passent à l'échelle lorsque la taille et la complexité des programmes devient plus importante.

Deuxième partie

ALGÈBRE LINÉAIRE FORMALISÉE

3 PRODUIT MATRICIEL DE STRASSEN

Le présent chapitre entame la partie de cette thèse consacrée à l'étude proprement dite d'algorithmes en algèbre linéaire. La plupart de ceux que nous étudierons sont présentés dans [J. Abdeljaoued et H. Lombardi, 2003]. Il peut paraître étonnant que nous ne commencions pas cette étude par la méthode de Gauss pour la résolution de système linéaires, malgré son caractère historique et omniprésent dans les présentations modernes de l'algèbre linéaire. Cependant, du point de vue de la complexité algébrique, le célèbre algorithme décrit par Volker Strassen pour le produit matriciel [V. Strassen, 1969] est peut-être plus fondamental encore. Outre la surprise qu'a constituée ce résultat, car il améliore l'exposant de la complexité de la multiplication de matrices par rapport à l'algorithme « naïf », ce que beaucoup croyaient impossible,[9] son intérêt est qu'une famille de problèmes importants en algèbre linéaire peuvent être réduits à la multiplication de matrices, et donc tirer parti de cette amélioration de complexité. Citons notamment l'inversion de matrices, le calcul du déterminant, la décomposition LUP ou la résolution de systèmes linéaires.

[9] « *[…] it touched off widespread speculation that* $n^3/2$ *multiplications would be necessary to multiply* $n \times n$ *matrices, because of the somewhat similar lower bound that was known to hold for polynomials in one variable.* »
[D. E. Knuth, 1997, p. 500]

Afin d'expliquer le principe de l'algorithme de Strassen, commençons par rappeler qu'étant donnés un anneau $\mathcal{R}$ et deux matrices $(a_{i,j}) \in \mathcal{R}^{m\times n}$ et $(b_{i,j}) \in \mathcal{R}^{n\times p}$, leur produit $(c_{i,j}) \in \mathcal{R}^{m\times p}$ est défini par :

$$\forall i\ \forall j\ c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$$

L'algorithme naturel pour le calcul de ce produit consiste à calculer les mp coefficients $c_{i,j}$ définis ci-dessus, chacun comportant $n-1$ sommes et n multiplications. Le nombre d'opérations final est donc :

$$T_{\text{naïf}}(m,n,p) = mp(2n-1) = \mathcal{O}(mnp)$$

Pour simplifier, on considérera dans la suite le produit de deux matrices carrées de dimension n, dont la complexité est alors :

$$T_{\text{naïf}}(n) = n^2(2n-1) = \mathcal{O}(n^3)$$

Supposons maintenant que les deux matrices en jeu ont une taille de la forme $n = 2^{k+1}$ et exprimons ce produit par blocs de taille 2^k :

$$\left[\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array}\right] \times \left[\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \end{array}\right] = \left[\begin{array}{c|c} C_{1,1} & C_{1,2} \\ \hline C_{2,1} & C_{2,2} \end{array}\right]$$

On a alors les identités suivantes :

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

Le calcul de chacun des blocs $C_{i,j}$ requiert 2 produits et 1 addition de matrices de taille $\frac{n}{2} = 2^k$. Ces blocs sont au nombre de 4, la complexité vérifie donc :

$$T_{\text{blocs}}(n) = T_{\text{blocs}}(2^{k+1}) = 4 \times 2^{2k} + 8T_{\text{blocs}}(2^k)$$

Ce qui, après calcul, nous donne :

$$\begin{aligned}
T_{\text{blocs}}(2^{k+1}) &= 4^{k+1} + 8T_{\text{blocs}}(2^k) \\
&= 4^{k+1} + 8(4^k + 8T_{\text{blocs}}(2^{k-1})) \\
&= 4^{k+1} + 8 \times 4^k + 8^2 \times 4^{k-1} + \cdots + 8^{k+1} \\
&= \sum_{i=0}^{k+1} 8^i 4^{k+1-i} \\
&= 4^{k+1} \sum_{i=0}^{k+1} 2^i \\
&= 2^{2k+2}(2^{k+2} - 1) = \mathcal{O}((2^{k+1})^3) = \mathcal{O}(n^3)
\end{aligned}$$

Il est important de remarquer que la complexité est ici contrôlée par le nombre de multiplications à chaque étape récursive (ici 8). L'algorithme de Strassen est essentiellement une réexpression du produit par blocs avec seulement 7 multiplications et 18 additions :

$$\begin{aligned}
P_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
P_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\
P_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\
P_4 &= A_{2,2}(-B_{1,1} + B_{2,1}) & C_{1,1} &= P_1 + P_4 - P_5 + P_7 \\
P_5 &= (A_{1,1} + A_{1,2})B_{2,2} & C_{2,1} &= P_2 + P_4 \\
P_6 &= (-A_{1,1} + A_{2,1})(B_{1,1} + B_{1,2}) & C_{1,2} &= P_3 + P_5 \\
P_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) & C_{2,2} &= P_1 + P_3 - P_2 + P_6
\end{aligned}$$

Le point clé est que ces identités ne reposent pas sur la commutativité de la multiplication, et peuvent donc être appliquées récursivement sur des blocs des matrices d'entrée. La complexité devient alors :

$$T_{\text{Strassen}}(2^{k+1}) = 7T_{\text{Strassen}}(2^k) + 18 \times 2^{2k}$$

$$T_{\text{Strassen}}(n) = \mathcal{O}(n^{\log 7})$$

Winograd améliore un peu en limitant le nombre d'additions à 15 [S. Winograd, 1971]. La complexité asymtotique est inchangée, mais le gain peut tout de même être sensible. C'est cette présentation que nous garderons dans la suite :

$$\begin{aligned}
S_1 &= A_{2,1} + A_{2,2} & P_1 &= A_{1,1} \times B_{1,1} & U_1 &= P_1 + P_6 \\
S_2 &= S_1 - A_{1,1} & P_2 &= A_{1,2} \times B_{2,1} & U_2 &= U_1 + P_7 \\
S_3 &= A_{1,1} - A_{2,1} & P_3 &= S_4 \times B_{2,2} & U_3 &= U_1 + P5 \\
S_4 &= A_{1,2} - S_2 & P_4 &= A_{2,2} \times T_4 & C_{1,1} &= P_1 + P_2 \\
T_1 &= B_{1,2} - B_{1,1} & P_5 &= S_1 \times T_1 & C_{1,2} &= U_3 + P_3 \\
T_2 &= B_{2,2} - T_1 & P_6 &= S_2 \times T_2 & C_{2,1} &= U_2 - P_4 \\
T_3 &= B_{2,2} - B_{1,2} & P_7 &= S_3 \times T_3 & C_{2,2} &= U_2 + P_5 \\
T_4 &= T_2 - B_{2,1}
\end{aligned}$$

L'origine de l'idée de Strassen n'est pas claire, mais elle a ouvert la voie à de nombreux travaux sur la complexité du produit matriciel. Pour un corps

$\mathcal{K}$, nous considérons l'ensemble des réels x tels que le produit de deux matrices quelconques de $\mathcal{K}^{n\times n}$ peut être effectué en $\mathcal{O}(n^x)$ opérations arithmétiques. Notons ω la borne inférieure de cet ensemble. Il est immédiat que $\omega \geq 2$ car la construction du produit nécessite le calcul de n^2 coefficients. Le résultat de Strassen montre que $\omega \leq \log 7 < 2.807$. En fait, ω ne dépend que de la caractéristique de $\mathcal{K}$ [A. Schönhage, 1981], même si tous les algorithmes décrits jusqu'à ce jour s'appliquent indifféremment à tous les corps. Par ailleurs, il sera établi que 7 multiplications sont nécessaires pour multiplier deux matrices 2×2 sur un anneau non commutatif [J. E. Hopcroft et L. R. Kerr, 1971], et que dans ce cas 15 additions sont également requises [R. L. Probert, 1976]. La variante de Winograd présentée ci-dessus est donc optimale parmi les approches basées sur l'itération de la multiplication des matrices 2×2.

Bien sûr, d'autres approches ont été développées, à commencer par l'itération de la multiplication de matrices plus grandes que 2×2. La valeur de ω n'est toujours pas connue à ce jour, même si de nombreux algorithmes ont peu à peu amélioré les majorants, le record actuel est $\omega < 2.3727$ [V. V. Williams, 2012], affinant le précédent record vieux de vingt ans établi à $\omega < 2.376$ [D. Coppersmith et S. Winograd, 1990]. Cependant, l'intérêt de ces algorithmes est essentiellement théorique car seul celui de Strassen semble être utilisable en pratique. En effet, les constantes cachées dans la complexité des meilleurs algorithmes sont si grandes qu'ils ne deviennent efficaces que sur des matrices bien plus grandes que celles représentables sur les machines actuelles.

3.1 FORMALISATION DE L'ALGORITHME

La bibliothèque SSReflect fournit de nombreuses définitions et propriétés sur les objets usuels en algèbre linéaire [G. Gonthier, 2011]. Nous souhaitons donc réutiliser au maximum ces outils, et commençons par rappeler quelques notations.

Pour un anneau `R`, le type des matrices de taille `m × n` sur `R` est noté `'M[R]_(m,n)`. Pour des matrices carrées de taille `n`, ce type est abrégé en `'M[R]_n`. Lorsque l'anneau peut être inféré à partir du contexte, il peut être omis des notations précédentes, qui deviennent `'M_(m,n)` et `'M_n`.

Le produit matriciel est défini dans la bibliothèque comme suit :

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

Le domaine de l'indice `j` est inféré à partir du type des matrices `A` et `B`. Ainsi, `j` parcourt les entiers de `0` à `n`.

où le mot clé `\sum` désigne l'opérateur de somme itérée sur l'anneau `R`, défini à l'aide du mécanisme de grands opérateurs canoniques [Y. Bertot et al., 2008]. Le produit `mulmx A B` est noté `A *m B`.

Pour exprimer le découpage par blocs d'une matrice, la bibliothèque fournit les fonctions `ulsubmx`, `ursubmx`, `dlsubmx` et `drsubmx` qui extraient respectivement les blocs supérieur gauche, supérieur droit, inférieur gauche et inférieur droit. Le type de la matrice d'origine doit faire apparaître une somme dans le nombre de lignes et dans le nombre de colonnes. La taille des blocs est alors lue à partir de cette expression. Ainsi, si ce type est `'M_(m1 + m2, n1 + n2)`, les quatre blocs mentionnés auront pour tailles respectives `m1 × n1`, `m1 × n2`, `m2 × n1` et `m2 × n2`.

Une petite complication technique est que les tailles indexant le type des matrices sont exprimés par des entiers de Peano (type `nat`). Or le schéma

d'induction associé à ce type de données ne convient pas au mieux à notre contexte. En effet, chaque pas de récursion de l'algorithme de Strassen divise la taille des blocs par 2. Le type de données adapté à ce schéma est celui des entiers binaires défini dans la librairie standard de Coq :

Pour un entier `p`, `xI p` *représente* `2p+1` *tandis que* `xO p` *représente* `2p`. *Le constructeur* `xH` *encode l'entier* `1`.

```
Inductive positive : Set :=
  | xI : positive -> positive
  | xO : positive -> positive
  | xH : positive.
```

Pour utiliser de manière transparente des entiers de type `positive` comme tailles de matrices, nous utilisons une coercion :

```
Local Coercion nat_of_pos : positive >-> nat.
```

Nous pouvons maintenant décrire une étape de l'algorithme de Strassen. Nous adoptons la présentation de [B. Boyer et al., 2009], mais qui est sensible équivalente à celle présentée en introduction de ce chapitre :

```
Definition Strassen_step {p : positive} (A B : 'M_(p + p)) f :=
  let A11 := ulsubmx A in let A12 := ursubmx A in
  let A21 := dlsubmx A in let A22 := drsubmx A in
  let B11 := ulsubmx B in let B12 := ursubmx B in
  let B21 := dlsubmx B in let B22 := drsubmx B in
  let X := A11 - A21 in
  let Y := B22 - B12 in
  let C21 := f X Y in
  let X := A21 + A22 in
  let Y := B12 - B11 in
  let C22 := f X Y in
  let X := X - A11 in
  let Y := B22 - Y in
  let C12 := f X Y in
  let X := A12 - X in
  let C11 := f X B22 in
  let X := f A11 B11 in
  let C12 := X + C12 in
  let C21 := C12 + C21 in
  let C12 := C12 + C22 in
  let C22 := C21 + C22 in
  let C12 := C12 + C11 in
  let Y := Y - B21 in
  let C11 := f A22 Y in
  let C21 := C21 - C11 in
  let C11 := f A12 B21 in
  let C11 := X + C11 in
  block_mx C11 C12 C21 C22.
```

La fonction `block_mx` appelée à la fin de l'étape reconstruit la matrice composée des quatre blocs passés en arguments. L'abstraction de la fonction `f` est simplement un moyen pratique d'exprimer la récursion mutuelle par un encodage d'ordre supérieur. Cet argument `f` est destiné à être instancié par la procédure globale représentant l'algorithme de Strassen. La correction de l'étape s'exprime donc de la manière suivante :

```
Lemma Strassen_stepP (p : positive) (A B : 'M[R]_(p + p)) f :
```

```
f =2 mulmx -> Strassen_step A B f = A *m B.
```

Le symbole `=2` exprime l'égalité extensionnelle de deux fonctions binaires : pour des types `T`, `U`, `V` et `f g : T -> U -> V`, `f =2 g` dénote la proposition `forall x y, f x y = g x y`.

Afin de rendre confortable la preuve de `Strassen_stepP`, qui consiste essentiellement à vérifier les identités algébriques mises en jeu dans l'algorithme, nous utilisons la tactique `non_commutative_ring` introduite dans la version 8.4 de Coq. Elle permet, comme son nom l'indique, d'automatiser la preuve des identités sur un anneau non commutatif, ici celui des matrices carrées de dimension `p + p` à coefficients dans `R`.

Pour exprimer l'algorithme global, nous prenons en compte le fait qu'il est préférable de traiter le produit de petites matrices par l'algorithme « naïf », qui est en fait plus rapide que celui de Strassen jusqu'à une certaine taille. Nous paramétrons donc par un entier `K`. Dès que la procédure récursive atteindra des matrices de taille inférieure à `K`, l'algorithme naïf sera utilisé.

```
Variable K : nat.
```

Une autre question se pose : que faire si des matrices de tailles impaires apparaissent, c'est-à-dire si les matrices de départ avaient des tailles qui n'étaient pas des puissances de 2 ? Quatre approches sont possibles. Une première, appelée « static padding » (remplissage statique), consiste à compléter, avant le début de l'exécution de l'algorithme, les matrices d'entrée par des lignes et des colonnes de zéros, jusqu'à obtenir des tailles qui sont des puissances de 2. Puis de tronquer le résultat en sortie. De manière duale, on peut extraire des matrices d'entrée les plus grandes sous-matrices ayant pour taille une puissance de 2, puis ajuster le résultat en sortie. C'est le « static peeling » (pelage statique).

Deux autres approches, dites dynamiques, consistent à traiter le problème en cours d'algorithme. Lorsqu'une des matrices présente un nombre impair de lignes (resp. de colonnes), on peut rajouter une ligne (resp. une colonne) de zéros (c'est le « dynamic padding » ou remplissage dynamique) ou au contraire en enlever une, en n'oubliant pas d'ajuster le résultat de l'étape de calcul (c'est le « dynamic peeling » ou pelage dynamique).

Aucune de ces approches n'est meilleure que les autres sur l'ensemble des cas pouvant se présenter. Cependant, nous avons choisi d'implémenter la dernière (« dynamic peeling ») parce qu'elle est réputée donner de bons résultats en pratique et que son caractère dynamique éprouvera mieux la flexibilité de notre formalisation que les approches statiques, plus faciles à représenter. En effet, être capable de concilier l'encodage des tailles des matrices dans leur type (fondamentalement statique) avec des critères dynamiques est un enjeu important.

Pour le réaliser, nous aurons recours à la fonction de conversion de type sur les matrices `castmx`. Elle permet, étant données une matrice `M` de type `'M_(m1,n1)` et des preuves d'égalité `m1 = m2` et `n1 = n2`, de construire une copie de `M` dans le type `'M_(m2,n2)`. Nous l'utiliserons en particulier avec des instances des preuves d'égalités suivantes :

```
Lemma addpp p : x0 p = p + p :> nat.

Lemma addp1 p : xI p = x0 p + 1 :> nat.

Lemma esym x y : x = y -> y = x.
```

La notation `x = y :> T` *permet de plonger les termes* `x` *et* `y` *dans* `T` *au moyen de coercions si nécessaires avant de les comparer.*

Pour comprendre comment fonctionne le pelage dynamique, considérons deux matrices A et B de taille $n+1$. Nous les découpons chacune en quatre blocs et notons $A_{1,1}$ (resp. $B_{1,1}$) le bloc supérieur gauche de taille $n \times n$, $A_{2,1}$ (resp. $B_{2,1}$) le bloc supérieur droit de taille $1 \times n$, $A_{1,2}$ (resp. $B_{1,2}$) le bloc inférieur gauche de taille $n \times 1$ et enfin a (resp. b) l'unique élément du bloc inférieur droit. Leur produit s'écrit de la façon suivante :

$$\left[\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & a \end{array}\right] \times \left[\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & b \end{array}\right] = \left[\begin{array}{c|c} A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & R_{1,2} \\ \hline R_{2,1} & R_{2,2} \end{array}\right]$$

avec :

$$\begin{aligned} R_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}b \\ R_{2,1} &= A_{2,1}B_{1,1} + aB_{2,1} \\ R_{2,2} &= A_{2,1}B_{1,2} + ab \end{aligned}$$

Seul le produit $A_{1,1}B_{1,1}$ met en jeu deux matrices qui ne sont pas réduites à une ligne, une colonne ou un élément. C'est donc ce produit que l'on va calculer récursivement. Pour les autres, on utilise l'algorithme naïf de multiplication de matrices. En Coq, cela donne la fonction :

```
Fixpoint Strassen {n : positive} {struct n} :=
  match n return let M := 'M[R]_n in M -> M -> M with
  | xH => fun A B => A *m B
  | xO p => fun A B =>
    if p <= K then A *m B else
    let A := castmx (addpp p,addpp p) A in
    let B := castmx (addpp p,addpp p) B in
    castmx (esym (addpp p), esym (addpp p)) (Strassen_step A B ↙
        ↳ Strassen)
  | xI p => fun A B =>
    if p <= K then A *m B else
    let A := castmx (addpp1 p, addpp1 p) A in
    let B := castmx (addpp1 p, addpp1 p) B in
    let A11 := ulsubmx A in let A12 := ursubmx A in
    let A21 := dlsubmx A in let A22 := drsubmx A in
    let B11 := ulsubmx B in let B12 := ursubmx B in
    let B21 := dlsubmx B in let B22 := drsubmx B in
    let C := Strassen_step A11 B11 Strassen + A12 *m B21 in
    let R12 := A11 *m B12 + A12 *m B22 in
    let R21 := A21 *m B11 + A22 *m B21 in
    let R22 := A21 *m B12 + A22 *m B22 in
    castmx (esym (addpp1 p), esym (addpp1 p)) (block_mx C R12 ↙
        ↳ R21 R22)
end.
```

Sa correction s'établit par une induction immédiate et l'utilisation du lemme précédent :

```
Lemma StrassenP (n : positive) (M N : 'M[R]_n) :
  Strassen M N = M *m N.
```

À notre connaissance, cette preuve constitue la première vérification formelle complète de l'algorithme de Strassen. Une formalisation antérieure existe en ACL2, mais elle est limitée aux matrices dont la taille est une puissance de 2 [F. Palomo-Lozano et al., 2001].

3.2 EXTENSION AUX MATRICES RECTANGULAIRES

Dans la section précédente, nous avons prouvé la correction de l'algorithme de Strassen pour des matrices carrées. Informellement, tous les arguments mathématiques mis en jeu peuvent être transposés tels quels au cas rectangulaire. Cependant, comme nous l'avons mentionné, notre preuve formelle utilise la tactique `non_commutative_ring` qui permet de prouver automatiquement des identités sur un anneau. Or, dans le cas de matrices rectangulaires, nous n'avons plus immédiatement de structure d'anneau. En effet, si l'on considère les matrices de taille fixée, la multiplication n'est plus une loi interne. Si au contraire l'on oublie les tailles, elle devient une opération partielle. Nous donnons dans cette section des pistes pour réaliser cette extension, même si nous ne l'avons pas formalisée.

Pour être plus précis, il est possible de décrire la structure des matrices sur un anneau $\mathcal{R}$ comme une catégorie dont les objets sont les $\mathcal{R}^n$, pour n entier naturel, et dont les morphismes de $\mathcal{R}^m$ vers $\mathcal{R}^n$ sont interprétés par des matrices de taille $m \times n$. La composition des flèches est le produit matriciel. Cette catégorie est enrichie, c'est-à-dire que chaque ensemble de morphismes $\mathrm{Hom}(m,n)$ est muni d'une addition lui donnant une structure de groupe abélien, et que la composition est distributive par rapport à l'addition des flèches. Cette structure est généralement appelée *catégorie préadditive*. Un anneau en est un cas particulier, lorsque la catégorie est réduite à un objet.

Pour étendre notre méthodologie de preuve aux matrices rectangulaires, nous pourrions donc implémenter une nouvelle tactique de preuve automatique des identités sur les catégories préadditives, prenant en charge le type des matrices indexé par leur taille. Cependant, une autre approche est possible, en établissant un théorème de détypage [D. Pous, 2012]. Schématiquement, cette approche consiste à définir une fonction « oubliant » l'annotation de taille et à prouver que si en appliquant cette fonction à une expression bien typée on obtient une identité prouvable à partir des propriétés d'anneau, alors l'identité était prouvable dans la catégorie préadditive de départ. En d'autres termes, on construit un foncteur fidèle de la catégorie de départ vers une catégorie à un objet.

L'intérêt dans notre contexte est qu'il suffit d'appliquer le théorème de détypage pour pouvoir réutiliser ensuite l'essentiel de la mécanique de la tactique `non_commutative_ring`. En particulier, il n'est pas nécessaire de réimplémenter une fonction de normalisation travaillant sur des structures efficaces (comme des représentations de Hörner creuses de polynômes [B. Grégoire et A. Mahboubi, 2005]). Même si la preuve du théorème de détypage sur les catégories préadditives utilise une fonction de normalisation, seule son existence et ses propriétés importent, il y a donc tout intérêt à la définir de façon à garder des preuves simples, et non de rendre les calculs efficaces.

Dans le développement formel accompagnant [D. Pous, 2012], la seule pièce manquante est la preuve de la complétude de la fonction de normalisation naïve, qui est nécessaire pour établir le théorème de détypage, contrairement au développement de tactiques réflexives comme `ring` ou `non_commutative_ring` qui ne demande qu'une preuve de correction des fonctions de normalisation sous-jacentes. Nous prévoyons de réaliser cette preuve afin de fournir une version formellement vérifiée du produit de Strassen pour des matrices rectangulaires.

3.3 IMPLÉMENTATION EFFECTIVE

Nous allons maintenant appliquer les techniques du chapitre 2 pour obtenir une implémentation effective de l'algorithme de Strassen, c'est-à-dire qui pour des valeurs concrètes de ses entrées calcule un résultat en un temps raisonnable. Mais commençons par comprendre pourquoi l'algorithme décrit à la section précédente ne remplit pas ces critères. Pour ce faire, nous allons étudier l'encodage interne des matrices de SSReflect.

3.3.1 Représentation des matrices dans SSReflect

La notation `'M[R]_(m,n)` que nous avons utilisée précédemment cache en fait le type inductif suivant :

```
Inductive matrix R m n := Matrix of {ffun 'I_m * 'I_n -> R}.
```

Ainsi, les matrices sont définies comme des fonctions finies (`ffun`) sur des ordinaux. Pour un entier `n`, un ordinal de type `'I_n` est simplement composé d'un entier et d'une preuve qu'il est majoré par `n` :

```
Inductive ordinal (n : nat) : predArgType := Ordinal m of m < n.
```

Indexer les matrices par des ordinaux permet de faire en sorte que le bon typage exclue tout débordement (i.e. accès en dehors de la matrice). Les fonctions finies, elles, sont définies en interne à partir de leur graphe :

```
Variables (aT : finType) (rT : Type).
Inductive finfun_type : predArgType := Finfun of #|aT|.-tuple rT.
```

La notation `n.-tuple rT` désigne un `n`-uplet d'éléments du type `rT`. Ces `n`-uplets sont définis à leur tour comme des listes accompagnées d'une preuve que leur longueur est `n` :

```
Variables (n : nat) (T : Type).
Structure tuple_of : Type :=
  Tuple {tval :> seq T; _ : size tval == n}.
```

Le symbole `:>` déclare que la projection `tval` est une coercion permettant d'utiliser de manière transparente un `n`-uplet comme une liste.

Remarquons que la preuve sur la longueur utilise l'égalité booléenne. L'intérêt en est que SSReflect fournit un traitement générique des types de la forme `{x : T | p x}` pour un `p` de type `T -> bool`, dont les `n`-uplets sont alors une instance. En particulier, on dispose alors d'une preuve de l'injectivité de `tval`, qui repose sur la propriété suivante, dite d'*unicité des preuves d'égalité* sur les booléens :

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

Cette propriété est prouvable dans le Calcul des Constructions Inductives. On obtient alors une forme d'extensionnalité : deux `n`-uplets sont égaux si et seulement si leurs listes sous-jacentes énumèrent les mêmes éléments (dans le même ordre). Ceci est transféré aux fonctions finies et aux matrices. Ces définitions en cascade permettent ainsi de concilier l'expression de contraintes de taille par typage et l'extensionnalité, sans ajouter d'axiome au système logique.

Mais cette approche a un coût du point de vue opérationnel. Pour le comprendre, regardons comment se passe l'évaluation du terme `M i j`, pour

une matrice `M : 'M[R]_(m,n)` et des indices `i : 'I_m` et `j : 'I_n`. C'est la fonction finie `f : {ffun 'I_m * 'I_n -> R}` sous-jacente à `M` qui est évaluée. Pour cela, est d'abord recherché l'indice de `(i,j)` dans l'énumération canonique du type fini `'I_m * 'I_n`. Cette opération induit $\mathcal{O}(mn)$ comparaisons d'éléments de ce type, chacune ayant un coût $\mathcal{O}(m+n)$. Ensuite, l'indice obtenu permet de retrouver dans le graphe de `f` la valeur de `f (i,j)` par un accès de coût $\mathcal{O}(mn)$. En résumé, l'accès à un élément dans une matrice de taille $m \times n$ a une complexité en $\mathcal{O}(mn(m+n))$.

Bien sûr, cette complexité pourrait être améliorée en spécialisant la représentation des fonctions finies pour être plus efficace. Mais cela reviendrait à casser une barrière d'abstraction (ici, l'encodage des types finis et de leur produit cartésien), pourtant bien pratique comme nous l'avons montré pour hériter de propriétés comme l'extensionnalité, sans avoir à recourir à un axiome ou prouver à nouveau les propriétés pour la représentation spécialisée. Notre approche ne consiste donc pas à remplacer cette représentation des matrices, mais à la compléter par une représentation alternative, plus adaptée au calcul, dans l'esprit du chapitre 2.

3.3.2 Représentation adaptée au calcul

Nous choisissons de représenter les matrices par des listes de listes :

```
Variable A : Type.
Definition seqmatrix := seq (seq A).
```

L'opérateur `nth` fourni par SSReflect permet d'accéder à un élément dans une liste par son indice. En cas de débordement, `nth` renvoie une valeur par défaut prise en argument. Ainsi, étant donnée une matrice `M` de type `seqmatrix A`, l'accès à l'élément `(i,j)` s'écrit `nth x0 (nth [::] M i) j`, où `[::]` désigne la liste vide et `x0` une valeur par défaut dans `A`.

Le coût d'accès à un élément dans une matrice de taille $m \times n$ utilisant cette représentation est donc $\mathcal{O}(m+n)$, ce qui constitue une nette amélioration par rapport à la représentation adaptée aux preuves. D'autre part, aucune preuve n'est manipulée lors du calcul, ce qui était inévitable avec des indices exprimés par des ordinaux (contenant une preuve de majoration). On évite ainsi des constructions d'objets et des projections, qui sont inutiles au sens où elles servent de garanties statiques mais n'apparaissent pas dans le résultat final de l'évaluation.

Il peut paraître décevant de ne pas avoir un accès en temps constant aux coefficients d'une matrice, au regard de la représentation usuelle à base de tableaux dans les langages qui en disposent. Cependant, nous ne nous attachons pas tant à l'efficacité d'un accès pris isolément qu'à celle des opérations de plus haut niveau sur les matrices comme l'addition, le produit, la transposition, dont nous verrons que la complexité n'est pas dégradée par l'utilisation de listes au lieu de tableaux.

Afin de lier notre représentation des matrices à celle adaptée aux preuves, nous suivons la méthodologie mise en place au chapitre 2 et définissons une fonction de représentation `mx_of_seqmx` des listes de listes vers les matrices (son type est : `forall m n : nat, seqmatrix -> option 'M_(m, n)`). Puis, à partir de cette fonction nous construisons une relation de raffinement :

```
Definition Rseqmx {m n} := ofun_hrel (mx_of_seqmx m n).
```

Nous avons défini la fonction `ofun_hrel` à la section 2.1.

Pour définir l'addition, on peut recourir à un opérateur `zipwith` qui, étant données deux listes, en construit une troisième en appliquant une opération binaire aux éléments successifs des deux listes passées en entrée :

La construction **if** *est ici une expression concise du filtrage.*

```
Variables (T1 T2 R : Type) (f : T1 -> T2 -> R).

Fixpoint zipwith (s1 : seq T1) (s2 : seq T2) :=
  if s1 is x1 :: s1' then
    if s2 is x2 :: s2' then f x1 x2 :: zipwith s1' s2' else [::]
  else [::].
```

Cet opérateur s'étend immédiatement à nos matrices :

```
Definition zipwithseqmx (f : A -> A -> A) (M N : seqmatrix) :=
  zipwith (zipwith f) M N.
```

Pour établir la correction de `zipwithseqmx`, nous définissons une fonction `zipwithmx` et prouvons un lemme reliant les deux :

```
Definition zipwithmx m n (f : A -> A -> A) (M N : 'M[A]_(m,n)) :=
  \matrix_(i,j) f (M i j) (N i j).

Instance refines_zipwithseqmx m n :
  param (eq ==> Rseqmx ==> Rseqmx ==> Rseqmx) (@zipwithmx m n) ↙
      ↳ (@zipwithseqmx A).
```

Nous implémentons alors génériquement l'addition sur les matrices en paramétrant par une opération d'addition sur les coefficients :

```
Context `{add A}.
Instance addseqmx : add seqmatrix := zipwithseqmx +%C.
```

Le lemme de correction de l'addition est une conséquence immédiate de celui de `zipwithseqmx`. Suivant la méthodologie du chapitre 2, nous le prouvons dans le cas particulier où A est un $\mathbb{Z}$-module en utilisant l'addition définie par cette structure (un lemme de paramétricité permettra le transport de cette preuve à d'autres instances de `A`) :

```
Variable A : zmodType.
Instance add_A : Op.add A := +%R.
Instance refines_addseqmx m n :
  param (Rseqmx ==> Rseqmx ==> Rseqmx)
    (+%R : 'M[A]_(m,n) -> 'M[A]_(m,n) -> 'M[A]_(m, n))
    (+%C : seqmatrix A -> seqmatrix A -> seqmatrix A).
```

L'opérateur `zipwith` permet d'écrire également la transposition des matrices :

La fonction `nseq` *appliquée à un entier* `n` *et un élément* `a` *construit une liste de* `n` *copies de* `a`.

```
Definition trseqmx (M : seqmatrix) : seqmatrix :=
  foldr (zipwith cons) (nseq (size (nth [::] M 0)) [::]) M.
```

Cette fonction de transposition commence par initialiser une liste de listes vides, dont la quantité correspond au nombre de colonnes de la matrice d'origine. Puis, pour chaque ligne de `M`, chaque élément de cette ligne est ajouté en tête d'une ligne du résultat en cours de construction. Ainsi, si la ligne de `M` courante est d'indice `i`, l'élément d'indice `j` (c'est-à-dire appartenant à la colonne `j`) se retrouvera sur la ligne `j` du résultat. Le parcours de `M` s'effectue de droite à gauche, ce qui garantit que l'insertion des éléments en

tête donne l'ordre voulu. La correction de `trseqmx` s'établit par rapport à la fonction de transposition `trmx` définie par SSReflect :

```
Instance refines_trseqmx m n :
  param (Rseqmx ==> Rseqmx) (@trmx A m.+1 n) (@trseqmx A).
```

Comme souvent, cette preuve de correction ne présente pas de difficulté particulière, sauf de trouver le bon invariant pour faire fonctionner la preuve par induction structurelle sur la liste passée en argument. L'invariant que nous utilisons est le suivant :

```
forall s2 k l s1, k < size (foldr (zipwith cons) s1 s2) ->
 nth (x j i) (nth [::] (foldr (zipwith cons) s1 s2) k) l =
  if l < size s2 then nth (x j i) (nth [::] s2 l) k
  else nth (x j i) (nth [::] s1 k) (l - size s2).
```

En ce qui concerne le produit matriciel, pour l'algorithme naïf, c'est-à-dire suivant directement la définition, il faut veiller à ce que le coût linéaire de l'accès à un élément ne pénalise pas la complexité. Ainsi, pour obtenir le produit de deux matrices de taille $n \times n$ en $\mathcal{O}(n^3)$ opérations arithmétiques, une solution est de transposer une des deux matrices (la transposition se fait en $\mathcal{O}(n^2)$) :

```
Context `{zero A, add A, mul A}.
Definition mulseqmx (M N : seqmatrix) : seqmatrix :=
  let N' := trseqmx N in
  map (fun r => map (foldl2 (fun z x y => x * y + z) 0 r) N') M.
```

La fonction `foldl2` est un analogue de `foldl`, mais s'applique à deux listes et une fonction attendant trois arguments.

Le lemme de correction du produit est :

```
Variable (A : ringType).
Instance zero_A : zero A := 0%R.
Instance add_A : add A := +%R.
Instance mul_A : mul A := *%R.
Instance refines_mulseqmx m n p :
  param (Rseqmx ==> Rseqmx ==> Rseqmx)
        (mulmx : 'M[A]_(m, n) -> 'M[A]_(n, p) -> _)
        (Op.mulmx : seqmatrix A -> seqmatrix A -> seqmatrix A).
```

L'invariant clé exprime simplement les sommes partielles de produits de coefficients deux à deux :

```
forall s1 s2 (t : A), (foldl2 F t s1 s2) =
  (t + \sum_(0 <= k < minn (size s1) (size s2)) s1`_k * s2`_k).
```

Enfin, l'implémentation effective de l'algorithme de Strassen ne nécessite aucun effort, une fois l'infrastructure mise en place. En effet, grâce à un jeu sur les notations, l'algorithme tel que décrit à la section 3.1 peut être relu comme une version générique, pouvant être instanciée aussi bien par les matrices de SSReflect que par nos listes de listes, suivant la méthodologie du chapitre 2.

Reste donc uniquement à établir le lemme de paramétricité associé à la version générique de l'algorithme. Une difficulté surgit alors : la gestion automatique de l'induction. Nous découpons la fonction `Strassen` de la section 3.1 suivant ses branches : `Strassen_xI`, `Strassen_x0`. Puis nous prouvons le lemme de paramétricité associé à l'induction sur le type `positive` :

```
Instance param_elim_positive P P' (R : forall p, P p -> P' p -> ↙
    ↳ Prop) txI txI' txO txO' txH txH' :
  (forall p, getparam (R p ==> R (p~1)) (txI p) (txI' p)) ->
  (forall p, getparam (R p ==> R (p~0)) (txO p) (txO' p)) ->
  (getparam (R 1) txH txH') ->
  forall p, getparam (R p) (positive_rect P txI  txO  txH p) ↙
      ↳ (positive_rect P' txI' txO' txH' p).
```

Une fois ceci mis en place, la preuve du lemme de paramétricité associé à `Strassen` est presque entièrement automatique :

```
Variable (A : ringType) (mxC : nat -> nat -> Type).
Variable (RmxA : forall {m n}, 'M[A]_(m, n) -> mxC m n -> Prop).
Instance param_Strassen p :
  param (RmxA ==> RmxA ==> RmxA) (@Strassen (@matrix A) (p := ↙
      ↳ p)) (@Strassen mxC (p := p)).
```

3.4 PERFORMANCES

Nous évaluons les performances avec comme objectif principal de s'assurer que nos structures de données et notre environnement d'exécution n'induisent pas un surcoût prohibitif par rapport à la complexité théorique des algorithmes.

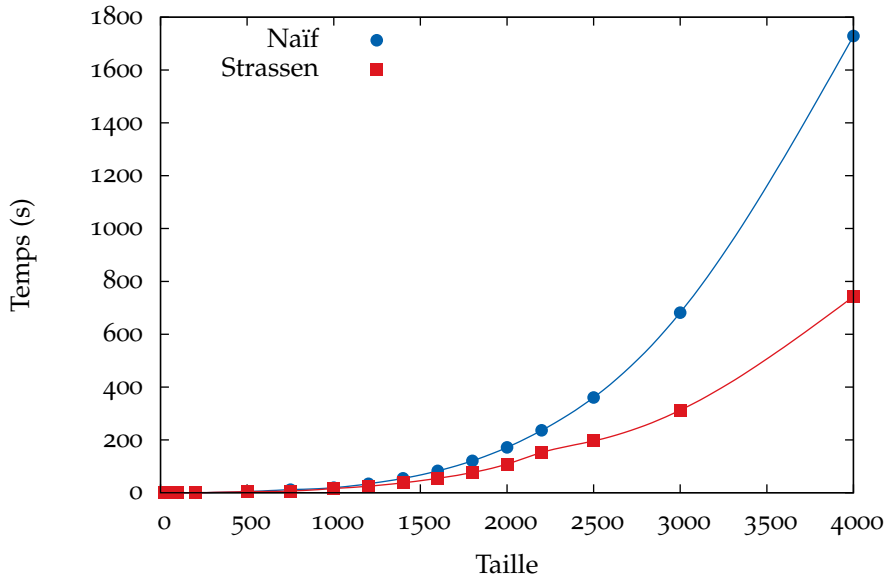


FIGURE 3.1 – Temps d'exécution des algorithmes de produit matriciel en fonction de la taille des matrices.

La figure 3.1 montre les temps d'exécution de nos implémentations formellement vérifiées du produit naïf et de l'algorithme de Strassen en fonction de la taille des matrices dont les coefficients sont des entiers machines. L'arithmétique modulaire permet d'espérer que la complexité arithmétique, où chaque opération sur ces entiers a un coût 1, soit un bon modèle. Ce ne serait pas le cas, par exemple, avec des entiers de taille arbitraire où le coût

des opérations dépendrait de la taille des coefficients. De fait, les courbes obtenues ont la forme théoriquement attendue, ce qui encourage à penser que le comportement de nos structures de données est raisonnable.

Taille	**Temps (s)**	
	Naïf	Strassen
500	4,35	4,09
1 000	19,01	15,71
2 000	171,52	108,46
4 000	1 728,81	742,59

TABLE 3.1 – Temps d'exécution des algorithmes de produit matriciel.

La table 3.1 confirme que l'exposant de complexité mesuré est proche de ce que donne la théorie : pour des matrices de taille n suffisamment grande, on retrouve un coefficient proche de 8 pour $\frac{T_{\text{naïf}}(2n)}{T_{\text{naïf}}(n)}$ et proche de 7 pour $\frac{T_{\text{Strassen}}(2n)}{T_{\text{Strassen}}(n)}$.

Bien sûr, des optimisations ultérieures sont possibles pour améliorer par exemple le coefficient dominant du temps d'exécution. Mais la possibilité de mener des calculs *à l'intérieur de* Coq (nos mesures sont effectuées en utilisant la normalisation par compilation en code natif décrite au chapitre 1) sur des matrices de tailles significatives, en gardant l'exposant de complexité attendu théoriquement nous paraît être une avancée significative par rapport à l'état de l'art.

RÉGLAGE DU SEUIL Pour déterminer la meilleure valeur du seuil k en dessous duquel le produit de deux matrices carrées revient sur l'algorithme naïf, nous procédons empiriquement.

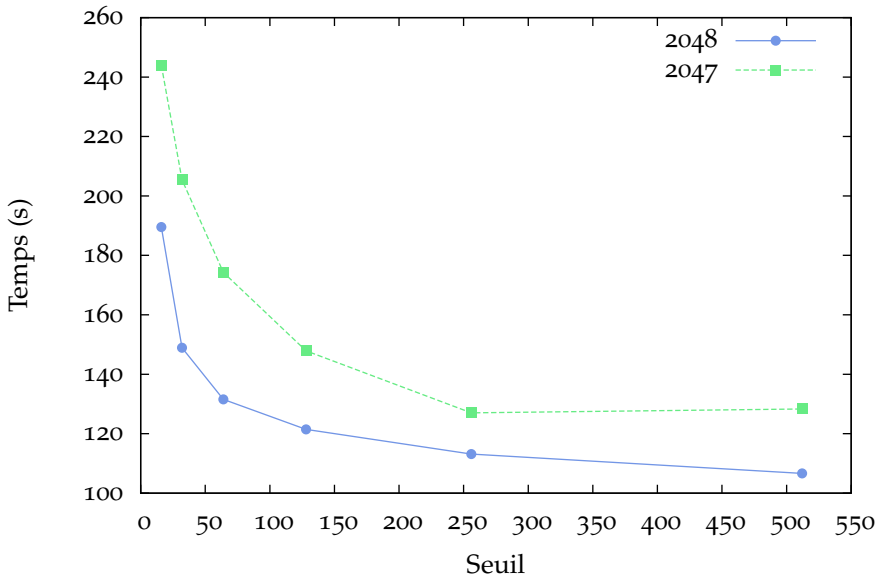


FIGURE 3.2 – Temps d'exécution du produit de deux matrices en fonction du seuil.

La figure 3.2 montre les temps d'exécution du produit de deux matrices carrées de taille n dans le pire ($n = 2^k - 1$) et le meilleur cas ($n = 2^k$). Nous fixons $k = 11$ et faisons varier le seuil (en le restreignant aux puissances de

2) afin d'observer les variations. Sur la machine que nous utilisons pour nos évaluations de performances, 256 semble être un seuil raisonnable, ce qui est du même ordre de grandeur que les valeurs indiquées par des expérimentations plus poussées pour des algorithmes non vérifiés formellement [S. Huss-Lederman et al., 1996].

Nous n'attachons pas une importance cruciale à déterminer la valeur optimale du seuil, mais il est néanmoins nécessaire de ne pas trop s'en éloigner. À titre d'exemple, enlever le test du seuil et utiliser l'algorithme de Strassen pour tous les appels récursifs jusqu'aux matrices de taille 1×1 rend l'algorithme moins efficace que son équivalent naïf pour toutes les tailles de matrices que nous avons observées (y compris 4000×4000) !

L'optimisation de la multiplication des matrices rectangulaires est un sujet difficile, des études ayant parfois des conclusions contradictoires [P. A. Knight, 1995 ; P. D'Alberto et A. Nicolau, 2009]. Nous le laissons donc pour un travail futur.

3.5 CONCLUSION

Nos contributions dans ce chapitre sont :

- Une preuve formelle de correction de l'algorithme de Strassen pour des matrices (carrées) de taille quelconque.
- Une implémentation calculatoire des opérateurs de la bibliothèque de matrices de SSReflect : construction, addition, multiplication naïve, transposée, opérations par blocs, multiplication par un scalaire, test d'égalité.
- L'utilisation de ces opérateurs pour fournir une version de l'algorithme de Strassen efficace en pratique à l'intérieur de Coq.

Une version préliminaire de ces contributions est décrite dans [M. Dénès et al., 2012].

Nous pensons que ce travail montre que la méthodologie et le développement décrits au chapitre 2 sont utilisables en pratique, et passent à l'échelle. Mais l'algorithme étudié ici a aussi son intérêt propre. Nous avons cité en introduction de nombreux problèmes en algèbre linéaire dont la complexité est réduite à celle de la multiplication de matrices. Il est possible d'aller plus loin car de nombreux problèmes, de complexité plus grande, peuvent être *vérifiés* par un produit matriciel. Citons notamment la diagonalisation de matrices ou plus généralement le calcul de leur forme normale de Jordan, ou encore le calcul de la forme normale de Smith (ces formes canoniques sont évoquées au chapitre 5). Tous ces problèmes s'expriment comme la décomposition d'une matrice en un produit. Il est alors possible d'utiliser un logiciel de calcul formel externe comme un oracle fournissant les facteurs matriciels, puis de vérifier le résultat obtenu en calculant leur produit dans Coq. Le seul inconvénient est alors que la *complétude* de la méthode n'est pas garantie (seule la *correction* de chaque résultat l'est).

En outre, la capacité de prendre en charge des matrices de taille réaliste (4000×4000 par exemple) ouvre la voie à une meilleure intégration, au sein d'assistants à la preuve, de fonctionnalités habituellement trouvées dans les logiciels de calcul formel.

En ce qui concerne la poursuite de ce travail, nous avons déjà indiqué des pistes pour étendre le résultat aux matrices rectangulaires. Il serait égale-

ment intéressant de tester des représentations de matrices plus adaptées à la structure de l'algorithme, par exemple un type inductif construisant une matrice à partir de quatre sous-blocs, ce qui simplifierait les opérations de découpage mises en jeu.

4 ÉLIMINATION GAUSSIENNE ET DÉCOMPOSITION LU

La méthode du pivot de Gauss est un algorithme très ancien pour la résolution de systèmes linéaires. On en trouve une description dans le chapitre 8 de l'ouvrage chinois « Les neuf chapitres de l'art mathématique » [K. Chemla et G. Shuchun, 2004], écrit entre le II^e et le I^er siècle av. J.C.

Par la suite, elle semble avoir été redécouverte au XVII^e siècle en Europe, où elle est parfois éclipsée par des développements de la théorie du déterminant et en particulier les formules de Cramer. Cependant, l'utilisation de ces formules en pratique est très inefficace : pour un système de n équations à n inconnues, $\mathcal{O}(n!)$ opérations sont nécessaires. Tant que les systèmes restent petits, ce coût est acceptable, mais la mécanisation du calcul entraîne vite un regain d'intérêt pour la méthode du pivot, à laquelle de nombreux mathématiciens contribuent en l'affinant (stratégie de choix du pivot), en la reformulant (algèbre matricielle) ou en l'appliquant à de nouveaux contextes. C'est ainsi que le nom de Gauss, qui fait partie de ces contributeurs, va être attaché à cette méthode bien qu'il n'en soit pas l'inventeur.

ALGORITHME Pour rappel, la méthode consiste à résoudre un système de la forme $AX = B$ pour $A \in \mathcal{K}^{m\times n}$ et $B \in \mathcal{K}^{m\times 1}$ donnés, en le réduisant à une forme échelonnée au moyen d'opérations élémentaires sur les lignes. Ces opérations sont de trois types :

1. Échange de deux lignes
2. Multiplication d'une ligne par une constante de $\mathcal{K}$ non nulle
3. Ajout à une ligne du produit d'une autre ligne par une constante

Une propriété importante de ces opérations est qu'elles transforment un système en un système équivalent (c'est-à-dire admettant les mêmes solutions). Ainsi, considérons le système suivant :

$$\begin{array}{ccccccc} a_{1,1}x_1 & + & \cdots & + & a_{1,n}x_n & = & b_1 \\ \vdots & & & & & & \vdots \\ a_{m,1}x_1 & + & \cdots & + & a_{m,n}x_n & = & b_m \end{array}$$

Pour que la première étape d'élimination soit possible, il faut que le *pivot* $a_{1,1}$ soit non nul. Si ce n'est pas le cas, on cherche un élément non nul sur la première colonne. S'il n'en existe pas, on passe à l'étape suivante. Sinon, on ramène cet élément à la position $(1,1)$ par une permutation de lignes. Cette stratégie est appelée *pivot partiel*. Une alternative est de chercher le pivot dans toute la matrice et d'autoriser les permutations de colonnes (c'est-à-dire de variables). On parle alors de *pivot total*.

Lorsqu'un pivot a été trouvé, on réalise les combinaisons de lignes $L_i \leftarrow L_i - \frac{a_{i,1}}{a_{1,1}}L_1$, $2 \le i \le m$ pour éliminer l'inconnue x_1 des lignes 2 à m. On obtient alors le système de la figure 4.1.

On itère ensuite le procédé sur le sous-système encadré de cette même figure. À la fin de l'algorithme, on obtient un système sous forme échelonnée qui est facilement résolu par *substitution arrière* des inconnues, c'est-à-dire

$$
\begin{array}{ccccccc}
a_{1,1}x_1 & + & a_{1,2}x_2 & + & \cdots & = & b_1 \\
0 & + & \left(a_{2,2} - \frac{a_{2,1}}{a_{1,1}} \times a_{1,2}\right)x_2 & + & \cdots & = & b_2 - \frac{a_{2,1}}{a_{1,1}} \times b_1 \\
\vdots & & & & & & \vdots \\
0 & + & \left(a_{m,2} - \frac{a_{m,1}}{a_{1,1}} \times a_{1,2}\right)x_2 & + & \cdots & = & b_n - \frac{a_{m,1}}{a_{1,1}} \times b_1
\end{array}
$$

Figure 4.1 – Système d'équations linéaires obtenu après une étape d'élimination gaussienne.

qu'on résout la dernière équation, on propage le résultat à l'équation précédente que l'on résout à son tour, jusqu'à remonter à la première.

COMPLEXITÉ Pour calculer la complexité de la méthode du pivot de Gauss décrite ci-dessus, nous affectons un coût 1 aux opérations du corps des coefficients. En calcul exact, ce modèle est réaliste dans le cas où on travaille sur des matrices à coefficients dans un corps fini. Cependant dans d'autres contextes, par exemple si les coefficients sont des fractions d'entiers ou de polynômes, la taille des coefficients joue un rôle dominant, et il faut alors en tenir compte.

Soit A une matrice de taille $m \times n$ et de rang r. Nous ne prenons pas en compte des opérations de comparaison à 0 ou de permutation de lignes. Une étape d'élimination telle que décrit plus haut appliquée à A met en jeu $(m-1) \times (n-1)$ multiplications et soustractions, ainsi que $m-1$ divisions.

On sait qu'au cours de la procédure globale de triangularisation, au plus r pivots non nuls seront trouvés. En effet, à chaque fois qu'un tel pivot est trouvé, une étape d'élimination est effectuée et la ligne courante devient linéairement indépendante des suivantes. De plus, du point de vue du nombre d'opérations arithmétiques, le pire cas est lorsque r pivots sont trouvés au cours des r premières étapes, car la taille des matrices en jeu est alors maximale.

Remarquons que dans le cas du pivot partiel, le nombre de pivots non nuls rencontrés au cours de l'algorithme peut être strictement inférieur au rang de la matrice donnée en entrée. Considérons par exemple la matrice suivante :

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

La première étape de l'algorithme ne va apporter aucun changement visible à la matrice, car il y a déjà un pivot non nul en position supérieure gauche et les termes à soustraire sont nuls. La seconde étape va donc s'appliquer à la sous-matrice :

$$\begin{bmatrix} 0 & 1 \end{bmatrix}$$

Cette matrice n'a pas de pivot non nul sur la première colonne. L'algorithme se termine alors, et un seul pivot non nul a été rencontré alors que la matrice de départ a un rang égal à 2.

Revenons au calcul de la complexité de l'algorithme. En notant $T(m,n,r)$ le coût de la triangularisation d'une matrice $m \times n$ de rang r dans le pire cas, on a :

$$\begin{aligned} T(m,n,r) &= 2\sum_{k=1}^{r}(m-k)(n-k) + \sum_{k=1}^{r}(m-k) \\ &= 2[mnr - (m+n)\sum_{k=1}^{r}k + \sum_{k=1}^{r}k^2] \\ &= 2mnr - (m+n)r(r+1) + \frac{1}{3}r(r+1)(2r+1) \\ &\leq 2mnr - (m+n)r^2 + \frac{2}{3}r^3 \end{aligned}$$

Dans le cas d'une matrice carrée de taille n inversible, la complexité est donc $\mathcal{O}(n^3)$.

DÉCOMPOSITIONS MATRICIELLES L'algorithme que nous avons décrit peut se réinterpréter de façon matricielle. Au système $AX = B$ est associée la *matrice augmentée* $[A|B]$, que le processus réduit à une forme échelonnée. Plus généralement, l'élimination gaussienne peut être appliquée à une matrice sur un corps pour en déterminer le rang ou si elle est carrée, son déterminant ou son inverse. Ainsi, outre une bien meilleure efficacité que les formules de Cramer, c'est le caractère de couteau suisse de cette méthode qui justifie son importance.

En outre, l'interprétation matricielle peut être poussée plus loin. En effet, chaque opération élémentaire sur les lignes peut être vue comme le produit à gauche par une matrice d'opération élémentaire. Ces matrices ont de plus la propriété d'être inversibles. L'élimination gaussienne peut donc se voir comme un algorithme réalisant une décomposition matricielle :

$$A = PLU \quad \text{où} \begin{cases} P \in \mathcal{K}^{m\times m} \text{ est une matrice de permutation} \\ L \in \mathcal{K}^{m\times m} \text{ est une matrice unitriangulaire inférieure} \\ U \in \mathcal{K}^{m\times n} \text{ est une matrice triangulaire supérieure} \end{cases}$$

Il existe d'autres variantes de l'élimination gaussienne que celle que nous avons décrite. Notamment, il est possible d'autoriser les opérations élémentaires sur les colonnes, qui correspondent alors à un produit à droite par des matrices associées à ces opérations. Dans ces conditions, toute matrice $A \in \mathcal{K}^{m\times n}$ peut se décomposer comme suit :

$$A = PLI_rUQ \quad \text{où} \begin{cases} P \in \mathcal{K}^{m\times m} \text{ est une matrice de permutation} \\ L \in \mathcal{K}^{m\times m} \text{ est une matrice unitriangulaire inférieure} \\ r \text{ est le rang de } A \\ U \in \mathcal{K}^{n\times n} \text{ est une matrice triangulaire supérieure} \\ Q \in \mathcal{K}^{n\times n} \text{ est une matrice de permutation} \end{cases}$$

Cette décomposition a été formalisée dans la librairie SSReflect. En fait, elle y est même à la base des principales définitions en algèbre matricielle, comme le rang, le noyau ou les sous-espaces vectoriels représentés par des matrices. De nombreux résultats, comme le théorème de la base incomplète, sont encapsulés dans cette procédure algorithmique. Cependant, nous prenons encore une fois ici un point de vue calculatoire, qui incite à se restreindre à un pivot partiel et des opérations sur les lignes uniquement, car elles sont moins coûteuses.

Dans ce chapitre, nous étudierons tout d'abord deux variations autour de l'élimination gaussienne : un algorithme de calcul du rang (section 4.1), puis

la décomposition PLU (section 4.2) et enfin nous verrons comment réutiliser les résultats du chapitre 3 pour améliorer la complexité de l'inversion des matrices triangulaires (section 4.3).

4.1 CALCUL DU RANG

[10] Cet algorithme nous a été suggéré par Thierry Coquand.

Nous allons maintenant présenter un algorithme[10] qui calcule le rang d'une matrice à coefficients dans un corps.

DESCRIPTION DE L'ALGORITHME Soit $A = (a_{i,j})$ une matrice de taille $m \times n$ sur un corps $\mathcal{K}$. Une étape d'élimination consiste à trouver un pivot non nul sur la première colonne de A. S'il n'y en a pas, la première colonne peut être supprimée sans changer le rang. Sinon, il existe un indice i tel que $a_{i,1} \neq 0$. Par des opérations élémentaires sur les lignes (préservant le rang), A peut être ramenée à la matrice B suivante :

$$B = \begin{bmatrix} 0 & a_{1,2} - \frac{a_{1,1}\times a_{i,2}}{a_{i,1}} & \cdots & a_{1,n} - \frac{a_{1,1}\times a_{i,n}}{a_{i,1}} \\ 0 & \vdots & & \vdots \\ a_{i,1} & a_{i,2} & \cdots & a_{i,n} \\ 0 & \vdots & & \vdots \\ 0 & a_{n,2} - \frac{a_{n,1}\times a_{i,2}}{a_{i,1}} & \cdots & a_{n,n} - \frac{a_{n,1}\times a_{i,n}}{a_{i,1}} \end{bmatrix}$$

$$= \begin{bmatrix} 0 & & \\ \vdots & R_1 & \\ 0 & & \\ a_{i,1} & \cdots & a_{i,n} \\ 0 & & \\ \vdots & R_2 & \\ 0 & & \end{bmatrix}$$

Posons maintenant $R = \begin{pmatrix} R_1 \\ R_2 \end{pmatrix}$, puisque $a_{i,1} \neq 0$, on a $\mathrm{rang}(A) = \mathrm{rang}(B) = 1 + \mathrm{rang}(R)$. Donc le rang courant est incrémenté et l'algorithme est appliqué récursivement à R.

IMPLÉMENTATION Comme dans les chapitres précédents, nous définissons une implémentation générique de cet algorithme en réutilisant les notations de la bibliothèque SSReflect, mais en les attachant à des classes de types opérationnelles.

Les opérateurs intervenant dans cet algorithme sont quantifiés sur les types suivants :

```
Variable A : Type.
Variable mxA : nat -> nat -> Type.
Variable ordA : nat -> Type.
```

La variable `A` est le type des coefficients, tandis que `mxA` est celui des matrices sur `A`. Enfin, `ordA` est le type des indices. Dans l'instance orientée preuve, il deviendra le type `ordinal` évoqué au chapitre 3. Dans la version orientée calcul, `ordA` sera simplement instancié à `fun _ => nat`.

Dans ce contexte, nous définissons la fonction `elim_rank` suivante :

```
Fixpoint elim_rank {m n : nat} : 'M_(m,n) -> nat :=
  match n return 'M_(m,n) -> nat with
  | p.+1 => fun (M : 'M_(m, 1 + p)) =>
    if find_pivot M is Some k then
      let a := M.(k,0) in
      let u := rsubmx (row k M) in
      let R := row' k M in
      let v := a^-1 *: lsubmx R in
      let R := rsubmx R - v *m u in
      1 + elim_rank R
    else elim_rank (rsubmx M)
  | _ => fun _ => 0
  end.
```

La notation `a *: M` *désigne le produit externe d'une matrice* `M` *par un scalaire* `a`.

L'expression `row' k M` désigne la matrice `M` à laquelle a été enlevée la ligne d'indice `k` (l'opérateur `row'` fait partie de la théorie des matrices fournie par SSReflect). Dualement, `row k M` dénote la ligne d'indice `k` de `M`. Les fonctions `lsubmx` et `rsubmx` réalisent un découpage gauche-droite de la matrice en argument, à partir de l'expression des tailles dans le type, de manière similaire aux découpages déjà rencontrés au chapitre 3.

La seule notation qui ne soit pas exactement identique à son équivalent dans SSReflect est l'accès à un élément `M.(i,j)`. En effet, à cause d'une limitation technique du système de coercions de Coq, nous n'avons pas pu réutiliser telle quelle la notation `M i j` pour noter l'accès à l'élément à la ligne `i` et la colonne `j` de `M`.

Enfin, la fonction `find_pivot` représente la stratégie de recherche de pivot (non nul) dans la première colonne d'une matrice. Afin d'obtenir une preuve de correction indépendante du choix d'une stratégie particulière, nous supposons simplement que cette fonction est correcte, c'est-à-dire que `find_pivot M` renvoie `Some i` avec un `i` tel que le coefficient `M.(i,0)` est non nul, ou `None` s'il n'y a aucun élément non nul sur la première colonne de `M`. La bibliothèque SSReflect fourni un moyen d'exprimer cette spécification de façon concise :

```
Variable find_pivot : forall m n, 'M_(m,n.+1) -> option 'I_m.

Hypothesis find_pivotP : forall m n (M : 'M_(m, n.+1)),
  pick_spec [pred k | M k 0 != 0] (find_pivot M).
```

La notation `[pred k | P k]` permet de voir une expression comme un prédicat booléen, en indiquant sur quelle variable quantifier. Le prédicat inductif `pick_spec` est défini par :

```
Inductive pick_spec : option T -> Type :=
  | Pick x of P x         : pick_spec (Some x)
  | Nopick of P =1 xpred0 : pick_spec None.
```

L'énoncé `P =1 xpred0` *exprime que le prédicat booléen* `P` *est insatisfiable.*

En ne supposant rien de plus sur `find_pivot`, nous représentons une forme de non-déterminisme qui disparaîtra lors du raffinement vers l'implémentation effective.

Sous ces hypothèses, et en instanciant tous nos types génériques à leur version orientée preuve, nous prouvons la correction de `elim_rank` exprimée en la reliant à la fonction `\rank` de SSReflect [G. Gonthier, 2011] :

```
Lemma elim_rankP m n (M : 'M[K]_(m,n)) : elim_rank M = \rank M.
```

La preuve se fait par induction sur `n`. Le cas de base est immédiat, en utilisant les lemmes `thinmx0` et `mxrank0` de la bibliothèque, qui assurent respectivement qu'une matrice de 0 colonnes est nulle, et qu'une matrice nulle a un rang égal à 0.

La suite de l'induction est plus intéressante. Une analyse par cas sur `find_pivotP` permet d'isoler deux branches de la preuve. Si aucun pivot n'a été trouvé (`find_pivot M` s'évalue à `None`), la première colonne est nulle et nous prouvons le lemme suivant pour conclure :

```
Lemma rank_row0mx (m n p : nat) (M : 'M[F]_(m,n)) :
  \rank (row_mx (0: 'M[F]_(m,p)) M) = \rank M.
```

Pour traiter le cas où un pivot est trouvé, nous prouvons un lemme exprimant que si $a \neq 0$, le rang d'une matrice de la forme $\begin{bmatrix} a & A' \\ 0 & A \end{bmatrix}$ est $1 + \text{rang}(A)$:

La notation `a%:M` *désigne une matrice scalaire avec des coefficients* `a` *sur la diagonale.*

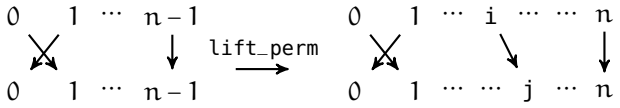
```
Lemma rank_block0dl m n a Aur (Adr : 'M[F]_(m,n)) : a != 0 ->
  \rank (block_mx (a%:M : 'M_1) Aur 0 Adr) = 1 + \rank Adr
```

Mais une petite difficulté se présente alors : dans notre algorithme, le pivot ne se trouve pas nécessairement sur la première ligne. Dans la preuve, nous réexprimons la matrice `M` à laquelle a été enlevée la ligne `k` (`row' k M`) comme une permutation de `M` dont la première ligne a été supprimée. Il nous faut donc construire cette permutation. Une simple transposition entre la première ligne et celle d'indice `k` ne convient pas car le reste de la matrice ne sera pas identique après suppression des lignes.

Heureusement, la bibliothèque SSReflect fournit une fonction `lift_perm` qui, étant donnés une permutation `s` des ordinaux `'I_n` $= \{0, \ldots, n-1\}$ et deux ordinaux `i` et `j` de type `'I_n.+1`, construit une nouvelle permutation `t = lift_perm s i j` sur `'I_n.+1` de la manière suivante :

$$t(k) = \begin{cases} s(k), \text{ si } k < i \\ j, \text{ si } k = i \\ s(k-1)+1, \text{ si } k > i \end{cases}$$

Graphiquement, cela donne sur un exemple :



Il suffit alors d'observer que la permutation `lift_perm 1%g 0 k`, où `1%g` désigne la permutation identité, est bien celle que nous souhaitions appliquer aux lignes de la matrice en entrée de notre algorithme. Nous établissons le lemme suivant, qui permet de conclure la preuve après quelques manipulations algébriques :

```
Lemma row'_row_perm m n M k : row' k M =
  dsubmx (row_perm (lift_perm 0 k 1%g) M : 'M_(1 + m, n)).
```

LEMME DE PARAMÉTRICITÉ Ayant ainsi établi la correction de l'instance orientée preuve de notre algorithme, nous continuons selon la méthodologie présentée au chapitre 2, et prouvons le lemme de paramétricité associé à `elim_rank`.

Pour ce faire, nous commençons par définir, sur les structures orientées preuve, une fonction `find_pivot` qui implémente une stratégie de recherche de pivot. Nous choisissons une simple recherche linéaire, en parcourant la première colonne de la matrice en commençant par la ligne d'indice 0 :

```
Fixpoint find_pivot_rec k {m n} (M : 'M[F]_(m.+1,n.+1)) :=
  if k is k'.+1 return option 'I_m.+1 then
    if M (inord (m - k)) 0 != 0 then Some (inord (m - k))
    else find_pivot_rec k' M
  else None.

Definition find_pivot m n :=
  if m is m'.+1 return 'M_(m,n.+1) -> option 'I_m then
    find_pivot_rec m
  else fun _ => None.
```

L'opérateur `inord` injecte un entier `k` dans un ordinal `I_n.+1` si `k <= n`, en inférant `n` à partir du contexte. Si `n < k`, `k` est envoyé vers `0`.

Puis nous paramétrons le contexte de preuve par des implémentations des opérations sur les matrices, en supposant qu'elles se comportent bien par rapport aux versions orientées preuves. Pour l'opérateur `row` par exemple, nous écrivons :

```
Context (mxA : nat -> nat -> Type) (ordA : nat -> Type)
        (RmxA : forall {m n}, 'M[F]_(m, n) -> mxA m n -> Prop)
        (RordA : forall m, 'I_m -> ordA m -> Prop).

Context '{forall m n, param (RordA ==> RmxA ==> RmxA) ↙
    ↳ (@matrix.row F m n) (@row _ _ _ m n)}.
```

La fonction `matrix.row` est l'implémentation de SSReflect*, tandis que `row` est une variable, représentant une instance quelconque de l'opérateur.*

Nous procédons de même pour la fonction `find_pivot`, dont nous supposons qu'une implémentation sur notre type `mxA` reflète la même stratégie de recherche linéaire de pivot :

```
Context '{find_pivotC : forall m n : nat, mxA m n.+1 -> option ↙
    ↳ (ordA m)}.

Context '{forall m n, param (RmxA ==> ohrel RordA) (@find_pivot ↙
    ↳ m n) (@find_pivotC m n)}.
```

La fonction `ohrel` transporte une relation entre des types `T` et `U` vers une relation entre `option T` et `option U`.

Une fois mis en place ce paramétrage, nous pouvons prouver le lemme de paramétricité associé à `elim_rank` :

```
Instance param_elim_rank m n :
   param (RmxA ==> Logic.eq)
         (elim_rank F (matrix F) ordinal find_pivot m n)
         (elim_rank F mxA ordA find_pivotC m n).
```

4.2 DÉCOMPOSITION PLU

Nous allons maintenant enrichir le premier exemple de la section 4.1 pour obtenir, pour une matrice $A \in \mathcal{K}^{m \times n}$, une décomposition $PA = LU$[11] comme évoqué dans l'introduction de ce chapitre. En fait, SSReflect fournit déjà une fonction `cormen_lup` réalisant cette décomposition et basée sur un algorithme décrit dans [T. H. Cormen et al., 2001] :

[11] *Nous parlons ici de décomposition PLU, réservant le terme de décomposition LUP à la recherche de matrices L, U et P telles que $A = LUP$, ce qui est possible pour toute matrice A surjective.*

```
1  Fixpoint cormen_lup {n} :=
2    match n return let M := 'M[F]_n.+1 in M -> M * M * M with
3    | 0 => fun A => (1, 1, A)
4    | _.+1 => fun A =>
5      let k := odflt 0 [pick k | A k 0 != 0] in
6      let A1 : 'M_(1 + _) := xrow 0 k A in
7      let P1 : 'M_(1 + _) := tperm_mx 0 k in
8      let Schur := ((A k 0)^-1 *: dlsubmx A1) *m ursubmx A1 in
9      let: (P2, L2, U2) := cormen_lup (drsubmx A1 - Schur) in
10     let P := block_mx 1 0 0 P2 *m P1 in
11     let L := block_mx 1 0 ((A k 0)^-1 *: (P2 *m dlsubmx A1)) L2 ↙
           ↳ in
12     let U := block_mx (ulsubmx A1) (ursubmx A1) 0 U2 in
13     (P, L, U)
14   end.
```

La fonction `cormen_lup` prend en entrée une matrice carrée `A` de taille non nulle sur un corps `F` (ligne 2) et renvoie une matrice de permutation `P`, ainsi que deux matrices `L` et `U`, respectivement triangulaire inférieure et triangulaire supérieure. La décomposition obtenue est telle que `PA = LU`.

Le cas de base est celui d'une matrice de taille 1, auquel cas la fonction renvoie la matrice d'entrée inchangée, accompagnée d'une matrice de permutation `P` et d'une matrice `L` égales à l'identité (ligne 3). Pour une matrice `A` de taille supérieure, un élément non nul est recherché sur la première colonne (ligne 5, `odflt 0 t` se réduit vers `0` si `t` s'évalue à None et vers `a` si `t` s'évalue à `Some a`). Si un tel pivot est trouvé sur la ligne d'indice `k`, les lignes `0` et `k` de `A` sont permutées (opérateur `xrow`), et une matrice `P1` de permutation représentant cette transposition (fonction `tperm_mx`) est construite (lignes 6 et 7). Aux lignes 8 et 9, la fonction est appliquée récursivement au complément de Schur de la matrice `A` permutée[12] qui représente la sous-matrice encadrée de la figure 4.1 vue plus haut. Cet appel récursif renvoie des matrices `P2`, `L2` et `U2` qui représentent la décomposition du complément de Schur. Il ne reste plus qu'à composer les matrices de permutation (ligne 10), appliquer les transformations de la figure 4.1 aux lignes de la matrice triangulaire inférieure (ligne 11) et enfin recomposer la matrice triangulaire supérieure obtenue (ligne 12).

[12] *La terminologie « complément de Schur » est habituellement employée pour l'expression notée* `drsubmx A1 - Schur` *dans le code ci-dessus.*

Pour nos besoins, nous modifions cette fonction en remplaçant les matrices de permutations par des permutations proprement dites. Sans compliquer l'algorithme, ce changement nous permettra d'obtenir une implémentation plus efficace lorsque nous le raffinerons vers un programme exécutable.

Suivant toujours la méthodologie du chapitre 2, nous définissons une fonction générique implémentant cet algorithme, qui encore une fois ressemble de très près à `cormen_lup` :

```
Fixpoint Gauss_plu {m n} :=
  match m, n return 'M_(m.+1,n.+1) ->
      'S_m.+1 * 'M_(m.+1,m.+1) * 'M_(m.+1,n.+1) with
  | p.+1, _.+1 => fun (A : 'M_(1 + (1 + p), 1 + _)) =>
    let k := odflt 0 (find_pivot A) in
    let A1 : 'M_(1 + _, 1 + _) := xrow 0 k A in
    let P1 : 'S_(1 + (1 + p)) := tperm 0 k in
    let Schur := (A.(k,0)^-1 *: dlsubmx A1) *m ursubmx A1 in
    let: (P2, L2, U2) := cormen_lup (drsubmx A1 - Schur)%HC in
```

La fonction `lift0_perm` *transforme une permutation* `s : 'S_n` *en une permutation* `s' : 'S_n.+1` *telle que* $s'(0) = 0$ *et* $s'(i+1) = s(i)$.

```
    let P := (lift0_perm P2) * P1 in
    let pA1 := row_perm P2 (dlsubmx A1) in
    let L := block_mx 1%:M (const_mx 0) (A.(k,0)^-1 *: pA1) L2 in
    let U := block_mx (ulsubmx A1) (ursubmx A1) (const_mx 0) U2 in
    (P, L, U)
  | _, _ => fun A => (1, 1%:M, A)
  end.
```

Le principal lemme de correction exprime qu'en multipliant les matrices `L` et `U` obtenues en sortie, on obtient bien la matrice `A` donnée en entrée, dont les lignes ont été permutées selon `P` également obtenue en sortie :

```
Lemma Gauss_plu_correct n (A : 'M_n.+1) :
  let: (P, L, U) := Gauss_plu A in row_perm P A = L * U.
```

Il reste également à vérifier que les matrices `L` et `U` sont respectivement triangulaire inférieure unitaire et triangulaire supérieure : .

```
Lemma Gauss_plu_lower n (A : 'M_n.+1) (i j : 'I_n.+1) :
  i <= j -> (Gauss_plu A).1.2 i j = (i == j)%:R.

Lemma Gauss_plu_upper n A (i j : 'I_n.+1) :
  j < i -> (Gauss_plu A).2 i j = 0 :> F.
```

Pour un booléen `b`, `b%:R` *vaut* `1` *si* `b` *est vrai,* `0` *sinon*

Ces preuves sont adaptées à partir des preuves correspondantes sur la fonction `cormen_lup` fournies par la bibliothèque SSReflect.

PERMUTATIONS Par rapport à l'algorithme de la section 4.1, le raffinement de la décomposition PLU vers un programme exécutable nécessite en plus une implémentation effective des permutations. Celles-ci sont représentées dans SSReflect par une fonction finie et une preuve d'injectivité :

```
Inductive perm_type : predArgType :=
  Perm (pval : {ffun T -> T}) & injectiveb pval.
```

Plus précisément, nous nous intéressons aux permutations sur le type des ordinaux `'I_n`. Le type de ces permutations est noté `'S_n`. La performance des permutations n'étant pas critique dans les algorithmes que nous utilisons, nous choisissons de les raffiner simplement vers des fonctions de type `nat -> nat`, ce qui sera suffisant pour notre usage. Nous écrivons une fonction de conversion de `nat -> nat` vers les fonctions finies (`ffun`), que nous avons déjà rencontrées au chapitre 3 (section 3.3), puis vers les permutations, dont le type est noté `'S_n` :

```
Definition funperm := nat -> nat.

Definition finfun_of_funperm n f : {ffun ('I_n.+1 -> 'I_n.+1)} :=
  [ffun k => inord (f k)].

Definition perm_of_funperm n (f : funperm) : option 'S_n :=
  if n is n'.+1 return option 'S_n then
    insub (@finfun_of_funperm n' f)
  else Some 1%g.
```

Nous définissons alors une relation de raffinement à partir de la fonction `perm_of_funperm` :

```
Definition Rfunperm {n} := ofun_hrel (perm_of_funperm n).
```

À titre d'exemple, le lemme de correction de l'opérateur de composition des matrices s'écrit comme suit :

```
Lemma refines_funperm_comp n :
  param (@Rfunperm n ==> Rfunperm ==> Rfunperm) (@perm_mul _) ↙
      ↳ cperm_comp.
```

Cette dernière preuve repose de manière cruciale sur l'extensionnalité des fonctions sous-jacentes aux permutations, qui est prouvable sans ajouter d'axiome au Calcul des Constructions Inductives car il s'agit de fonctions finies.

4.3 INVERSION RAPIDE DES MATRICES TRIANGULAIRES

Comme le montre Strassen dans l'article où il introduit l'algorithme portant son nom [V. Strassen, 1969], celui-ci peut être réutilisé pour améliorer la complexité de l'inversion de matrices par rapport aux méthodes basées sur l'élimination gaussienne. Ainsi, la méthode qu'il y décrit permet d'inverser une matrice carrée de taille n en un nombre d'opérations arithmétiques $\mathcal{O}(n^\omega)$, où ω est l'exposant de la complexité de la multiplication matricielle.

Mais cette méthode ne s'applique qu'à des matrices fortement régulières (c'est-à-dire que toutes les divisions rencontrées dans l'algorithme doivent se faire par un élément non nul, ce qui n'est pas le cas pour toute matrice inversible). Des généralisations importantes de cette approche existent, dont l'algorithme de Bunch et Hopcroft [J. R. Bunch et J. E. Hopcroft, 1974] qui calcule une décomposition LUP pour toute matrice surjective. Il permet donc en particulier d'inverser toute matrice carrée non singulière.[13]

[13] *C'est-à-dire inversible.*

Nous étudions une sous-routine de cet algorithme, qui calcule l'inverse d'une matrice carrée triangulaire supérieure non singulière avec le même exposant de complexité que la multiplication de matrices. L'idée est d'exprimer l'inversion d'une telle matrice par blocs, en utilisant le produit rapide de matrices pour reconstruire le résultat global. L'identité de base est la suivante :

$$\begin{bmatrix} A_1 & A_2 \\ 0 & A_3 \end{bmatrix}^{-1} = \begin{bmatrix} {A_1}^{-1} & -{A_1}^{-1} A_2 {A_3}^{-1} \\ 0 & {A_3}^{-1} \end{bmatrix}$$

Ce qui se traduit formellement par le lemme suivant :

Le prédicat `unitmx` *exprime l'inversibilité d'une matrice. La fonction* `invmx`*, elle, calcule cet inverse.*

```
Lemma invmx_ublock m n (Aul : 'M[F]_m) Aur (Adr : 'M[F]_n) :
  block_mx Aul Aur 0 Adr \in unitmx ->
  invmx (block_mx Aul Aur 0 Adr) = block_mx (invmx Aul)
    (- invmx Aul *m Aur *m invmx Adr) 0 (invmx Adr).
```

La preuve de ce lemme se fait en multipliant la matrice d'origine par l'expression par blocs de son inverse, et en vérifiant que l'on obtient bien la matrice identité. Il faut pour cela montrer que si la matrice d'origine est inversible, les blocs situés sur sa diagonale le sont aussi. Nous le faisons en utilisant les propriétés du déterminant.

Cette identité nous donne un schéma récursif pour l'inversion des matrices triangulaires supérieures de taille paires. En effet, dans ce cas on peut prendre des blocs A_1, A_2 et A_3 de même taille, et ainsi garantir que A_1 et A_3 sont également triangulaires supérieures. Il suffit alors de leur appliquer récursivement la même fonction d'inversion. La traduction formelle de cette identité est la suivante :

Dans le cas impair, deux problèmes se posent. Le premier est que le bloc A_3 n'est plus carré et le calcul du produit ${A_1}^{-1}A_2{A_3}^{-1}$ requiert un algorithme de Strassen applicable aux matrices non carrées, dont nous avons mentionné au chapitre 3 que nous ne l'avons pas formellement vérifié.

L'autre problème, plus sérieux, est la condition de garde de Coq. En effet, pour s'assurer qu'il n'est pas possible de définir un point fixe non terminant (ce qui mettrait en danger la cohérence de la logique), un critère syntaxique restreint les appels récursifs. Plus précisément, le point fixe doit avoir un de ses arguments qui à chaque appel récursif prend une valeur structurellement plus petite que celle en entrée. Dans notre cas, le type naturel pour le découpage par blocs que nous souhaitons faire récursivement est le type `positive` des entiers binaires strictement positifs, comme nous l'avons déjà utilisé pour l'algorithme de Strassen au chapitre 3.

Or, dans le cas d'une matrice de taille impaire $2p+1$ en entrée, un des blocs A_1 ou A_3 aura une taille qui, même si elle est plus petite en tant qu'entier, ne sera pas reconnue structurellement plus petite que $2p+1$ (par exemple, $p+1$).

Pour contourner ces deux problèmes, nous traitons spécifiquement le cas impair en enlevant la dernière ligne et la dernière colonne de la matrice avant d'appliquer récursivement notre fonction, puis en corrigeant le résultat en prenant en compte la ligne et la colonne qui ont été retirées. Cette technique est très proche du « pelage dynamique » (*dynamic pealing*) que nous avons adopté au chapitre 3. L'identité précédente devient dans le cas impair :

$$\begin{bmatrix} A_1 & A_2 & a \\ 0 & A_3 & b \\ 0 & 0 & c \end{bmatrix}^{-1} = \begin{bmatrix} {A_1}^{-1} & -{A_1}^{-1}A_2{A_3}^{-1} & {A_1}^{-1}(A_2{A_3}^{-1}b - a)c^{-1} \\ 0 & {A_3}^{-1} & -{A_3}^{-1}ac^{-1} \\ 0 & 0 & c^{-1} \end{bmatrix}$$

avec, pour une matrice globale carrée de taille $2p+1$, des blocs A_1, A_2 et A_3 de taille $p \times p$, a et b de taille $p \times 1$ et c de taille 1×1.

Nous traduisons ceci formellement en implémentant la fonction suivante, qui prend une matrice triangulaire supérieure non singulière et renvoie son inverse :

```
1  Fixpoint upper_tri_inv {n : positive} : mxA n n -> mxA n n :=
2    match n return let M := mxA n n in M -> M with
3    | xH => fun A => (fun_of_matrix A 0%C 0%C)^-1%:M
4    | xO p => fun A =>
5      let A := castmx (addpp p, addpp p) A in
6      let iA1 := @upper_tri_inv p (ulsubmx A) in
7      let iA3 := @upper_tri_inv p (drsubmx A) in
8      let R := Strassen (Strassen (- iA1) (ursubmx A)) iA3 in
9      castmx (esym (addpp p), esym (addpp p)) (block_mx iA1 R 0 ↙
           ↳ iA3)
10   | xI p => fun (A' : mxA (xI p) (xI p)) =>
11     let A := castmx (addpp1 p, addpp1 p) A' in
12     let iA1 := @upper_tri_inv p (ulsubmx A) in
```

```
13      let A2 := ursubmx A in
14      let lA2 := lsubmx A2 in
15      let rA2 := rsubmx A2 in
16      let A3 := drsubmx A in
17      let A3ul := ulsubmx A3 in
18      let A3ur := ursubmx A3 in
19      let A3dr := drsubmx A3 in
20      let iA3ul := @upper_tri_inv p A3ul in
21      let iA3dr := (fun_of_matrix A3dr 0%C 0%C)^-1%:M in
22      let R3 := - iA3ul *m A3ur *m iA3dr in
23      let iA3 := block_mx iA3ul R3 0 iA3dr in
24      let R := row_mx (- iA1 *m lA2 *m iA3ul) (Strassen (Strassen ↙
            ↳ iA1 lA2) iA3ul *m A3ur *m iA3dr - iA1 *m rA2 *m iA3dr)
25      in
26      castmx (esym (addpp1 p), esym (addpp1 p)) (block_mx iA1 R ↙
            ↳ (0 : mxA (p + 1) p) iA3)
27    end.
```

La ligne 3 traite spécifiquement le cas d'une matrice de taille 1×1. Les lignes 4 à 9 utilisent l'identité prouvée par le lemme `invmx_ublock` pour traiter le cas d'une matrice de taille paire. La fonction `castmx` est utilisée pour convertir la taille `x0 p` à `p + p`, cette dernière permettant au découpage de s'effectuer. Les lignes restantes prennent en charge les matrices de taille impaire.

Pour exprimer la correction de notre fonction `upper_tri_invmx`, nous définissons un prédicat `upper_triangular_mx` testant si une matrice est triangulaire supérieure. La fonction `upper_tri_invmx` est alors correcte si, pour toute matrice inversible satisfaisant ce prédicat, elle renvoie bien son inverse :

```
Lemma upper_tri_invP (p : positive) (M : 'M[F]_p) :
  M \in unitmx -> upper_triangular_mx M ->
  upper_tri_inv M = invmx M.
```

La preuve de ce lemme se fait par récurrence sur la taille de la matrice. Celle-ci est découpée en quatre blocs. Nous utilisons alors des lemmes auxiliaires que nous prouvons sur le prédicat `upper_triangular_mx` appliqué à des matrices par blocs. Notamment, sous certaines conditions sur le découpage, les blocs diagonaux sont également triangulaires supérieurs et le bloc en bas à gauche est nul. Le reste de la preuve consiste en des manipulations algébriques sur chaque bloc.

4.4 PERFORMANCES

La figure 4.2 montre les performances en pratique des différents algorithmes étudiés dans ce chapitre : le calcul du rang (« Rang »), celui de la décomposition PLU (« PLU ») et enfin l'inversion rapide de matrices triangulaires (« Inverse »).

La forme des courbes obtenues reflète clairement les complexités théoriques respectives de ces algorithmes : le calcul du rang et de la décomposition PLU se font en $\mathcal{O}(n^3)$ tandis que l'utilisation du produit de Strassen dans l'inversion rapide permet d'avoir une complexité en $\mathcal{O}(n^{\log 7})$.

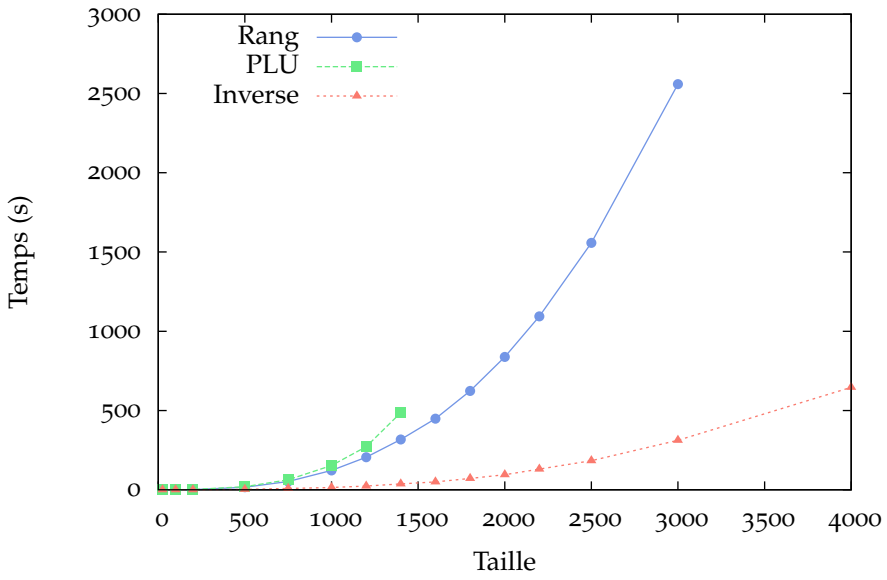


FIGURE 4.2 – Temps d'exécution des algorithmes présentés dans ce chapitre.

Taille	Temps (s)		
	Rang	PLU	Inverse
500	15,99	19,57	2,16
1 000	121,77	153,54	14,49
2 000	838,19		95,12
4 000	5 847,98		646,1

TABLE 4.1 – Temps d'exécution des algorithmes présentés dans ce chapitre.

La table 4.1 permet de quantifier plus précisément ces résultats. Lorsque les temps ne sont pas indiqués, cela signifie que la machine sur laquelle ces jeux de tests ont été exécutés ne disposait pas d'assez de mémoire pour mener les calculs à bien.

Un point important que nous n'avons pas discuté est la croissance des coefficients. En effet, lors des tests nous avons appliqué nos procédures à des matrices à coefficients dans un corps fini, ce qui rend réaliste un modèle où chaque opération arithmétique a un coût constant.

Cependant, si l'on travaille par exemple sur des nombres rationnels, les opérations comme l'addition et la multiplication ont un coût qui dépend de la taille du numérateur et du dénominateur des résultats intermédiaires. Nos algorithmes sont directement applicables dans un tel contexte ; nous n'avons fait l'hypothèse d'être sur un corps fini qu'au moment de ces tests, pour simplifier l'étude de performances.

4.5 CONCLUSION

Récapitulons nos contributions introduites dans le présent chapitre :

- Un algorithme formellement vérifié et exécutable pour le calcul du rang de matrices sur un corps.

- Un algorithme formellement vérifié et exécutable pour la décomposition PLU des matrices sur un corps.
- Un algorithme formellement vérifié et exécutable calculant l'inverse d'une matrice triangulaire avec la même complexité que l'algorithme de Strassen du chapitre 3.

L'intérêt des algorithmes étudiés ici est qu'ils permettent de résoudre de nombreux problèmes fondamentaux en algèbre linéaire, dont nous avons cité un certain nombre en introduction. Ils constituent également un exemple intéressant d'application de notre méthodologie basée sur les raffinements établie au chapitre 2.

En outre, l'algorithme de calcul du rang sera utilisé au chapitre 7 pour le calcul de groupes d'homologie issus d'images numériques.

Parmi les perspectives de prolongement de ce travail, mentionnons l'intégration d'autres représentations de matrices. Notamment, il serait intéressant de réutiliser les tableaux persistants introduits au chapitre 1 pour bénéficier d'un accès en temps constant aux éléments d'une matrice. Ainsi, les opérations de pivot pourraient être rendues plus efficaces. D'autres représentations seraient utiles, comme des matrices creuses.

Il est également possible d'expérimenter d'autres stratégies de recherche du pivot. Par souci de simplicité, nous avons choisi une recherche linéaire mais, selon le corps de coefficients, il peut être intéressant de chercher par exemple un pivot de valeur absolue maximale. Nous avons pris soin de rendre nos preuves modulaires de façon à ce qu'un tel changement de stratégie ne demande pas de retoucher l'ensemble de la preuve de correction de l'algorithme concerné, mais simplement d'établir un nouveau lemme de correction de la stratégie de recherche.

Enfin, nous avons évoqué le fait que l'inversion rapide de matrices triangulaires formalisée dans la section 4.3 est une sous-routine de l'algorithme de Bunch et Hopcroft [J. R. Bunch et J. E. Hopcroft, 1974] d'inversion rapide de toute matrice carrée inversible. Formaliser l'ensemble de cet algorithme est un objectif intéressant et devrait être réalisable avec les techniques que nous avons présentées.

5 FORMES MATRICIELLES CANONIQUES

Pour étudier des propriétés de matrices sur une structure algébrique donnée, il est souvent pratique de classer ces matrices selon une relation d'équivalence préservant ces propriétés. Il suffit alors de choisir un représentant particulier facile à traiter. Calculer la forme canonique d'une matrice consiste donc à réduire cette matrice vers le représentant de sa classe.

Nous notons l'équivalence matricielle par le symbole ~ et la similitude par ≅.

Nous traiterons dans ce chapitre deux telles relations : l'équivalence matricielle et la similitude, qui sont des notions fondamentales en algèbre matricielle. Deux matrices A et B (non nécessairement carrées) sont équivalentes s'il existe deux matrices inversibles M et N telles que $MAN = B$. Ceci exprime que les systèmes linéaires représentés par A et B admettent le même espace de solutions (à isomorphisme près). Dit encore autrement, les matrices A et B représentent une même application linéaire modulo un changement de base dans l'espace de départ et un autre dans l'espace d'arrivée.

Par ailleurs, deux matrices carrées A et B sont semblables s'il existe une matrice inversible P telle que $PAP^{-1} = B$ (ou encore $PA = BP$). La similitude est une propriété plus forte que l'équivalence, mais ne s'applique qu'à des matrices carrées et exprime un même changement de base appliqué au départ et à l'arrivée : si A représente un endomorphisme f dans une base $\mathcal{B}$ et est semblable à B, alors B représente également f dans une base $\mathcal{B}'$.

Les formes canoniques associées à ces deux relations dépendent de la structure algébrique des coefficients. Nous verrons que l'équivalence donne sur un anneau euclidien[14] la forme normale de Smith et sur un corps celle de Gauss-Jordan, tandis que la similitude donne sur un corps la forme de Frobenius et sur un corps algébriquement clos celle de Jordan. Ceci est récapitulé par la table 5.1.

[14] *Un anneau est euclidien s'il est muni d'une division euclidienne, comme $\mathbb{Z}$ l'anneau des entiers ou $\mathbb{K}[X]$ l'anneau des polynômes sur un corps $\mathbb{K}$.*

Structure	Relation	Forme canonique
Anneau euclidien	~	Smith
Corps	~	Gauss-Jordan
Corps	≅	Frobenius
Corps algébriquement clos	≅	Jordan

Table 5.1 – Exemples de relations d'équivalence matricielles sur des structures algébriques et formes canoniques associées.

Nous avons déjà rencontré, sans citer son nom, la forme de Gauss-Jordan dans l'introduction du chapitre 4. En effet, dans la décomposition matricielle $A = PLI_rUQ$, la matrice I_r n'est autre que la forme de Gauss-Jordan de A. Nous traiterons donc dans le présent chapitre des trois autres formes canoniques figurant dans la table 5.1.

Le coeur de notre étude est un algorithme de calcul de la forme normale de Smith que nous formaliserons de manière à pouvoir l'exécuter à l'intérieur de Coq. Cet algorithme peut être vu comme une extension à la structure d'anneau euclidien de l'élimination de Gauss que nous avons vue au chapitre 4 et qui s'appliquait aux matrices sur un corps. Nous évoquerons son utilité dans le contexte du calcul de groupes d'homologie au chapitre 7.

À partir d'une description effective de cet algorithme (section 5.2), nous déduirons à moindre frais l'existence de la forme de Frobenius (section 5.3) puis de celle de Jordan (section 5.4) ainsi que les propriétés habituelles de diagonalisation des endomorphismes d'un espace vectoriel de dimension finie.

Nous verrons dans la section 5.1 que ces deux relations sont reliées par un résultat important. La notion de similitude de deux matrices sur un corps joue un rôle important car elle préserve de nombreuses propriétés. Ainsi, étudier le rang, la trace, le déterminant, les valeurs propres, le polynôme minimal ou caractéristique d'une matrice revient à l'étudier sur sa forme normale de Frobenius, ce qui peut simplifier le problème.

À titre d'exemple, la forme de Frobenius donne un moyen efficace d'élever une matrice à de grandes puissances [M. Giesbrecht, 1995]. Ceci peut être utile pour calculer l'état d'une chaîne de Markov à l'instant n, où n est grand. En effet, ce calcul revient à élever la matrice de transition de la chaîne à la puissance n.

5.1 ÉQUIVALENCE ET SIMILITUDE DE MATRICES

Nous définissons formellement la similitude de matrices comme suit :

```
Definition similar m n (A : 'M[R]_m) (B : 'M[R]_n) :=  m = n /\
  (exists P : 'M_m , P \in unitmx /\ P *m A = (conform_mx P B) ↙
      ↳ *m P).
```

La fonction `conform_mx` *prend en arguments deux matrices* `A` *et* `B` *de types respectifs* `'M_(m1,n1)` *et* `'M_(m2,n2)`*, et renvoie* `B` *si* `m1 = m2` *et* `n1 = n2`*,* `A` *sinon. Le type de* `conform_mx` *est* `'M_(m1,n1) -> 'M_(m2,n2) -> 'M_(m1,n1)`.

Cette définition relâche le type des matrices en arguments : `similar` peut être appliqué à des matrices `A` et `B` ayant des tailles non convertibles, mais `similar A B` ne sera prouvable que si `A` et `B` ont des tailles prouvablement égales. Nous utilisons la même astuce pour la définition de l'équivalence :

```
Definition equivalent m1 n1 m2 n2 (A : 'M[R]_(m1,n1)) (B : ↙
    ↳ 'M[R]_(m2,n2)) :=
    [/\ m1 = m2, n1 = n2 & exists M, exists N,
    [/\ M \in unitmx , N \in unitmx & M *m A *m N = conform_mx ↙
        ↳ A B]].
```

Un lien important entre ces deux notions est le fait que deux matrices carrées sont semblables si et seulement si leurs matrices caractéristiques[15] sont équivalentes :

[15] *Ici et dans toute la suite, pour une matrice A sur un anneau R, nous appelons matrice caractéristique la matrice $XI - A$ où X est l'indéterminée de l'anneau de polynômes $R[X]$. Le polynôme caractéristique de A est donc le déterminant de sa matrice caractéristique.*

```
Theorem similar_fundamental m n (A: 'M[R]_m) (B : 'M[R]_n) :
  similar A B <-> equivalent (char_poly_mx A) (char_poly_mx B).
```

Ici `char_poly_mx A` désigne la matrice caractéristique de `A`.

Ce résultat est parfois appelé « théorème fondamental de similitude sur un corps ». Pour l'établir, nous suivons la preuve décrite dans [J. H. M. Wedderburn, 2002].

Une des deux implications est facile : si A et B sont semblables, il existe une matrice inversible P telle que $A = PBP^{-1}$ et donc $XI - A = P(XI - B)P^{-1}$, ce qui implique que $XI - A$ et $XI - B$ sont équivalentes.

La réciproque est plus difficile. Supposons qu'il existe des matrices inversibles M et N telles que :

$$M(XI - A)N = XI - B$$

Jusqu'à présent, les objets que nous manipulions étaient des matrices de polynômes. Mais nous devons à présent les voir formellement comme des polynômes de matrices, ce qui est possible en utilisant l'isomorphisme suivant :

$$\phi : M(R[X]) \to M(R)[X]$$

Cet isomorphisme était déjà un des ingrédients clés dans la preuve formelle du théorème de Cayley-Hamilton décrit dans [S. Ould Biha, 2008]. Il a été défini dans la bibliothèque SSReflect de la manière suivante :

```
Definition phi n (A : 'M[{poly R}]_n.+1) :=
  \poly_(k < \max_i \max_j size (A i j)) \matrix_(i, j) (A i ↙
      ↳ j)`_k.
```

La notation `\poly_(i < n) p i` *permet de définir un polynôme par l'expression générale de ses coefficients.*

Dans la bibliothèque SSReflect un polynôme `p` est vu comme une séquence de coefficients. De ce fait, le terme `size p` désigne la taille de cette séquence et, si `p` est non nul, son degré est représenté par `(size p).-1`.

Nous définissons ensuite les polynômes de matrices M_1, M_0 et N_1, N_0 respectivement par division à gauche de $\phi(M)$ et division à droite de $\phi(N)$:

$$\phi(M) = (X - B)M_1 + M_0 \quad \text{avec} \quad \deg M_0 = 0$$

$$\phi(N) = N_1(X - B) + N_0 \quad \text{avec} \quad \deg N_0 = 0$$

L'étape clé de cette preuve consiste à établir l'identité suivante :

$$M_0(X - A)N_0 = (1 - (X - B)R_1)(X - B)$$

avec $R_1 = M_1\phi(M^{-1}) + \phi(N^{-1})N_1 - M_1(X - A)N_1$.

Ceci se fait par des manipulations algébriques élémentaires. Ensuite, le degré du membre gauche étant 1 (M_0 et N_0 sont des constantes), R_1 doit être nul (si $R_1 \neq 0$, le membre droit aurait un degré au moins 2). L'identité précédente devient alors :

$$M_0(X - A)N_0 = (X - B)$$

À partir de quoi on peut déduire, en identifiant les coefficients :

$$M_0N_0 = 1$$

$$M_0AN_0 = B$$

D'où $M_0 = N_0^{-1}$, $N_0^{-1}AN_0 = B$, c'est-à-dire que A et B sont semblables.

Même si cette preuve peut paraître naturelle, les objets d'étude (des polynômes sur un anneau non commutatif et non intègre[16] en général) demandent des précautions, car ils ne bénéficient évidemment pas de toutes les propriétés habituelles des polynômes sur un anneau commutatif intègre. L'utilisation d'un assistant de preuve permet d'éviter la tentation d'arguments par analogie incorrects.

[16] *Un anneau est intègre si tous ses éléments non nuls sont réguliers, i.e. non diviseurs de zéro.*

Par exemple, raisonner sur le degré de tels polynômes demande d'avoir à disposition des lemmes spécifiques pour les polynômes unitaires, ou plus généralement pour le cas où le coefficient dominant est un élément régulier de l'anneau de base. Heureusement, la bibliothèque SSReflect fournit les niveaux d'abstraction adéquats.

5.2 FORME NORMALE DE SMITH

Soit $\mathcal{R}$ un anneau euclidien. Une matrice est dite en forme normale de Smith si elle se présente de la manière suivante :

$$\begin{bmatrix} d_1 & 0 & \cdots & \cdots & & \cdots & 0 \\ 0 & d_2 & \ddots & & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & & & \vdots \\ 0 & \cdots & 0 & d_k & 0 & \cdots & 0 \end{bmatrix}$$

avec la particularité que $d_1 \mid d_2 \mid \cdots \mid d_k$.

Nous allons voir qu'un algorithme permet, au moyen d'opérations compatibles avec l'équivalence des matrices, de mettre toute matrice sur $\mathcal{R}$ en forme normale de Smith. Nous pourrions travailler plus généralement sur un anneau principal[17] mais comme nous le verrons plus loin, la preuve de terminaison est plus simple à formaliser dans le cas euclidien.

[17] *En algèbre constructive, un anneau intègre est principal si toute suite croissante d'idéaux principaux admet deux termes consécutifs égaux [R. Mines et al., 1988].*

5.2.1 Algorithme de mise en forme de Smith

Remarquons tout d'abord que l'élimination gaussienne décrite au chapitre 4 n'est pas immédiatement applicable à une matrice sur cet anneau. En effet, lors de l'étape d'élimination des coefficients, une division par le pivot est nécessaire. Or, dans le cadre d'un anneau, rien ne garantit que cette division soit exacte ni que le pivot soit inversible. L'algorithme que nous allons décrire peut donc être vu comme une généralisation aux anneaux principaux de l'élimination gaussienne des matrices sur un corps.

Pour permettre cette extension, il nous faut introduire un nouveau type d'opérations élémentaires, qui généralisent celles déjà mises en œuvre au chapitre 4. Soient a et b des éléments de $\mathcal{R}$. Le théorème de Bezout nous assure qu'il existe u et v tels que $ua + vb = \gamma$ où $\gamma = \text{pgcd}(a, b)$. En posant $a' = \frac{a}{\gamma}$ et $b' = \frac{b}{\gamma}$, ces divisions étant exactes par définition du pgcd, on obtient l'identité : $ua' + vb' = 1$. Considérons la matrice carrée de taille m suivante :

Les coefficients n'apparaissant pas explicitement dans E_{Bezout} sont nuls.

$$E_{\text{Bezout}}(a, b, m, k) = \begin{array}{c} \\ \\ \\ \\ \text{(ligne } k\text{)} \\ \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccc} & & & & \text{(col. } k\text{)} & & \end{array} \\ \begin{bmatrix} u & & & & v & & \\ & 1 & & & & & \\ & & \ddots & & & & \\ & & & 1 & & & \\ -b' & & & & a' & & \\ & & & & & 1 & \\ & & & & & & \ddots \\ & & & & & & & 1 \end{bmatrix} \end{array}$$

On remarque que $\det E_{\text{Bezout}}(a, b, m, k) = ua' + vb' = 1$, donc en particulier la matrice ci-dessus est inversible. On formalise ces matrices comme suit :

L'expression `a *+ b`, où `b` est un booléen, vaut `a` si `b` est vrai, `0` sinon. Rappelons que `b%:R` vaut `1` si `b` est vrai, `0` sinon.

```
Definition combine_mx (a b c d : R) (m : nat) (k : 'I_m) :=
  let k' := rshift 1 k in
  let d := \row_j (a *+ (j == 0) + d *+ (j == k') + ((j != 0) ↙
      ↪ && (j != k'))%:R) in
  diag_mx d + c *: delta_mx k' 0 + b *: delta_mx 0 k'.

Definition Bezout_mx (a b : R) (m : nat) (k : 'I_m) :=
  let:(_,u,v,a1,b1) := egcdr a b in combine_mx u v (-b1) a1 k.
```

Comme pour les autres matrices d'opérations élémentaires, la multiplication à gauche d'une matrice quelconque par $E_{\text{Bezout}}(a, b, m, k)$ peut s'interpréter comme une opération sur les lignes :

$$E_{\text{Bezout}}(a,b,m,k) \times \begin{bmatrix} L_1 \\ L_2 \\ \vdots \\ L_{k-1} \\ L_k \\ L_{k+1} \\ \vdots \\ L_m \end{bmatrix} = \begin{bmatrix} uL_1 + vL_k \\ L_2 \\ \vdots \\ L_{k-1} \\ -b'L_1 + a'L_k \\ L_{k+1} \\ \vdots \\ L_m \end{bmatrix}$$

Ces opérations sur les lignes sont décrites formellement par :

```
Definition combine_step (a b c d : R) (m n : nat) (M : 'M_(1 + ↙
    ↳ m, 1 + n)) (k : 'I_m) :=
  let k' := rshift 1 k in
  let r0 := a *: row 0 M + b *: row k' M in
  let rk := c *: row 0 M + d *: row k' M in
  \matrix_i (r0 *+ (i == 0) + rk *+ (i == k') + row i M *+ ((i ↙
      ↳ != 0) && (i != k'))).

Definition Bezout_step (a b : R) (m n : nat) (M : 'M_(1 + m, 1 ↙
    ↳ + n)) (k : 'I_m) :=
  let:(_,u,v,a1,b1) := egcdr a b in combine_step u v (-b1) a1 M ↙
      ↳ k.
```

Un lemme relie ces opérations sur les lignes aux matrices élémentaires correspondantes :

```
Lemma Bezout_stepE a b (m n : nat) (M : 'M_(1 + m,1 + n)) k :
  Bezout_step a b M k = Bezout_mx a b k *m M.
```

Soit maintenant $A = (a_{i,j})$ une matrice à coefficients dans $\mathcal{R}$. Nous allons montrer comment réduire A sous forme de Smith en utilisant des opérations élémentaires. Comme pour l'élimination gaussienne, on commence par trouver un pivot g non nul dans A, que l'on ramène en haut à gauche (si $A = 0$, elle est en forme de Smith). On recherche alors dans la première colonne un élément non divisible par g. Supposons que $g \nmid a_{k,1}$, on multiplie alors à gauche la matrice par $E_{\text{Bezout}}(g, a_{k,1}, m, k)$:

$$E_{\text{Bezout}}(g,a_{k,1},m,k) \times \begin{bmatrix} g & L_1 \\ a_{2,1} & L_2 \\ \vdots & \vdots \\ a_{k,1} & L_k \\ \vdots & \vdots \\ a_{m,1} & L_m \end{bmatrix} = \begin{bmatrix} \gamma & uL_1 + vL_k \\ a_{2,1} & L_2 \\ \vdots & \\ -g'g + a'a_{k,1} & -g'L_1 + a'L_k \\ \vdots & \vdots \\ a_{m,1} & L_m \end{bmatrix}$$

avec l'identité de Bezout $ug + va_{k,1} = \gamma = \text{pgcd}(g, a_{k,1},)$ et en notant comme précédemment $g' = \frac{g}{\gamma}$, $a' = \frac{a_{k,1}}{\gamma}$.

Or, par définition de γ, on a : $\gamma \mid -g'g + a'a_{k,1}$. D'autre part, tous les coefficients de la première colonne de A qui étaient divisibles par g le sont par γ. On peut donc répéter cette procédure jusqu'à obtenir une matrice dont le coefficient en haut à gauche (que nous notons toujours g) divise tous

les coefficients de la première colonne. Dès lors, des combinaisons linéaires sur les lignes permettent de se ramener à une matrice B de la forme :

$$B = \begin{bmatrix} g & b_{1,2} & \cdots & b_{1,n} \\ g & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ g & b_{m,2} & \cdots & b_{m,n} \end{bmatrix}$$

Nous recherchons alors dans toute la sous-matrice de B encadrée un élément non divisible par g. Si un tel coefficient $b_{i,j}$ est trouvé, il est ramené en haut par permutation des lignes 1 et i. Le coefficient en haut à gauche est alors toujours g[18] et des multiplications à droite par des matrices E_{Bezout} permettent, de la même manière que précédemment, d'obtenir une matrice dont le coefficient en haut à gauche divise tous les autres coefficients.

[18] *Cette astuce nous a été inspirée par un développement orienté preuve de l'algorithme de Smith par Georges Gonthier.*

Cette première étape est implémentée par la fonction `improve_pivot` :

```
1  Fixpoint improve_pivot k {m n} : 'M_(1 + m) -> 'M_(1 + m, 1 + ↙
       ↪ n) -> 'M_(1 + n) -> 'M_(1 + m) * 'M_(1 + m, 1 + n) * ↙
       ↪ 'M_(1 + n) :=
2    match k with
3    | 0 => fun L A R => (L,A,R)
4    | p.+1 => fun L A R =>
5      let a := A.(0,0) in
6      if find1 A a is Some i then
7        let L := Bezout_step A.(0,0) A.(rshift 1 i,0) L i in
8        let A := Bezout_step A.(0,0) A.(rshift 1 i,0) A i in
9        improve_pivot p L A R
10     else let u := dlsubmx A in let vA := ursubmx A in
11     let vL := usubmx L in
12     let u' := map_mx (fun x => 1 - odflt 0 (x %/? a)) u in
13     let L := col_mx (usubmx L) (u' *m vL + dsubmx L) in
14     let A := block_mx a%:M vA (const_mx a) (u' *m vA + drsubmx ↙
           ↪ A) in
15     if find2 A a is Some ij then
16       let A := xrow 0 ij.1 A in let L := xrow 0 ij.1 L in
17       let R := (Bezout_step A.(0,0) A.(0,rshift 1 ij.2)) R^T ↙
             ↪ ij.2)^T in
18       let A := (Bezout_step A.(0,0) A.(0,rshift 1 ij.2)) A^T ↙
             ↪ ij.2)^T in
19       improve_pivot p L A R
20     else (L, A, R)
21   end.
```

Cette fonction prend en arguments un entier `k` qui représente le nombre d'étapes restantes, la matrice d'origine et deux matrices de passage courantes. Si le nombre d'étapes restantes `k` est nul, les matrices sont renvoyées inchangées (ligne 3). Dans le cas contraire, un élément non divisible par le pivot est recherché sur la première colonne (fonction `find1`, ligne 6). Si un tel élément est trouvé sur une ligne d'indice `i`, une étape de Bezout est effectuée entre la première ligne et celle d'indice `i`, et la fonction est appelée récursivement (lignes 7 à 9). Si au contraire le pivot divise tous les éléments de la première colonne, des combinaisons linéaires (lignes 10 à 14) ramènent à une matrice de la forme de la matrice B vue plus haut. Enfin, les lignes

restantes recherchent un élément non divisible par le pivot dans toute la matrice (fonction `find2`), effectuent une étape de Bezout sur les colonnes le cas échéant, et rappellent récursivement la fonction.

Nous avons fait plusieurs choix pour implémenter cette fonction. Tout d'abord, l'argument `k` bornant le nombre d'étapes permet d'avoir facilement une récursion structurelle (cet entier décroît de 1 à chaque étape). Dans cette technique usuelle, `k` est souvent appelé le « carburant » (*fuel*) de la récursion. La contrepartie est que pour appeler la fonction, il faut être à même de borner a priori le nombre d'étapes. C'est à ce moment que l'hypothèse que nous avons faite que $\mathcal{R}$ soit un anneau euclidien se révèle importante : elle permet de prendre comme majorant du nombre d'étapes la norme euclidienne du coefficient en haut à gauche de la matrice d'origine.

Nous avons également fait le choix de passer en arguments des matrices de passage initiales, qui sont mises à jour au fur et à mesure de la procédure. Du point de vue du calcul, cette approche a deux avantages. Tout d'abord, elle évite de recourir à des multiplications matricielles de ces matrices de passage, asymptotiquement plus coûteuses que d'effectuer directement les opérations élémentaires. Ensuite, elle permet à la fonction `improve_pivot` d'être récursive terminale, ce qui peut avoir un bon impact sur les performances.

Le point négatif est qu'il est légèrement plus difficile d'exprimer et de manipuler formellement le lien entre les matrices passées en arguments et celles renvoyées par la fonction. En effet, la spécification de celle-ci fait intervenir les inverses des matrices de passage :

```
Inductive improve_pivot_spec L M R :
  'M_(1 + m) * 'M_(1 + m, 1 + n) * 'M_(1 + n) -> Type :=
  ImprovePivotSpec L' A R' of
    L^-1 *m M *m R^-1 = L'^-1 *m A *m R'^-1
    & (forall i j, A 0 0 %| A i j)
    & (forall i, A i 0 = A 0 0)
    & A 0 0 %| M 0 0
    & L' \in unitmx & R' \in unitmx
  : improve_pivot_spec L M R (L',A,R').
```

L'énoncé ci-dessus se lit comme suit : étant données trois matrices `L`, `M` et `R`, un triplet `(L',A,R')` vérifie la spécification si appliquer à `M` l'inverse des opérations représentées par les matrices de passage initiales `L` et `R` donne le même résultat que l'application des inverses des matrices de passage `L'` et `R'` à la matrice `A`.

Le lemme de correction de la fonction `improve_pivot` énonce que pour une matrice `M` de départ dont le coefficient en haut à gauche est non-nul et de norme inférieure à un entier `k`, et pour des matrices inversibles `L` et `R`, le triplet renvoyé par `improve_pivot k L M R` satisfait la spécification représentée par le type inductif `improve_pivot_spec` :

```
Lemma improve_pivotP k (L : 'M_(1 + m)) (M : 'M_(1 + m, 1 + n)) ↙
    ↳ R :
  (enorm (M 0%R 0%R) <= k)%N -> M 0 0 != 0 -> L \in unitmx -> R ↙
      ↳ \in unitmx ->
  improve_pivot_spec L M R (improve_pivot k L M R).
```

Par soustractions successives de la première ligne à toutes les autres puis par des combinaisons linéaires de colonnes, on obtient une matrice C :

$$C = \left[\begin{array}{c|ccc} g & 0 & \cdots & 0 \\ \hline 0 & & & \\ \vdots & & C' & \\ 0 & & & \end{array}\right]$$

où g divise tous les coefficients de C'.

L'algorithme global de calcul de la forme de Smith procède de la manière suivante : il stocke le pivot g obtenu à l'étape précédente, puis divise tous les coefficients de C' par g et s'applique récursivement à la matrice obtenue. Notons $k = \min(m, n)$. À partir des pivots $g_1, \ldots, g_k$ obtenus, la sortie finale de l'algorithme est donnée par la séquence $d_1, \ldots, d_k$ suivante :

$$d_1, d_2, \ldots, d_k = g_1, g_1 g_2, \ldots, \prod_{i=1}^{k} g_i$$

La forme normale de Smith de la matrice d'origine est alors la matrice diagonale de taille $m \times n$ suivante :

$$\begin{bmatrix} d_1 & & & & & & \\ & d_2 & & & & & \\ & & \ddots & & & & \\ & & & d_k & & & \\ & & & & 0 & & \\ & & & & & \ddots & \\ & & & & & & 0 \end{bmatrix}$$

Cette procédure globale est implémentée par la fonction `Smith` :

```
1  Fixpoint Smith {m n} : 'M_(m,n) -> 'M_(m) * seq R * 'M_(n) :=
2    match m, n return 'M_(m, n) -> 'M_(m) * seq R * 'M_(n) with
3    | _.+1, _.+1 => fun A : 'M_(1 + _, 1 + _) =>
4      if find_pivot A is Some (i, j) then
5        let a := A.(i,j) in let A := xrow i 0 (xcol j 0 A) in
6        let: (L,A,R) := improve_pivot (enorm a) 1%:M A 1%:M in
7        let a := A.(0,0) in
8        let u := dlsubmx A in let v := ursubmx A in
9        let v' := map_mx (fun x => odflt 0 (x %/? a)) v in
10       let A := drsubmx A - const_mx 1 *m v in
11       let: (L', d, R') := Smith (map_mx (fun x => odflt 0 (x ↙
             ↳ %/? a)) A) in
12       (lift0_mx L' *m block_mx 1 0 (-const_mx 1) 1%:M *m (xcol ↙
             ↳ i 0 L),
13       a :: [seq x * a | x <- d],
14       (xrow j 0 R) *m block_mx 1 (-v') 0 1%:M *m lift0_mx R')
15     else (1%:M, [::], 1%:M)
16   | _, _ => fun A => (1%:M, [::], 1%:M)
17   end.
```

La fonction `Smith` prend en argument une matrice et renvoie une séquence composée des coefficients diagonaux non-nuls de sa forme de Smith, ainsi que les matrices de passage correspondantes. La première étape (lignes 4 et 5) consiste à rechercher un pivot non-nul dans toute la matrice et le ramener en haut à gauche. Si aucun pivot n'est trouvé, tous les coefficients sont nuls et une séquence vide est donc renvoyée. Sinon, la fonction `improve_pivot`

définie précédemment est appelée (ligne 6) puis des opérations élémentaires sur les lignes sont effectuées (lignes 8 à 10) pour obtenir une matrice de la forme de la matrice C représentée plus haut. La sous-matrice en bas à droite est alors divisée par le pivot et un appel récursif est effectué (lignes 11 à 13). La séquence de coefficients et les matrices de passage ainsi obtenues sont alors mises à jour (lignes 14 à 16).

Pour écrire formellement la spécification de la fonction `Smith`, nous définissons `diag_seq_mx`, qui reconstruit la matrice diagonale à partir de cette séquence des coefficients diagonaux :

```
Definition diag_mx_seq m n s :=
  \matrix_(i < m, j < n) s`_i *+ (i == j :> nat).
```

Rappelons que la notation `x *+ n`, où `x` est élément d'un anneau et `n` un entier naturel, désigne la somme `x +...+ x` itérée `n` fois. Dans l'expression du coefficient général de la matrice ci-dessus, `i` et `j` sont des ordinaux (i.e. des entiers bornés respectivement par `m` et `n`) ; ils ont donc des types distincts (notés respectivement `'I_m` et `'I_n`). La notation `i == j :> nat` permet d'indiquer à Coq de les comparer en tant qu'entiers naturels et de renvoyer un booléen. Une coercion envoie ensuite ce booléen sur un entier (`true` est interprété par `1` et `false` par `0`). Ainsi, `s`_i *+ (i == j :> nat)` désigne l'élément d'indice `i` dans `s` si `i` et `j` ont la même valeur, `0` sinon.

La spécification finale exprime que la fonction `Smith` appliquée à une matrice `A` renvoie une liste `s` et des matrices `L0` et `R0` telles que :

- La séquence `s` est triée pour la relation de division.
- La matrice `diag_mx_seq m n s` est équivalente à `A`, avec pour matrices de passage `L0` et `R0`

Ce qui se traduit formellement par un prédicat inductif :

```
Inductive Smith_spec {m n} M : 'M_m * seq R * 'M_n -> Type :=
  SmithSpec L0 d R0 of L0 *m M *m R0 = diag_mx_seq m n d
    & sorted (@dvdr R) d
    & L0 \in unitmx & R0 \in unitmx : Smith_spec M (L0, d, R0).
```

et le lemme de correction suivant :

```
Lemma SmithP (m n : nat) (M : 'M_(m,n)) : Smith_spec M (Smith M).
```

5.2.2 Unicité

L'algorithme de mise en forme de Smith met en oeuvre des calculs de pgcd, dont le résultat n'est unique que modulo association (c'est-à-dire la relation $\sim$ définie par $a \sim b$ si et seulement si $a|b$ et $b|a$). Nous ne pouvons donc pas espérer associer à toute matrice une forme de Smith strictement unique, mais seulement modulo la relation $\sim$ sur les coefficients diagonaux. Comme nous sommes dans le cas d'un anneau intègre, cela revient à dire que la forme de Smith est unique modulo la multiplication des coefficients diagonaux par des inversibles.

Ce résultat a été formalisé par Guillaume Cano.[19] Plus précisément, il établit que les coefficients de la forme de Smith d'une matrice quelconque A s'expriment à partir de pgcd de mineurs de A (i.e. le déterminant des

[19] *Communication personnelle.*

sous-matrices de A). C'est-à-dire, si en notant $\wedge$ le pgcd et $|A|_k$ l'ensemble des mineurs d'ordre k de A :

$$\prod_{i=1}^{k} d_i \sim \bigwedge_{x \in |A|_k} x$$

Ce résultat est exprimé formellement en utilisant des notions de sous-matrices et de mineurs définis à l'aide de fonctions de ré-indexation :

```
Variables p q m n : nat.

Definition submatrix (f : 'I_p -> 'I_m) (g : 'I_q -> 'I_n) (A : ↙
    ↳ 'M_(m,n)) :=
  \matrix_(i < p, j < q) A (f i) (g j).

Definition minor (f : 'I_p -> 'I_m) (g : 'I_p -> 'I_n) (A : ↙
    ↳ 'M_(m,n)) :=
  \det (submatrix f g A).
```

Cette preuve fait appel à une formalisation de la preuve de la formule de Binet-Cauchy [V. Siles, 2012].

Le lemme précédent utilisé avec $k = 1$ permet de déterminer de manière unique (modulo la relation d'association $\sim$) le premier élément diagonal de la forme de Smith. Puis il détermine ainsi, de proche en proche, tous les éléments diagonaux. Parmi les matrices équivalentes modulo $\sim$ à la forme de Smith d'une matrice `A`, nous avons fixé un représentant de même déterminant que `A` :

```
Definition Smith_seq n m (M : 'M[R]_(n,m)) :=
  let: (L,d,R) := (Smith M) in
  if d is a :: d' then (\det L)^-1 * (\det R)^-1 * a :: d' else ↙
      ↳ nil.

Definition Smith_form m n (A : 'M[R]_(m,n)) :=
  diag_mx_seq m n (Smith_seq A).

Lemma det_Smith n (A :'M[R]_n) : \det (Smith_form A) = \det A.
```

5.2.3 Facteurs invariants

Soit $\mathcal{K}$ un corps, et A une matrice à coefficients dans $\mathcal{K}$. Nous allons appliquer l'algorithme de Smith à la matrice caractéristique $XI - A$ de A. D'après le lemme `det_Smith` évoqué plus haut, le déterminant de la forme normale de Smith de $XI - A$ est le polynôme caractéristique de A. Ce qui nous assure qu'aucun élément diagonal de la forme normale de Smith n'est nul pour les deux raisons suivantes :

- Le déterminant de la forme normale de Smith est le produit de ses coefficients diagonaux.
- Le polynôme caractéristique d'une matrice n'est jamais nul.

Les coefficients diagonaux sont donc des polynômes non nuls à coefficients dans un corps ; on peut donc diviser chacun de ces polynômes par son coefficient dominant pour avoir des polynômes unitaires :

```
Definition Frobenius_seq n (A : 'M[F]_n) :=
  [seq (lead_coef p)^-1 *: p | p <- (take n (Smith_seq ↙
      ↪ (char_poly_mx A)))].
```

où `take n s` est la séquence des `n` premiers éléments de la séquence `s`. L'algorithme de Smith nous renvoie une séquence, mais ne nous donne aucune information sur sa taille. Ici l'utilisation de la fonction `take` permet de montrer que `size (Frobenius_seq A) = n`. C'est un résultat qui sera utile dans le développement formel. L'utilisation de la fonction `take` ne change rien pour la forme normale de Smith comme le montre le résultat suivant :

```
Lemma diag_mx_seq_take n s :
  diagmx_seq n n s = diag_mx_seq n n (take n s).
```

Les facteurs invariants sont les polynômes non constants de la séquence `Frobenius_seq` :

```
Definition invariant_factors n (A : 'M[F]_n) :=
  [seq p : {poly R} <- (Frobenius_seq A) | 1 < size p].
```

Nous avons défini les facteurs invariants de telle manière qu'ils soient unitaires, car plus loin nous parlerons des matrices compagnes de ces polynômes. Or les propriétés usuelles des matrices compagnes ne sont vraies que pour les polynômes unitaires.

5.3 FORME DE FROBENIUS

La forme normale de Frobenius d'une matrice M est une matrice diagonale par blocs dont les blocs sont les matrices compagnes des facteurs invariants de M. Nous montrons donc d'abord notre formalisation des matrices compagnes avant de donner une définition formelle de la forme normale de Frobenius. Nous présentons ensuite un schéma de la preuve formelle qu'une matrice et sa forme normale de Frobenius sont semblables.

5.3.1 Matrices diagonales par blocs

Nous avons déjà évoqué dans les chapitres précédents le choix fait dans la librairie SSReflect d'indexer le type des matrices par leur taille, permettant souvent une écriture plus concise des énoncés. Mais ce choix a aussi des inconvénients, notamment le fait que le typage considère les tailles seulement modulo convertibilité. Ainsi, `'M[R]_(m,n)` et `'M[R]_(0 + m, n)` sont bien un seul et même type, mais `'M[R]_(m + 0, n)` en est un autre. Une contrainte rend ce problème particulièrement visible : la structure d'anneau telle que définie dans SSReflect exclut l'anneau trivial. En particulier, les matrices carrées ne forment un anneau que lorsque leur taille est convertible au successeur d'un entier (i.e. à un terme de la forme `m.+1`).

La notion de matrice diagonale par blocs n'a de sens que si on spécifie le découpage des blocs. L'intérêt de telles matrices est que de nombreuses opérations, comme l'addition, peuvent s'exprimer bloc par bloc, à condition toutefois que les deux matrices auxquelles on applique l'opération aient le même découpage en blocs.

Une première idée serait de représenter une matrice diagonale par blocs par une séquence de paires dépendantes (où chaque élément de la séquence

est un bloc indexé par sa taille). Cependant, cela ne permettrait pas d'exprimer par le typage que deux matrices ont des découpages compatibles.

Notre définition de matrice diagonale par blocs prend donc en premier argument une liste d'entiers naturels qui expriment la dimension de chaque bloc. Nous décidons de représenter les blocs eux-mêmes par une fonction `F : forall (n : nat), nat -> 'M[R]_n`. Ainsi, si le bloc numéro `i` a pour taille `n`, il sera représenté par `F n i`, ou encore, si la liste des tailles des blocs est `s`, par `F s`_i i`.

Pour construire une matrices à partir de ses blocs, nous utilisons la fonction `block_mx`. Plus précisément, si `A`, `B`, `C` et `D` sont des matrices de dimensions compatibles, `block_mx A B C D` représente la matrice :

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

Un premier essai pour décrire les matrices diagonales par blocs consiste à appliquer récursivement `block_mx` :

```
Fixpoint diag_block_mx (s : seq nat) (F : forall n, nat -> ↙
    ↳ 'M_n) :=
  if s is n :: s' return 'M_(sumn s) then
    block_mx (F n 0) 0 0 (diag_block_mx s' (fun n i => F n i.+1))
  else 0.
```

où `sumn s` est la somme des éléments de la séquence d'entiers `s`.

Mais cette définition ne remplit pas la condition que nous évoquions plus haut, d'avoir une taille convertible à un successeur. Dès lors, il sera impossible d'appliquer les opérations d'anneau aux matrices diagonales par blocs, ce qui était pourtant une des motivations initiales pour les définir.

Nous devons donc trouver un moyen de mettre en évidence un successeur dans la taille des matrices diagonales par blocs. Pour que ce soit possible, il faut exiger qu'il y ait au moins un bloc, et qu'un des blocs soit non-vide. De plus, si on veut pouvoir appliquer les opérations d'anneau bloc par bloc, il faut même imposer que tous les blocs soient non-vides. Le type de `F` devient donc `forall (n : nat), nat -> 'M[R]_n.+1`, et on définit les matrices diagonales par blocs en deux étapes, d'abord les fonctions auxiliaires suivantes :

```
Fixpoint size_sum_rec k (s : seq nat) : nat :=
  if s is x :: l then k + (size_sum_rec x l).+1 else k.

Fixpoint diag_block_mx_rec k (s : seq nat) (F : (forall n, nat ↙
    ↳ -> 'M[R]_n.+1)) :=
  if s is x :: l return 'M_((size_sum_rec k s).+1) then
    block_mx (F k 0) 0 0 (diag_block_mx_rec x l (fun n i => F n ↙
        ↳ i.+1))
  else F k 0.
```

Puis les fonctions principales :

```
Definition size_sum s :=
  if s is x :: l then size_sum_rec x l else 0.

Definition diag_block_mx s F :=
  if s is x :: l return 'M_((size_sum s).+1)
  then diag_block_mx_rec x l F else 0.
```

Nous avons défini la fonction `size_sum` de telle manière que la matrice construite par `diag_block_mx s F` soit de taille `(size_sum s).+1`. Avec cette nouvelle définition, la séquence `s` décrit les prédécesseurs des tailles des blocs (et non directement les tailles).

Notre définition de matrices diagonales par blocs est donc rendue moins naturelle du fait de cette contrainte sur la taille. Autoriser l'anneau trivial dans la définition des anneaux de SSReflect semble possible, mais serait moins confortable pour le traitement de certaines théories sur ces anneaux, notamment le développement des polynômes à une variable, qui y sont définis comme des listes de coefficients dont le dernier est non nul (l'ensemble des polynômes sur l'anneau trivial serait alors vide. En particulier ce ne serait pas un anneau).

5.3.2 Matrices compagnes

La matrice compagne d'un polynôme $p = X^n + a_{n-1}X^{n-1} + \ldots + a_1X + a_0$ est la matrice suivante :

$$\mathcal{C}_p = \begin{bmatrix} 0 & \ldots & 0 & 0 & -a_0 \\ 1 & \ddots & \vdots & \vdots & -a_1 \\ 0 & \ddots & 0 & \vdots & \vdots \\ \vdots & \ddots & 1 & 0 & \vdots \\ 0 & \ldots & 0 & 1 & -a_{n-1} \end{bmatrix}$$

Cette matrice est intéressante car p est à la fois son polynôme caractéristique et son polynôme minimal.

Formellement, si `p` est un polynôme non constant, la dimension de la matrice compagne de `p` est `(size p).-1`. Mais ici encore, pour les raisons vues précédemment, la taille d'une matrice compagne doit être convertible à un successeur. Nous définissons donc les matrices compagnes de sorte que leur taille soit `(size p).-2.+1`, qui sera prouvablement égal (bien que non convertible) à `(size p).-1`, pour peu que nous nous restreignions aux polynômes non constants :

Rappelons que le degré d'un polynôme `p` non nul est `(size p).-1` et non `size p`.

```
Definition companion_mx (p : {poly R}) :=
  \matrix_(i,j < (size p).-2.+1)
      ((i == j.+1 :> nat)%:R + p`_i *+ ((size p).-2 == j)).
```

Cette définition est parfaitement valide, mais elle ne nous permet pas de définir une matrice diagonale par blocs dont les blocs sont des matrices compagnes (ce dont nous aurons besoin pour définir la forme normale de Frobenius).

En effet, le type de `diag_block_mx` impose à la fonction décrivant les blocs d'avoir le type `forall (n : nat), nat -> 'M_n.+1`, mais la définition `companion_mx` est de type `forall (p : {poly R}), 'M_((size p).-2.+1)`

Pour résoudre ce problème, nous introduisons une définition intermédiaire `companion_mxn` qui relâche la taille de la matrice renvoyée :

```
Definition companion_mxn n (p : {poly R}) :=
  \matrix_(i,j < n) ((i == j.+1 :> nat)%:R + p`_i *+ ((size
      ↪ p).-2 == j)).

Definition companion_mx (p : {poly R}) :=
  companion_mxn (size p).-2.+1 p.
```

Ainsi, la matrice `diag_block_mx s companion_mxn` sera bien typée, et aura les propriétés attendues si `s` contient des tailles de la forme `(size p).-2`. Les lemmes sur les matrices compagnes seront exprimés, eux, sur `companion_mx`.

La forme normale de Frobenius d'une matrice A (à coefficients dans un corps) est la matrice suivante :

$$\begin{bmatrix} \mathcal{C}_{p_1} & & & 0 \\ & \mathcal{C}_{p_2} & & \\ & & \ddots & \\ 0 & & & \mathcal{C}_{p_k} \end{bmatrix}$$

où les polynômes p_i sont les facteurs invariants de la matrice A. Formellement, elle peut être définie de la manière suivante :

```
Definition Frobenius_form n (A : 'M[R]_n) :=
  let sizes :=  [seq (size p).-2 | p : {poly R} <- ↙
      ↳ (invariant_factors A)] in
  let blocks n i := companion_mxn n.+1 (nth 0 ↙
      ↳ (invariant_factors A) i) in
   diag_block_mx sizes blocks.
```

5.3.3 De Smith à Frobenius

Nous allons maintenant montrer que toute matrice sur un corps `F` est semblable à sa forme de Frobenius :

```
Lemma Frobenius n (A : 'M[F]_n.+1) :
  similar A (Frobenius_form A).
```

Le théorème `similar_fundamental` décrit section 5.1 montre que pour prouver ce résultat, il nous suffit d'établir que pour toute matrice `A`, les matrices caractéristiques de `A` et `Frobenius_form A` sont équivalentes.

Nous allons donc partir de la matrice $XI - A$ et, par transitivité de l'équivalence, arriver à la matrice caractéristique de `Frobenius_form A`.

Nous savons que $XI - A$ est équivalente à sa forme normale de Smith qui, en prenant les polynômes diagonaux unitaires, est de la forme :

$$\begin{bmatrix} 1 & & & & \\ & \ddots & & 0 & \\ & & 1 & & \\ & & & p_1 & \\ & 0 & & & \ddots \\ & & & & & p_n \end{bmatrix}$$

où les p_i sont les facteurs invariants de la matrice A.

Permuter les éléments diagonaux de la matrice ne change pas le fait que la matrice ainsi obtenue est équivalente à la matrice $XI - A$. Car les matrices de passage de l'une à l'autre sont des matrices de permutations et donc inversibles. L'énoncé formel de ce résultat se présente de la manière suivante :

```
Lemma similar_diag_mx_seq m n s1 s2 :
  m = n -> size s1 = m -> perm_eq s1 s2 ->
  similar (diag_mx_seq m m s1) (diag_mx_seq n n s2).
```

où `perm_eq s1 s2` exprime le fait que les séquence `s1` et `s2` ont les mêmes éléments mais permutés. Nous pouvons ainsi permuter les éléments diagonaux de la matrice précédente :

$$\begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & 0 & \\ & & 1 & & & & \\ & & & p_1 & & & \\ & & & & \ddots & & \\ & & & & & 1 & \\ & 0 & & & & & \ddots \\ & & & & & & 1 \\ & & & & & & & p_n \end{bmatrix}$$

Si `pi` représente le polynôme p_i, alors le nombre de 1 avant p_i peut s'écrire `(size pi).-2`. Cette matrice peut être vue comme la matrice diagonale par blocs suivante :

$$\left[\begin{array}{cc|cc} 1 & & & \\ & \ddots & & \\ & 1 & & 0 \\ & & p_1 & \\ \hline & & \ddots & \\ & 0 & & 1 \\ & & & \ddots \\ & & & & 1 \\ & & & & & p_n \end{array}\right]$$

Il nous reste maintenant à montrer que cette matrice est équivalente à la matrice caractéristique de la forme normale de Frobenius. Nous utilisons d'abord le fait que la matrice caractéristique d'une matrice diagonale par blocs est la matrice diagonale par blocs des blocs caractéristiques :

```
Lemma char_diag_block_mx s (F : forall n, nat -> 'M[R]_n.+1) :
  s != [::] ->
  char_poly_mx (diag_block_mx s F) =
  diag_block_mx s (fun n i => char_poly_mx (F n i)).
```

Il ne nous reste donc plus qu'à montrer que la matrice précédente est équivalente à :

$$\begin{bmatrix} XI - \mathcal{C}_{p_1} & & & 0 \\ & \ddots & & \\ & & \ddots & \\ 0 & & & XI - \mathcal{C}_{p_n} \end{bmatrix}$$

Pour cela, il suffit de prouver que les matrices sont équivalentes bloc à bloc, c'est-à-dire que pour chaque indice i, la matrice $XI - \mathcal{C}_{p_i}$ est équivalente à :

$$\begin{bmatrix} 1 & & \\ & \ddots & \\ & & 1 \\ & & & p_i \end{bmatrix}$$

Or il se trouve que cette matrice est la forme normale de Smith de la matrice $XI - \mathcal{C}_{p_i}$. Pour montrer ce dernier résultat, nous utilisons le lemme `Smith_gcdr_spec` mentionné dans la section 5.2. En effet, pour tout `k` tel que `k < (size pi).-2`, on peut trouver une sous-matrice de taille `k` de la matrice $XI - \mathcal{C}_{p_i}$ ci-dessous n'ayant que des -1 sur la diagonale :

$$XI - \mathcal{C}_{p_i} = \begin{bmatrix} X & \dots & 0 & 0 & a_0 \\ -1 & \ddots & \vdots & \vdots & a_1 \\ 0 & \ddots & X & \vdots & \vdots \\ \vdots & \ddots & -1 & X & \vdots \\ 0 & \dots & 0 & -1 & X + a_{n-1} \end{bmatrix}$$

C'est-à-dire que pour tout tel `k`, cette matrice a un mineur d'ordre `k` associé à 1 (car $(-1)^k \sim 1$), et donc le pcgd de tous les mineurs d'ordre `k` est lui-même associé à 1. Il est donc possible choisir les `(size pi).-2` premiers éléments diagonaux de la forme normale de Smith de la matrice $XI - \mathcal{C}_{p_i}$ tels qu'ils soient égaux à 1. Pour le dernier élément diagonal, le seul choix possible est le polynôme p_i, car le produit des éléments diagonaux est le déterminant de la forme normale de Smith, et est également le déterminant de la matrice $XI - \mathcal{C}_{p_i}$.

5.4 FORME DE JORDAN, TRIGONALISATION ET DIAGONALISATION

La forme normale de Jordan d'une matrice A est une matrice triangulaire supérieure dont les éléments diagonaux sont les racines du polynôme caractéristique de la matrice A (c'est-à-dire les valeurs propres de A). Pour que cette forme existe, il suffit que le polynôme caractéristique soit scindé. Afin d'assurer cette condition, nous choisissons de travailler sur un corps algébriquement clos `F`.

Nous récapitulons dans un premier temps les éléments de la théorie des corps algébriquement clos [C. Cohen, 2012] de SSReflect que nous utilisons. Nous définissons ensuite la forme normale de Jordan, puis nous montrons comment l'obtenir à partir de celle de Frobenius.

5.4.1 Polynômes à coefficients dans un corps algébriquement clos

Si p est un polynôme à coefficients dans un corps clos tel que

$$p = \prod_{i=1}^{m} (X - \lambda_i)^{\mu_i}$$

alors :

- `root_seq p` est la séquence des λ_i, où chaque λ_i apparaît μ_i fois.
- `root_mu_seq p` est la séquence des paires (μ_i, λ_i).
- `linear_factor_seq p` est la séquence des polynômes $(X - \lambda_i)^{\mu_i}$.
- Si `s` est une séquence de polynômes, alors `root_seq_poly s` est la concaténation de toutes les séquences obtenues lorsque l'on applique `root_mu_seq` sur chacun des polynômes de `s`.

5.4.2 Définitions

On appelle bloc de Jordan la matrice suivante :

$$J(\lambda, n) = \begin{bmatrix} \lambda & 1 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & 1 \\ 0 & \dots & \dots & 0 & \lambda \end{bmatrix}$$

Nous le définissons formellement de la manière suivante :

```
Definition Jordan_block lam n : 'M[F]_n :=
  \matrix_(i,j) (lam *+ (i == j) + (i.+1 == j)%:R).
```

La forme normale de Jordan est une matrice diagonale par blocs composée de blocs de Jordan :

```
Definition Jordan_form  n (A : 'M[R]_n.+1) :=
  let sp := root_seq_poly (invariant_factors A) in
  let sizes := [seq x.1 | x <- sp] in
  let blocks n i :=  Jordan_block (nth (0,0) sp i).2 n.+1 in
  diag_block_mx sizes blocks.
```

Dans la suite nous expliquons le passage de la forme normale de Frobenius à celle de Jordan. Cela permettra de mieux comprendre la définition donnée ci-dessus.

5.4.3 De Frobenius à Jordan

Soit A une matrice à coefficients dans un corps clos `F`. Nous avons déjà prouvé que la matrice A est semblable à sa forme normale de Frobenius. Nous allons montrer que la forme normale de Frobenius de A est semblable à sa forme normale de Jordan. Par transitivité de la similitude, nous aurons ainsi prouvé que toute matrice est semblable à sa forme normale de Jordan :

```
Lemma Jordan n (A : 'M[F]_n.+1) : similar A (Jordan_form A).
```

On peut sans perte de généralité travailler sur un seul bloc de la forme normale de Frobenius. Fixons donc un indice i, et travaillons sur la matrice $\mathcal{C}_{p_i}$ où p_i est le i-ème facteur invariant de A.

Montrons d'abord que si $q = q_1 \dots q_m$ et que les polynômes q_i sont premiers entre eux deux à deux, alors la matrice $\mathcal{C}_q$ est semblable à :

$$\begin{bmatrix} \mathcal{C}_{q_1} & & 0 \\ & \ddots & \\ 0 & & \mathcal{C}_{q_m} \end{bmatrix}$$

Par récurrence sur m il suffit de montrer que $\mathcal{C}_q$ est semblable à :

$$\begin{bmatrix} \mathcal{C}_{q_1} & 0 \\ 0 & \mathcal{C}_{q_2 \dots q_m} \end{bmatrix}$$

Notre première tentative pour prouver l'équivalence ci-dessus a été de donner explicitement la matrice de passage. Cela aboutit, mais c'est une preuve assez longue qui finalement consiste à refaire la preuve de l'algorithme de

Smith. Nous avons finalement opté pour une preuve utilisant le théorème `similar_fundamental` de la section 5.1. Cela nous ramène à prouver une équivalence entre des matrices ne contenant que des formes normales de Smith de matrices compagnes. Pour montrer cette équivalence nous utilisons le lemme `Smith_gcdr_spec` de la même manière que dans la fin de la section 5.3. Nous en déduisons que $q_1 * q_2 \dots q_m = q$ est le seul facteur invariant de la matrice ci-dessus ainsi que de la matrice $\mathcal{C}_q$. Les deux matrices ont les mêmes facteurs invariants, donc sont équivalentes. Ce qui conclut la démonstration de ce résultat.

Le corps `F` étant clos, nous pouvons décomposer le facteur invariant p_i comme suit :

$$p_i = \prod_{j=1}^{m_i} (X - \lambda_{ij})^{\mu_{ij}}$$

où les λ_{ij} sont les racines de p_i et les μ_i leur multiplicité. Le résultat précédent nous permet d'établir que la matrice compagne $\mathcal{C}_{p_i}$ est semblable à :

$$\begin{bmatrix} \mathcal{C}_{(X-\lambda_{i1})^{\mu_{i1}}} & & 0 \\ & \ddots & \\ & & \ddots \\ 0 & & \mathcal{C}_{(X-\lambda_{im_i})^{\mu_{im_i}}} \end{bmatrix}$$

Nous allons maintenant montrer que pour tout λ et n, le bloc de Jordan $J(\lambda, n)$ est semblable à la matrice compagne $\mathcal{C}_{(X-\lambda)^n}$.

Là encore, nous avons d'abord prouvé ce résultat en donnant explicitement la matrice de passage. Mais la vérification est longue et fait intervenir des calculs avec des coefficients binomiaux. Nous avons donc finalement appliqué la même méthode que précédemment, c'est-à-dire utilisé le théorème `similar_fundamental` pour ramener le problème à une équivalence, puis le lemme `Smith_gcdr_spec` pour déterminer les facteurs invariants des matrices. Nous montrons ainsi que le seul facteur invariant de $J(\lambda, n)$ est le polynôme $(X - \lambda)^n$.

Nous pouvons maintenant établir que la matrice $\mathcal{C}_{p_i}$ est semblable à :

$$\begin{bmatrix} J_{i1} & & 0 \\ & \ddots & \\ & & \ddots \\ 0 & & J_{im_i} \end{bmatrix}$$

avec $J_{ij} = J(\lambda_{ij}, \mu_{ij})$.

Donc chaque bloc de la forme normale de Frobenius de A est semblable à une matrice comme celle ci-dessus où les (λ_{ij}, μ_{ij}) sont les paires composées des racines des facteurs invariants et de leur multiplicité. Cela explique la définition de la forme normale de Jordan donnée plus haut, et démontre aussi que la forme normale de Frobenius de A est semblable à la forme normale de Jordan de A.

5.4.4 Diagonalisation

Nous venons de voir que, dans un corps clos, toute matrice est semblable à sa forme normale de Jordan, qui est triangulaire supérieure. Ceci nous donne directement un théorème de trigonalisation. Nous allons voir maintenant dans quelles conditions une matrice est diagonalisable.

Nous avons vu précédemment que la forme normale de Jordan d'une matrice A est composée de blocs de Jordan $J(\lambda, k)$, où λ est une racine d'un facteur invariant de A, et k sa multiplicité. Or k est aussi la taille du bloc $J(\lambda, k)$. Donc si $k = 1$, la forme normale de Jordan est diagonale. Comme les facteurs invariants se divisent successivement, il suffit que le dernier de la liste soit à racines simples pour que tous les facteurs invariants soient à racines simples. Or il se trouve que le dernier facteur invariant est le polynôme minimal de A. Donc il suffit que le polynôme minimal de la matrice A soit à racines simples pour que A soit diagonalisable :

```
Lemma diagonalization n (A : 'M[R]_n.+1) :
  uniq (root_seq (mxminpoly A)) ->
  similar A (diag_mx_seq n.+1 n.+1 (root_seq (char_poly A))).
```

où `uniq` est un prédicat exprimant que la séquence donnée en argument est sans doublons.

5.5 CONCLUSION

Notre contribution centrale dans ce chapitre est un algorithme certifié et exécutable en pratique pour le calcul de la forme normale de Smith d'une matrice sur un anneau euclidien. À partir de ce premier résultat, nous avons apporté trois autres contributions :

- Une preuve formelle du théorème fondamental de similitude sur un corps, qui relie la similitude de deux matrices à l'équivalence de leurs matrices caractéristiques.
- Une preuve formelle que toute matrice sur un corps est semblable à sa forme de Frobenius.
- Une preuve formelle que toute matrice sur un corps algébriquement clos est semblable à sa forme de Jordan.

Ces trois dernières contributions ont été réalisées en collaboration avec Guillaume Cano et ont donné lieu à une publication [G. Cano et M. Dénès, 2013].

Le théorème d'existence de la forme normale de Frobenius a une portée importante car il permet de capturer la structure des endomorphismes d'un espace vectoriel de dimension finie sur un corps quelconque (discret dans notre contexte).

Un autre intérêt de notre développement est de présenter des définitions et des propriétés des matrices diagonales par blocs ou encore des matrices compagnes, qui pourront être réutilisées car ce sont des notions de base en algèbre matricielle. De la même manière, bien que notre développement traite de notions mathématiques relativement élémentaires, il présente la particularité de reposer sur plusieurs formalisations préexistantes, qu'il contribue à éprouver. Citons par exemple les corps algébriquement clos [C. Cohen, 2012] ou encore la formule de Binet-Cauchy et les définitions de sous-matrice et de mineur associées [V. Siles, 2012].

Dans la plupart des cas, ces travaux antérieurs se sont révélés facilement adaptables à notre contexte. La preuve du théorème fondamental de similitude sur un corps notamment, exposée section 5.1, a bénéficié de la grande modularité de la théorie de la division de polynômes fournie par SSReflect.

En revanche, comme indiqué dans la section 5.3, nous pourrions utiliser une définition plus naturelle de matrice diagonale par blocs si une structure intermédiaire d'anneau n'excluant pas l'anneau trivial était fournie par SSReflect. Mais l'ajout d'une telle structure peut avoir des conséquences qui restent à étudier, comme l'allongement de la hiérarchie globale des structures algébriques, ce qui pourrait stresser trop fortement l'implémentation actuelle de Coq. Un autre point améliorable, indépendant du problème précédent, est que notre définition ne peut être itérée. En effet, il n'est pas possible en l'état de définir une matrice diagonale par blocs dont les blocs sont eux-mêmes des matrices diagonales par blocs, l'obstacle technique étant le même que celui décrit à la section 5.3 pour les matrices compagnes.

Il est à noter que parmi les travaux développés dans ce chapitre, seul l'algorithme de calcul de la forme de Smith est exécutable sur des entrées concrètes. Un prolongement naturel de ce travail consiste donc à utiliser les notions définies ici comme spécification pour une implémentation efficace d'un algorithme de calcul de la forme de Frobenius.

Par exemple, celui décrit dans [A. Storjohann, 2001] a une complexité en $\mathcal{O}(n^\omega \log n \log\log n)$ pour une implémentation du produit matriciel en $\mathcal{O}(n^\omega)$ et serait un bon candidat car il est déterministe. Une sous-routine de cet algorithme est connue sous le nom d'algorithme de Keller-Gehrig [W. Keller-Gehrig, 1985] et a son intérêt propre car elle permet de calculer le polynôme caractéristique d'une matrice de taille n en $\mathcal{O}(n^\omega \log n)$. Nous laissons leur étude formelle pour un travail futur.

Troisième partie

APPLICATION À L'HOMOLOGIE

6 MATRICES D'INCIDENCE DES COMPLEXES SIMPLICIAUX

Nous allons maintenant pouvoir appliquer les algorithmes de la partie II au sujet vaste et complexe qu'est la topologie algébrique. Comme nous l'avons évoqué en introduction de cette thèse, ce domaine consiste à aborder des problèmes topologiques en utilisant autant que possible des méthodes « algébriques ». Par exemple, en associant à chaque espace topologique un groupe particulier, d'une manière compatible avec la relation d'homéomorphisme des espaces, on peut ramener l'étude de propriétés portant sur ceux-ci à des énoncés sur les groupes associés, plus faciles à établir et souvent plus calculatoires.

Bien qu'il s'agisse d'un sujet mathématique abstrait, les méthodes de topologie algébrique peuvent être concrètement implémentées dans des systèmes logiciels puis appliquées à différents contextes comme la théorie du codage [J. A. Wood, 1989], la robotique [D. Mackenzie, 2003] ou l'analyse d'images numériques [R. González-Díaz, B. Medrano et al., 2005 ; R. González-Díaz et P. Real, 2011], parfois médicales [F. Ségonne et al., 2003]. Néanmoins, l'utilisation de ces systèmes en pratique pose le problème de leur correction, à quoi répond l'application de méthodes formelles. Dans ce chapitre, nous allons introduire les définitions de base que nous utiliserons tout au long de cette partie, en particulier la structure de complexe simplicial et les matrices d'incidence qui y sont associées. Nous présentons également la formalisation d'un résultat fondamental sur ces matrices, que nous appelons *lemme de nilpotence*.

Les complexes simpliciaux sont des structures topologiques abstraites qui fournissent un bon cadre pour appliquer des méthodes topologiques à l'analyse des images numériques. Intuitivement, un complexe simplicial est une généralisation de la notion de graphe en dimensions supérieures dans le sens où les complexes simpliciaux de dimension 1 ne sont autres que des graphes.

Un problème central dans ce contexte consiste à calculer les groupes d'homologie des complexes simpliciaux. Ces groupes caractérisent à la fois le nombre de composantes connexes et la structure des trous des complexes. Ce type d'informations est utilisé, par exemple, pour déterminer des similarités entre protéines en biologie moléculaire [T. K. Dey et al., 1999].

Ainsi, le système de calcul formel Kenzo [X. Dousson et al., 1998] implémenté en Common Lisp, permet le calcul de ces groupes d'homologie et a, dans certains cas, obtenu des résultats qui n'ont été ni confirmés ni réfutés par aucun autre moyen. La vérification formelle de ces méthodes vise donc à terme à améliorer le degré de confiance dans la connaissance de l'homologie de ces espaces particuliers.

Il y a deux façons différentes de calculer les groupes d'homologie dans Kenzo selon le type d'objet. D'un côté, le calcul des groupes d'homologie d'un *objet fini* est ramené à une problème de matrices d'incidence, comme nous le décrirons. D'un autre côté, dans le cas d'objets *de type non fini*, la théorie de l'homologie effective [J. Rubio et F. Sergeraert, 2006] de Sergeraert, implémentée dans Kenzo, fournit un cadre où cette question peut être traitée. De manière grossière, l'homologie effective relie un objet de type

non fini, X, à un objet de type fini, Y, avec les mêmes groupes d'homologie ; alors le problème de calculer les groupes d'homologie de X est réduit à la diagonalisation des matrices d'incidence de Y.

Les idées de Sergeraert ont été traduites dans des outils d'aide à la preuve avec pour objectif non seulement de formaliser la théorie de l'homologie effective, mais aussi d'appliquer des méthodes formelles à l'étude de Kenzo. Jusqu'à présent, les principaux efforts de formalisation ont été concentrés sur les théorèmes qui fournissent la connexion entre les objets de type non-fini et ceux de types finis ; citons notamment la vérification du lemme fondamental de perturbation dans l'assistant à la preuve Isabelle/HOL [J. Aransay et al., 2008], ou la formalisation en Coq de l'homologie effective des bicomplexes [C. Domínguez et J. Rubio, 2011].

Cependant, jusqu'à maintenant, la formalisation du calcul de l'homologie des groupes d'objets finis n'avait pas été entreprise. Ce chapitre constitue donc un premier pas dans ce sens, et est organisé comme suit. La section 6.1 contient des préliminaires de topologie algébrique. Un schéma de la preuve du théorème principal est présenté dans la section 6.2. Enfin, la formalisation est décrite à la section 6.3.

6.1 PRÉLIMINAIRES MATHÉMATIQUES

Dans cette section, nous fournissons le bagage minimal nécessaire au reste de cette partie. De nombreux ouvrages expliquant les définitions présentées ici et leurs propriétés sont disponibles. Citons notamment [J. R. Munkres, 1984].

La notion d'espace topologique est trop abstraite pour effectuer des calculs. Mais certains de ces espaces, admettant une triangulation, peuvent être représentés par des complexes simpliciaux fournissant une description purement combinatoire. Ces complexes font ainsi le pont entre topologie générale et topologie algébrique. Comme nous allons le voir, la calculabilité de leurs propriétés (comme les groupes d'homologie) ne fait appel qu'à des méthodes élémentaires, notamment d'algèbre linéaire.

Commençons par la terminologie de base. Soit V un ensemble ordonné, appelé *ensemble de sommets*. Un *simplexe (abstrait)* sur V est un sous-ensemble fini de V. Un n-*simplexe (abstrait)* est un simplexe sur V de cardinal égal à $n+1$. Étant donné un simplexe α sur V, nous appelons *faces* les sous-ensembles de α.

Définition 6.1. *Un* complexe simplicial (ordonné abstrait) *sur V est un ensemble de simplexes K sur V stable par sous-parties ; c'est-à-dire que si $\alpha \in K$ alors toutes les faces de α sont également dans K. On note $S_n(K)$ l'ensemble des n-simplexes de K.*

Nous notons un n-simplexe comme un $n+1$-uplet pour indiquer que les sommets sont ordonnés par l'ordre sur V. Cependant, aucune répétition de sommet n'est permise au sein d'un simplexe. Il est possible de définir l'homologie de structures où de telles répétitions sont permises, mais nous ne les traiterons pas.

Exemple 6.1. *Considérons $V = (0,1,2,3,4,5,6)$. Le complexe simplicial représenté par la figure 6.1 est défini mathématiquement comme l'objet :*

$$K = \left\{ \begin{array}{l} \varnothing, (0), (1), (2), (3), (4), (5), (6), (0,1), (0,2), (0,3), (1,2), \\ (1,3), (2,3), (3,4), (4,5), (4,6), (5,6), (0,1,2), (4,5,6) \end{array} \right\}$$

Il faut remarquer que des complexes simpliciaux peuvent être infinis. Par exemple si $V = \mathbb{N}$ et K est $\{(n)\}_{n\in\mathbb{N}} \cup \{(0,n)\}_{n\geq 1}$, le complexe simplicial obtenu peut être vu comme une union infinie de segments.

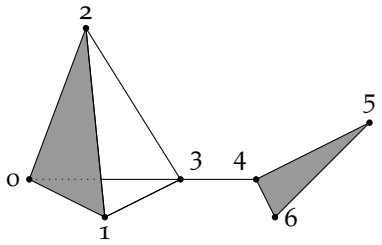


FIGURE 6.1 – Exemple de complexe simplicial, « papillon ».

Définition 6.2. *On appellera* facette *d'un complexe simplicial $\mathcal{K}$ sur V un simplexe maximal pour l'ordre d'inclusion $\subseteq$ parmi les simplexes de $\mathcal{K}$.*

Exemple 6.2. *Les facettes du complexe simplicial de la figure 6.1 sont :*

$$\{(0,3),(1,3),(2,3),(3,4),(0,1,2),(4,5,6)\}$$

Pour construire le complexe simplicial associée à une suite de facettes $\mathcal{F}$, nous générons toutes les faces des simplexes de $\mathcal{F}$. Ensuite, nous obtenons le complexe en effectuant l'union ensembliste de toutes les faces.

Définition 6.3. *Soit $\mathcal{K}$ un complexe simplicial sur V. Soient n et i deux entiers tels que $n \geq 1$ et $0 \leq i \leq n$. Alors l'*opérateur de face ∂_i^n *est l'application linéaire $\partial_i^n : S_n(\mathcal{K}) \to S_{n-1}(\mathcal{K})$ définie par :*

$$\partial_i^n((v_0,\dots,v_n)) = (v_0,\dots,v_{i-1},v_{i+1},\dots,v_n)$$

où le i-ième sommet du simplexe est enlevé, produisant un $(n-1)$-simplexe.

Étant donné un groupe abélien G, nous allons définir les groupes d'homologie à coefficients dans G associé à un complexe simplicial $\mathcal{K}$ en construisant une structure algébrique sur $\mathcal{K}$.

Définition 6.4. *Pour tout entier naturel n, une n-chaîne sur $\mathcal{K}$ est une somme formelle de n-simplexes de $\mathcal{K}$, à coefficients dans G, c'est-à-dire de la forme :*

$$\sum_i g_i \sigma_i, g_i \in G$$

Les n-chaînes forment donc un groupe abélien noté C_n. Lorsque l'on choisit $G = \mathbb{Z}$, C_n est plus précisément le groupe abélien libre engendré par les n-simplexes de $\mathcal{K}$.

Les notions de groupe abélien et de $\mathbb{Z}$-module étant synonymes, nous utiliserons indifféremment l'une ou l'autre. De même, « groupe abélien libre » ou « $\mathbb{Z}$-module libre » sont synonymes.

Intuitivement, certaines chaînes ont un sens géométrique. Par exemple, $-\sigma$ désigne la chaîne σ avec une orientation inversée. Ou encore, le bord d'une n-chaîne σ est une $n-1$-chaîne définie comme la somme alternée des faces de σ :

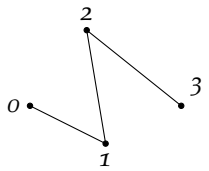
Définition 6.5. *Soit $\mathcal{K}$ un complexe simplicial sur V. L'opérateur de bord en dimension n sur $\mathcal{K}$ est défini par :*

$$d_n := \sum_{i=0}^{n} (-1)^i \partial_i^n$$

C'est-à-dire, en l'appliquant à un n-simplexe $(v_0,\dots,v_n)$:

$$d_n((v_0,\dots,v_n)) := \sum_{i=0}^{n} (-1)^i (v_0,\dots,v_{i-1},v_{i+1},\dots,v_n)$$

Exemple 6.3. *En interprétant une somme de simplexes* $(0,1)+(1,2)+(2,3)$ *comme un chemin, on constate que le bord d'un tel chemin est la différence entre ses sommets d'arrivée et de départ :*



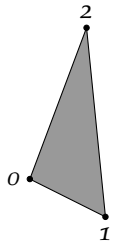
$$\begin{aligned} d_1((0,1)+(1,2)+(2,3)) &= d_1((0,1)) + d_1((1,2)) + d_1((2,3)) \\ &= (1)-(0)+(2)-(1)+(3)-(2) \\ &= (3)-(0) \end{aligned}$$

Si dans cet exemple les sommets (0) et (3) avaient été confondus, nous aurions trouvé un bord nul. Ceci caractérise des chemins partant d'un sommet et revenant au même. De tels chemins sont appelés *cycles*, cette notion se généralisant aux dimensions supérieures.

La lettre Z *provient de l'allemand « Zyklus ».*

Définition 6.6. *Une* n*-chaîne* σ *telle que* $d_n(\sigma) = 0$ *est appelée un* n*-cycle. Le sous-groupe des* n*-cycles est donc* $Z_n = \operatorname{Ker} d_n$.

Exemple 6.4. *Calculons maintenant le bord d'un triangle* $(0,1,2)$ *:*



$$d_2((0,1,2)) = (1,2)-(0,2)+(0,1)$$

On obtient la somme formelle des côtés du triangle, avec des signes correspondant à l'orientation du triangle.

Remarquons que le bord du triangle de l'exemple précédent est lui-même un cycle. Ce fait est général, et nous prouverons dans la section 6.2 que l'opérateur de bord vérifie l'identité :

$$d_n \circ d_{n+1} = 0 \quad \text{pour tout } n \in \mathbb{N}$$

Traditionnellement, les familles $(C_n)_{n\in\mathbb{N}}$ $(d_n)_{n\in\mathbb{N}}$ sont étendues sur $\mathbb{Z}$ en posant $\forall n < 0, C_n = 0$ (où 0 désigne le groupe trivial) et $\forall n \leq 0, d_n = 0$ (où 0 désigne l'application nulle, qui envoie tous les éléments sur le neutre). L'identité précédente est alors valable pour tout $n \in \mathbb{Z}$. La famille $(d_n)_{n\in\mathbb{Z}}$ est appelée *différentielle* et la famille $(C_n, d_n)_{n\in\mathbb{Z}}$ est appelée *complexe de chaînes* $C_*(\mathcal{K})$ *canoniquement associé à* $\mathcal{K}$. Pour clarifier cette notion, prenons un exemple :

Exemple 6.5. *Soit $\mathcal{K}$ le complexe simplicial décrit par la figure 6.1. Le complexe de chaînes $C_*(\mathcal{K})$ canoniquement associé à $\mathcal{K}$ est :*

$$\cdots \to 0 \to C_2(\mathcal{K}) \xrightarrow{d_2} C_1(\mathcal{K}) \xrightarrow{d_1} C_0(\mathcal{K}) \to 0 \to \cdots$$

avec 3 groupes de chaînes à coefficients dans $\mathbb{Z}$ associés :

- $C_0(\mathcal{K})$, *le $\mathbb{Z}$-module libre sur l'ensemble des 0-simplexes (sommets)* $\{(0),(1),(2),(3),(4),(5),(6)\}$.
- $C_1(\mathcal{K})$, *le $\mathbb{Z}$-module libre sur l'ensemble des 1-simplexes (arêtes)* $\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3),(3,4),(4,5),(4,6),(5,6)\}$.
- $C_2(\mathcal{K})$, *le $\mathbb{Z}$-module libre sur l'ensemble des 2-simplexes (triangles)* $\{(0,1,2),(4,5,6)\}$.

Ici encore, 0 désigne le groupe trivial. Les points de suspension indiquent que $\forall n < 0, C_n = 0$ et $\forall n > 2, C_n = 0$. Les flèches partant ou arrivant vers le groupe trivial représentent l'application qui envoie tout élément vers le neutre.

Les éléments de chacun de ces groupes C_p sont des combinaisons linéaires entières des bases correspondantes (ensembles de σ_i), c'est-à-dire des éléments de la forme $\sum \lambda_i \sigma_i$, $\lambda_i \in \mathbb{Z}$.

Étant donné un complexe de chaînes $C_* = (C_n, d_n)_{n\in\mathbb{Z}}$, nous notons $B_n = \operatorname{Im} d_{n+1}$ le sous-groupe des n-bords de C_n. Les identités $d_n \circ d_{n+1} = 0$ sont équivalentes aux relations d'inclusion $B_n \subseteq Z_n$: tout n-bord est un n-cycle mais la réciproque est fausse en général. Ainsi, la définition suivante a un sens :

Définition 6.7. . *Soit $C_* = (C_n, d_n)_{n\in\mathbb{Z}}$ un complexe de chaînes à coefficients dans un groupe abélien G. Pour chaque degré $n \in \mathbb{Z}$, le* n-ième groupe d'homologie *de C_* est défini comme le quotient :*

$$H_n(C_*, G) = \frac{Z_n}{B_n}$$

Dans la suite, nous noterons indifféremment $H_n(C_*, G)$ pour un groupe d'homologie du complexe de chaînes C_* ou $H_n(\mathcal{K}, G)$ pour un groupe d'homologie du complexe de chaînes canoniquement associé au complexe simplicial $\mathcal{K}$.

Exemple 6.6. *Après calcul, les groupes d'homologie à coefficients dans $\mathbb{Z}$ du complexe simplicial de la figure 6.1 sont les suivants :*

$$H_0 \cong \mathbb{Z}$$
$$H_1 \cong \mathbb{Z}^2$$

Géométriquement, H_0 représente les composantes connexes du simplexe. Plus précisément, pour un complexe à k composantes connexes, H_0 sera isomorphe à $\mathbb{Z}^k$.

À partir de la définition précédente, nous pouvons introduire un concept très utile pour le calcul des groupes d'homologie des complexes simpliciaux.

Définition 6.8. *Soit $\mathcal{K}$ un complexe simplicial sur l'ensemble de sommets V et soit n un entier tel que $n \geq 1$. La* n-ième matrice d'incidence de $\mathcal{K}$ *sur l'anneau $\mathbb{Z}$, dénotée par $M_n(\mathcal{K}, \mathbb{Z})$, représente les $(n-1)$-simplexes de $\mathcal{K}$ comme lignes et les n-simplexes de $\mathcal{K}$ comme colonnes. En supposant l'existence d'un ordre sur les simplexes de même dimension (ce que nous ferons à partir d'ici), $M_n(\mathcal{K}, \mathbb{Z})$ est $[a_i^j]$ où i prend des valeurs de 1 au cardinal de $S_{n-1}(\mathcal{K})$, j prend des valeurs de 1 au cardinal de $S_n(\mathcal{K})$ et la valeur de a_i^j est le coefficient du i-ième $(n-1)$-simplexe dans la différentielle du j-ième n-simplexe ; donc a_i^j est une valeur dans $\{0, \pm 1\}$.*

Exemple 6.7. *Si nous imposons un ordre lexicographique sur les simplexes de même dimension du complexe simplicial de la figure 6.1 (si* $v = (a_0, \ldots, a_n)$ *et* $w = (b_0, \ldots, b_n)$ *sont des* n*-simplexes du complexe simplicial, alors* $v < w$ *si* $a_0 < b_0$*, ou* $a_0 = b_0$ *et* $a_1 < b_1$*, ou* $a_0 = b_0$ *et* $a_1 = b_1$ *et* $a_2 < b_2, \ldots$*, ou* $a_0 = b_0, \ldots a_{n-1} = b_{n-1}$ *et* $a_n < b_n$*), alors sa première matrice d'incidence (pour* $n = 1$*) est :*

$$\begin{array}{c} \\ (0) \\ (1) \\ (2) \\ (3) \\ (4) \\ (5) \\ (6) \end{array} \begin{array}{c} \begin{array}{cccccccccc} (0,1) & (0,2) & (0,3) & (1,2) & (1,3) & (2,3) & (3,4) & (4,5) & (4,6) & (5,6) \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \end{array}$$

et sa seconde matrice d'incidence (pour $n = 2$*) est :*

$$\begin{array}{c} \\ (0,1) \\ (0,2) \\ (0,3) \\ (1,2) \\ (1,3) \\ (2,3) \\ (3,4) \\ (4,5) \\ (4,6) \\ (5,6) \end{array} \begin{array}{c} \begin{array}{cc} (0,1,2) & (4,5,6) \end{array} \\ \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} \end{array}$$

La pertinence des matrices d'incidence des complexes simpliciaux tient au fait qu'elle peuvent être utilisées pour calculer les groupes d'homologie via leur forme normale de Smith. En effet, la n-ième matrice d'incidence $M_n(\mathcal{K}, \mathbb{Z})$ d'un complexe simplicial $\mathcal{K}$ représente l'application linéaire d_n. Comme nous l'avons vu au chapitre 5, la mise en formale normale de Smith de cette matrice nous donne des matrices inversibles P et Q telles que la matrice $P \times M_n(\mathcal{K}, \mathbb{Z}) \times Q$ soit de la forme :

$$P \times M_n(\mathcal{K}, \mathbb{Z}) \times Q = \begin{array}{c} \begin{array}{ccccccccc} \sigma_1 & \cdots & \sigma_p & \sigma_{p+1} & \cdots & \cdots & \cdots & \cdots & \sigma_m \end{array} \\ \begin{bmatrix} * & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & \cdots & \cdots & \cdots & \cdots & \cdots & \vdots \\ \vdots & \ddots & * & \ddots & \cdots & \cdots & \cdots & \cdots & \vdots \\ \vdots & \cdots & \ddots & 0 & \ddots & \cdots & \cdots & \cdots & \vdots \\ \vdots & \cdots & \cdots & \ddots & \ddots & \ddots & \cdots & \cdots & \vdots \\ 0 & \cdots & \cdots & \cdots & 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \end{array}$$

où les symboles $*$ dénotent des valeurs non nulles quelconques.

La matrice $P \times M_n(\mathcal{K}, \mathbb{Z}) \times Q$ représente toujours d_n mais dans de nouvelles bases. Nous notons $\sigma_1, \ldots, \sigma_m$ les n-chaînes de la base obtenue de C_n. La famille $\sigma_{p+1}, \ldots, \sigma_m$ est une base du noyau de d_n et on a :

$$\operatorname{Ker} d_n \cong \mathbb{Z}^{m-p}$$

De la même manière, considérons la forme normale de Smith de la $n+1$-ième matrice d'incidence de $\mathcal{K}$:

$$P' \times M_{n+1}(\mathcal{K},\mathbb{Z}) \times Q' = \begin{array}{c} \\ \kappa_1 \\ \vdots \\ \kappa_r \\ \kappa_{r+1} \\ \vdots \\ \kappa_q \\ \kappa_{q+1} \\ \vdots \\ \kappa_m \end{array} \begin{array}{c} \begin{array}{ccccccccc} \kappa_1 & \cdots & \kappa_r & \kappa_{r+1} & \cdots & \kappa_q & \kappa_{q+1} & \cdots & \kappa_m \end{array} \\ \begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & \cdots & \cdots & \cdots & \cdots & \cdots & \vdots \\ \vdots & \ddots & 1 & \ddots & \cdots & \cdots & \cdots & \cdots & \vdots \\ \vdots & \cdots & \ddots & \lambda_1 & \ddots & \cdots & \cdots & \cdots & \vdots \\ \vdots & \cdots & \cdots & \ddots & \ddots & \ddots & \cdots & \cdots & \vdots \\ \vdots & \cdots & \cdots & \cdots & \ddots & \lambda_k & \ddots & \cdots & \vdots \\ \vdots & \cdots & \cdots & \cdots & \cdots & \ddots & 0 & \ddots & \vdots \\ \vdots & \cdots & \cdots & \cdots & \cdots & \cdots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 0 \end{bmatrix} \end{array}$$

avec $\lambda_1 \mid \lambda_2 \mid \cdots \mid \lambda_k$ et $\forall i, \lambda_i > 1$.
Nous obtenons une base $\kappa_1, \ldots, \kappa_m$ de C_n qui nous donne une base de l'image de d_{n+1} :

$$\kappa_1, \ldots, \kappa_r, \lambda_1\kappa_{r+1}, \ldots, \lambda_k\kappa_{r+k}$$

La forme du n-ième groupe d'homologie de $\mathcal{K}$ est ainsi connue :

$$H_n(\mathcal{K},\mathbb{Z}) = \frac{\operatorname{Ker} d_n}{\operatorname{Im} d_{n+1}} \cong \overbrace{\mathbb{Z} \oplus \cdots \oplus \mathbb{Z}}^{m-p-q \text{ termes}} \oplus \frac{\mathbb{Z}}{\lambda_1\mathbb{Z}} \oplus \cdots \oplus \frac{\mathbb{Z}}{\lambda_k\mathbb{Z}}$$

La somme $\mathbb{Z} \oplus \cdots \oplus \mathbb{Z}$ avec $m - p - q$ termes est appelée *partie libre*. Nous la notons également $\mathbb{Z}^{m-p-q}$ car dans le cas de groupes abéliens, somme et produit directs sont isomorphes. Le n-*ième nombre de Betti de* $\mathcal{K}$ (noté $\beta_n(\mathcal{K})$) désigne la dimension $m - p - q$ de la partie libre de $H_n(\mathcal{K},\mathbb{Z})$. La somme $\frac{\mathbb{Z}}{\lambda_1\mathbb{Z}} \oplus \cdots \oplus \frac{\mathbb{Z}}{\lambda_k\mathbb{Z}}$ est la *partie de torsion*. Les $\lambda_1, \ldots, \lambda_k$ sont appelés *coefficients de torsion*.

Un résultat important en algèbre homologique, nommé *théorème des coefficients universels*, exprime les groupes d'homologie à coefficients dans un groupe abélien G quelconque en fonction des groupes d'homologie à coefficients dans $\mathbb{Z}$. L'énoncé général de ce résultat dépasse le cadre de cette thèse. Cependant, un corollaire en est que si l'on calcule les nombres de Betti via l'homologie à coefficients dans un corps de caractéristique p, on trouve les mêmes résultats que pour les groupes d'homologie dans $\mathbb{Z}$, à condition que p ne soit pas un coefficient de torsion dans ces derniers groupes.

Or, nos complexes simpliciaux étant issus d'images numériques, leur partie de torsion est triviale (la torsion n'apparaît que sur des objets qu'on ne peut plonger dans l'espace euclidien $\mathbb{R}^3$, comme le plan projectif réel ou la bouteille de Klein). Les groupes d'homologies sont alors caractérisés par les nombres de Betti, que l'on peut calculer en prenant des coefficients dans n'importe quel corps. Par commodité, nous choisissons le corps $\mathbb{Z}/2\mathbb{Z}$ comme il est souvent d'usage dans le contexte de l'homologie des images numériques [R. González-Díaz, B. Medrano et al., 2005 ; R. González-Díaz et P. Real, 2011].

6.2 LEMME DE NILPOTENCE

Nous allons donc adapter les définitions de l'opérateur de face et des matrices d'incidence associées à des coefficients dans $\mathbb{Z}/2\mathbb{Z}$. En effet, puisque les coefficients de signes opposés sont identifiés, il n'est pas utile de se préoccuper de l'orientation des faces.

Ainsi, dans la suite $\mathcal{K}$ dénotera un complexe simplicial sur un ensemble fini et n un entier tel que $n \geq 1$. Les matrices d'incidences sont maintenant définies par :

Définition 6.9. *La n-ième matrice d'incidence de $\mathcal{K}$ sur le corps $\mathbb{Z}/2\mathbb{Z}$, dénotée par $M_n(\mathcal{K})$, est une matrice de taille $m \times p$, où m est le cardinal de $S_{n-1}(\mathcal{K})$ et p est le cardinal de $S_n(\mathcal{K})$. Ses coefficients $[a_i^j]$ sont 1 si le i-ième $(n-1)$-simplexe est une face du j-ième n-simplexe et 0 sinon.*

Il faut noter que la *n-ième matrice d'incidence de $\mathcal{K}$* sur le corps $\mathbb{Z}/2\mathbb{Z}$ est la valeur absolue de la *n-ième matrice d'incidence de $\mathcal{K}$* sur l'anneau $\mathbb{Z}$.

Exemple 6.8. *Si nous imposons un ordre lexicographique sur les simplexes de même dimension du complexe simplicial décrit à la figure 6.1, alors sa première matrice d'incidence sur le corps $\mathbb{Z}/2\mathbb{Z}$ est :*

$$\begin{array}{c} \\ (0) \\ (1) \\ (2) \\ (3) \\ (4) \\ (5) \\ (6) \end{array} \begin{array}{c} \begin{array}{cccccccccc} (0,1) & (0,2) & (0,3) & (1,2) & (1,3) & (2,3) & (3,4) & (4,5) & (4,6) & (5,6) \end{array} \\ \left[\begin{array}{cccccccccc} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array}\right] \end{array}$$

Comme nous l'avons exposé plus haut, les matrices d'incidence des complexes simpliciaux viennent des différentielles des complexes de chaînes canoniquement associées aux complexes. Ces différentielles satisfont une condition de nilpotence ($d_{n-1} \circ d_n = 0$).

Nous pouvons maintenant énoncer et prouver le lemme suivant qui est l'analogue de cette condition de nilpotence sur les matrices d'incidence.

Théorème 6.1. *Le produit de la n-ième matrices d'incidence de $\mathcal{K}$ sur le corps $\mathbb{Z}/2\mathbb{Z}$, $M_n(\mathcal{K})$, et la $(n+1)$-ième matrice d'incidence de $\mathcal{K}$ sur le même corps $M_{n+1}(\mathcal{K})$ est nul.*

Schéma de la preuve. Soient S_{n-1}, S_n, S_{n+1} respectivement l'ensemble des $(n-1)$-simplexes de $\mathcal{K}$, l'ensemble de ses n-simplexes et l'ensemble de ses $(n+1)$-simplexes. Alors,

$$M_n(\mathcal{K}) = \begin{array}{c} \\ S_{n-1}[1] \\ \vdots \\ S_{n-1}[r2] \end{array} \begin{array}{c} \begin{array}{ccc} S_n[1] & \cdots & S_n[r1] \end{array} \\ \left[\begin{array}{ccc} a_{1,1} & \cdots & a_{1,r1} \\ \vdots & \ddots & \vdots \\ a_{r2,1} & \cdots & a_{r2,r1} \end{array}\right] \end{array}$$

$$M_{n+1}(\mathcal{K}) = \begin{array}{c} \\ S_n[1] \\ \vdots \\ S_n[r1] \end{array} \begin{array}{c} \begin{array}{ccc} S_{n+1}[1] & \cdots & S_{n+1}[r3] \end{array} \\ \left[\begin{array}{ccc} b_{1,1} & \cdots & b_{1,r1} \\ \vdots & \ddots & \vdots \\ b_{r1,1} & \cdots & b_{r1,r3} \end{array}\right] \end{array}$$

où $r1 = \sharp|S_n|$, $r2 = \sharp|S_{n-1}|$ et $r3 = \sharp|S_{n+1}|$. Ainsi,

$$M_n(\mathcal{K}) \times M_{n+1}(\mathcal{K}) = \left[\begin{array}{ccc} c_{1,1} & \cdots & c_{1,r3} \\ \vdots & \ddots & \vdots \\ c_{r2,1} & \cdots & c_{r2,r3} \end{array}\right]$$

Pour prouver que $M_n \times M_{n+1}$ est nul, il suffit de montrer que pour tous i et j tels que $1 \leq i \leq \sharp|S_{n-1}|$ et $1 \leq j \leq \sharp|S_{n+1}|$, alors $c_{i,j} = 0$. Chacun de ces coefficients s'écrit :

$$c_{i,j} = \sum_{1 \leq k \leq r1} a_{i,k} \times b_{k,j}$$

Puisque k énumère les indices des éléments de S_n, nous pouvons écrire :

$$c_{i,j} = \sum_{X \in S_n} F(S_{n-1}[i], X) \times F(X, S_{n+1}[j]) \quad \text{avec } F(Y,Z) = \begin{cases} 1 & \text{si } Y \in dZ \\ 0 & \text{sinon} \end{cases} \tag{6.1}$$

dZ est l'analogue dans notre contexte de la différentielle définie à la section 6.1 et est égal à :

$$dZ = \{Z \setminus \{x\} \mid x \in Z\}$$

Cette somme peut être séparée selon que $X \in \partial S_{n+1}[j]$ ou $X \notin \partial S_{n+1}[j]$.

$$\begin{aligned} c_{i,j} &= \sum_{X \in S_n | X \in \partial S_{n+1}[j]} F(S_{n-1}[i], X) \times 1 \qquad (6.2) \\ &\quad + \sum_{X \in S_n | X \notin \partial S_{n+1}[j]} F(S_{n-1}[i], X) \times 0 \\ &= \sum_{X \in S_n | X \in \partial S_{n+1}[j]} F(S_{n-1}[i], X) \qquad (6.3) \end{aligned}$$

Cette dernière somme est exprimée sur l'image de l'opérateur de face $x \mapsto S_{n+1}[j] \setminus \{x\}$ qui, restreint à $S_{n+1}[j]$, est injectif. Ainsi, on peut réindexer :

$$c_{i,j} = \sum_{x \in S_{n+1}[j]} F(S_{n-1}[i], S_{n+1}[j] \setminus \{x\}) \tag{6.4}$$

Ensuite, cette somme peut également être découpée selon que $x \in S_{n-1}[i]$ ou $x \notin S_{n-1}[i]$.

$$\begin{aligned} c_{i,j} &= \sum_{x \in S_{n+1}[j] | x \in S_{n-1}[i]} F(S_{n-1}[i], S_{n+1}[j] \setminus \{x\}) + \\ &\quad \sum_{x \in S_{n+1}[j] | x \notin S_{n-1}[i]} F(S_{n-1}[i], S_{n+1}[j] \setminus \{x\}) \qquad (6.5) \end{aligned}$$

Notons que si $x \in S_{n-1}[i]$ alors $S_{n-1}[i] \notin S_{n+1}[j] \setminus \{x\}$, donc la première somme ci-dessus est nulle.

$$c_{i,j} = \sum_{x \in S_{n+1}[j] | x \notin S_{n-1}[i]} F(S_{n-1}[i], S_{n+1}[j] \setminus \{x\}) \tag{6.6}$$

Ici, on peut considérer deux cas : $S_{n-1}[i] \notin S_{n+1}[j]$ ou $S_{n-1}[i] \subseteq S_{n+1}[j]$.
Dans le premier cas, nous avons : $\forall x \in S_{n-1}[i], F(S_{n-1}[i], S_{n+1}[j] \setminus \{x\}) = 0$, d'où le résultat.
Dans le second cas, $S_{n-1}[i] \subseteq S_{n+1}[j]$ implique que si $x \notin S_{n-1}[i]$ alors $S_{n-1}[i] \in \partial S_{n+1}[j] \setminus \{x\}$, donc les termes sont tous égaux à 1.

$$\begin{aligned} c_{i,j} &= \sum_{x \in S_{n+1}[j] | x \notin S_{n-1}[i]} 1 \qquad (6.7) \\ &= \sharp|S_{n+1}[j] \setminus S_{n-1}[i]| \\ &= n + 2 - n = 2 = 0 \bmod 2 \end{aligned}$$

6.3 DÉVELOPPEMENT FORMEL

Pour formaliser le théorème 6.1, nous avons encore une fois mis à profit la bibliothèque SSReflect. Plus particulièrement, nous en avons utilisé le développement des types et ensembles finis pour définir les complexes simpliciaux, celui des matrices pour les matrices d'incidence et les groupes cycliques et corps finis pour notre anneau de coefficients. Enfin, le support des grands opérateurs [Y. Bertot et al., 2008] a joué un rôle crucial dans la formalisation de la preuve du lemme de nilpotence.

Tout d'abord, nous définissons les notions liées aux complexes simpliciaux. Les sommets sont représentés par un type fini `V`. Un simplexe est défini comme un ensemble fini de sommets. Ensuite, la définition d'un complexe simplicial comme ensemble de simplexes clos pour l'inclusion est immédiate :

```
Variable V : finType.
Definition simplex := {set V}.
Definition simplicial_complex (c : {set simplex}) :=
  forall x, x \in c -> forall y : simplex, y \subset x ->  y ↙
      ↳ \in c.
```

Puisque nous ne tenons pas compte des signes des coefficients apparaissant dans les matrices d'incidence, nous définissons un opérateur de face comme une différence ensembliste (nous enlevons un sommet d'un simplexe) et le bord comme l'image d'un simplexe par l'opérateur de face.

```
Definition face_op (S : simplex) (x : V) := S :\ x.
Definition boundary (S : simplex) := (face_op S) @: S.
```

Nous prouvons la correction de notre définition du bord en montrant qu'elle est équivalente a une relation de sous-ensemble avec des contraintes de cardinalité :

```
Lemma boundaryP: forall (S : simplex) (B : simplex),
  reflect (B \subset S /\ #|S| = #|B|.+1) (B \in boundary S).
```

L'énoncé `reflect P b` exprime une équivalence entre une proposition `P` et une expression booléenne `b`. Ceci permet de tirer parti de la décidabilité de certaines propositions en faisant des aller-retours entre leur expression logique (utile au raisonnement) et leur équivalent booléen (adapté aux calculs).

Un argument clé pour notre preuve est l'injectivité de l'opérateur de face ci-dessus, que nous établissons comme lemme :

```
Lemma face_op_inj (S : simplex) :
  {in S &, injective (face_op S)}.
```

La notation `{in S &, P}` effectue la localisation de prédicats : si `P` est de la forme `forall x y, Q x y` alors `{in S &, P}` devient simplement le prédicat `forall x y, x \in S -> y \in S -> Q x y`. En ce qui nous concerne, `injective f` désigne `forall x y, f x = f y -> x = y`.

Maintenant, avant de formaliser la définition de la n-ième matrice d'incidence d'un complexe simplicial, nous pouvons définir la notion plus générique de matrice d'incidence de deux ensembles finis de simplexes `Left` (pour les lignes) et `Top` (pour les colonnes).

Représenter une matrice requiert une indexation des simplexes dans `Left` et `Top`. Puisque `Left` et `Top` sont des ensembles finis, ils sont équipés d'une

énumération canonique : `(enum_val Left i)` renvoie le i-ième élément de l'ensemble `Left`. Un coefficient a_{ij} de la matrice d'incidence sera 1 si le i-ième simplexe de `Left` est une face (sous-ensemble) du j-ième simplexe de `Top` et 0 sinon.

Ainsi nous pouvons définir une matrice d'incidence de deux ensembles finis de simplexes comme suit :

L'annotation de type `'F_2` indique que les 0 et 1 apparaissant comme coefficients de la matrice sont les deux éléments de $\mathbb{F}_2$, c'est-à-dire $\mathbb{Z}/2\mathbb{Z}$ en tant que corps.

```
Variables Left Top : {set simplex}.
Definition incidenceMatrix :=
  \matrix_(i < #|Left|, j < #|Top|)
    if enum_val i \in boundary (enum_val j) then 1 else 0 : 'F_2.
```

Dans la définition ci-dessus, il peut être noté que le premier argument de `enum_val` est implicite et déterminé par le contexte. En effet, la notation `i < #|Left|` signifie que le type de `i` est `'I_(#|Left|)`, c'est-à-dire que `i` est un ordinal prenant des valeurs de 0 à `#|Left|`−1, où `#|X|` dénote le cardinal de l'ensemble `X`. Avec cette information de type, le système expanse `enum_val i` en `enum_val Left i`, résolvant ainsi l'ambiguïté (et de même pour `j`).

Nous définissons maintenant la `n`-ième matrice d'incidence d'un complexe simplicial `c`, en instanciant `Left` à l'ensemble des `n`−1-simplexes (de `c`) et `Top` à l'ensemble des `n`-simplexes. Il est à noter que `n` doit être non nul.

```
Section nth_incidence_matrix.
Variable c : {set simplex}.
Variable n : nat.
Definition n_1_simplices := [set x \in c | #|x| == n].
Definition n_simplices := [set x \in c | #|x| == n+1].
Definition incidence_matrix_n :=
  incidenceMatrix n_1_simplices n_simplices.
End nth_incidence_matrix.
```

Nous avons maintenant tous les ingrédients pour énoncer le théorème 6.1 :

```
Theorem incidence_matrices_sc_product:
  forall (V:finType) (n:nat) (sc: {set (simplex V)}),
    simplicial_complex sc ->
      (incidence_mx_n sc n) *m (incidence_mx_n sc (n.+1)) = 0.
```

La preuve formelle du théorème 6.1 suit le schéma présenté à la section 6.2. Une part importante de la preuve est dédiée au travail sur les sommes, pour lequel la bibliothèque de grands opérateurs a joué un rôle clé.

Par exemple, le premier découpage de somme (équation (6.2)) est réalisé par :

```
rewrite (bigID (mem (boundary (enum_val j)))).
```

où `j` appartient à S_{n+1}.

Le lemme `bigID` énonce qu'une opération itérée reposant sur un opérateur monoïdal commutatif peut être découpée :

$$\sum_{i \in r | P_i} F_i = \sum_{i \in r | P_i \wedge a_i} F_i + \sum_{i \in r | P_i \wedge \sim a_i} F_i$$

Il est également possible de découper une somme (équation (6.5)) et en même temps réécrire la première somme résultante à 0 :

```
rewrite (bigID (mem (enum_val i))) big1.
```

Le lemme `big1` énonce que lorsqu'un opérateur monoïdal est itéré sur des éléments tous égaux à l'élément neutre, alors le résultat est également le neutre :

$$\sum_{i \in r | P_i} 0 = 0$$

En conséquence, après la dernière tactique, le système demandera une preuve que tous les termes de la première somme résultante sont égaux à 0. `big1` est appliqué pour obtenir les équations (6.3) et (6.6) de la section 6.2.

Notre preuve repose sur deux réindexations principales : des ordinaux aux n-simplexes (6.1) et ensuite des simplexes aux sommets (6.4). Pour réaliser cette première réindexation, le script a la forme suivante :

```
rewrite (reindex_onto (enum_rank_in Hx0) enum_val) ; last first.
  by move=> x _ ; exact: enum_valK_in.
```

où :

- `Hx0` est une preuve qu'il existe au moins un n-simplexe ;
- `enum_rank_in` énumère les n-simplexes puisque `Hx0` assure qu'il y en a au moins un ;
- `enum_val` énumère les ordinaux sur lesquels la somme est exprimée ;
- `reindex_onto` réindexe des ordinaux aux n-simplexes, étant donnée une bijection entre les deux ensembles. En effet, la seconde ligne du script prouve que `enum_val ∘ enum_rank_in = Id`.

La seconde réindexation est basée sur l'injectivité de l'opérateur de face :

```
rewrite big_imset ; last exact: face_op_inj2.
```

Réécrire avec le lemme `big_imset` déclenche la vérification que la somme est exprimée sur l'image d'un ensemble par une fonction. Dans notre cas, le système infère automatiquement que cette fonction est l'opérateur de face `face_op`, et demande alors une preuve de son injectivité.

Le lemme `eq_big` et ses variantes `eq_bigl` et `eq_bigr` permettent de réécrire le prédicat ou l'opérande d'une opération itérée. Il est appliqué en particulier pour obtenir l'équation (6.7) de la section 6.2 :

```
rewrite (eq_bigr (fun _ => 1)).
```

Le système demandera bien sûr une preuve que l'opérande est égal à 1. Il réécrira alors l'expression vers une somme constante, autorisant l'utilisation du lemme `bigconst` pour la remplacer par un produit (cardinal de l'ensemble itéré par la valeur constante).

Des arguments d'arithmétique élémentaires sur les cardinaux complètent alors la preuve.

6.4 CONCLUSION

Notre contribution développée dans ce chapitre est une preuve formelle du lemme de nilpotence pour les complexes simpliciaux abstraits. Plus particulièrement, nous nous sommes intéressés à leurs matrices d'incidence sur le

corps $\mathbb{Z}/2\mathbb{Z}$. Ce lemme sera utilisé au chapitre 7 pour établir une formule calculant la dimension des groupes d'homologie.

Ce travail a été mené en collaboration avec Jónathan Heras, María Poza et Laurence Rideau, et a fait l'objet d'une publication [J. HERAS, M. POZA et al., 2011].

Une extension possible de ce résultat est de le vérifier formellement pour des matrices d'incidence sur $\mathbb{Z}$. Les deux modifications principales à apporter à la preuve sont les suivantes :

- L'opérateur de bord doit tenir compte des signes des coefficients, et donc ne plus renvoyer un ensemble mais une somme formelle de simplexes.
- Dans l'ultime étape de la preuve, il ne suffit plus de montrer que le nombre de simplexes intervenant dans la somme est pair, mais il faut établir que leurs signes se compensent.

Après avoir présenté ici la formalisation des complexes simpliciaux et de leurs matrices d'incidence, nous allons pouvoir définir et calculer les groupes d'homologie issus d'images numériques. C'est l'objet du chapitre 7.

7 HOMOLOGIE DES IMAGES NUMÉRIQUES

Nous allons prolonger ici le résultat du chapitre précédent en l'appliquant au calcul certifié de groupes d'homologie issus d'images numériques. Parmi les nombreuses références disponibles sur le traitement de tels groupes d'homologie, citons [D. Ziou et M. Allili, 2002].

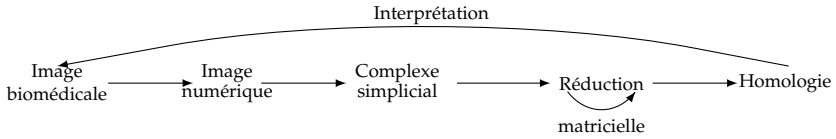


Figure 7.1 – Calcul de l'homologie d'une image numérique

La figure 7.1 schématise le processus que nous souhaitons vérifier. Un complexe simplicial est obtenu, au moyen d'une triangulation, à partir d'une image monochromatique. Ensuite, à partir du complexe simplicial, sont construites les matrices d'adjacence décrites au chapitre 6, et enfin l'homologie peut être calculée. Nous travaillons toujours avec des coefficients dans le corps $\mathbb{Z}/2\mathbb{Z}$ comme c'est usuellement le cas avec des images 2D ou 3D. Il nous suffira alors de calculer la *dimension* des groupes d'homologie (en tant qu'espaces vectoriels), au moyen d'un algorithme calculant le rang d'une matrice.

La suite de ce chapitre est organisée comme suit. La section 7.1 décrit le processus de triangulation. Puis, les groupes d'homologie et leur calcul sont formalisés dans la section 7.2. Ces calculs peuvent être facilités par un processus réduisant la taille des matrices d'incidence tout en conservant l'homologie, que nous présentons dans la section 7.4. Enfin, l'architecture développée sera appliquée à un cas pratique : le comptage des synapses d'une image biomédicale (section 7.3).

7.1 TRIANGULATION

Plusieurs méthodes permettent de construire un complexe simplicial à partir d'une image numérique [R. Ayala et al., 2003]. Nous choisissons d'en décrire une en particulier, qui génère à partir d'une image un ensemble de facettes. Comme nous l'avons expliqué au chapitre 6, ces facettes définissent de manière unique un complexe simplicial.

Nous travaillons sur des images monochromatiques en dimension 2, qui peuvent donc être représentées par des tableaux bidimensionnels de 0 (les pixels blancs) et de 1 (les pixels noirs).

Soit $\mathcal{I}$ une image ainsi représentée. Soit $V = \mathbb{N}^2$ l'ensemble des sommets. Soient $p = (a, b)$ les coordonnées d'un pixel noir de $\mathcal{I}$. Pour chaque p nous définissons deux 2-simplexes qui sont deux facettes du complexe simplicial associé à $\mathcal{I}$: Pour chaque $p = (a, b)$ nous prenons les facettes suivantes : $((a, b), (a + 1, b), (a + 1, b + 1))$ et $((a, b), (a, b + 1), (a + 1, b + 1))$. Si nous répétons le processus pour les coordonnées de tous les pixels noirs de $\mathcal{I}$, nous

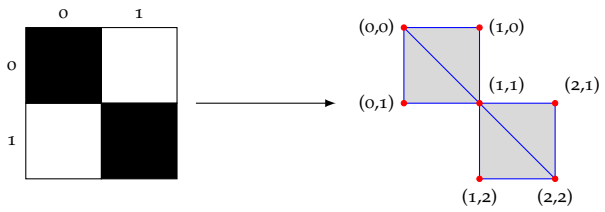


FIGURE 7.2 – Une image numérique et son complexe simplicial associé

obtenons les facettes du complexe simplicial associé à $\mathcal{I}$, que nous notons $\mathcal{K}_{\mathcal{I}}$.

Exemple 7.1. *Considérons l'image représentée par la figure 7.2. Cette image peut être encodée par le tableau à 2 dimensions :*

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Les pixels noirs ont pour coordonnées $(0,0)$ *et* $(1,1)$*, donc le processus de triangulation expliqué ci-dessus donne les facettes suivantes :*

$$((0,0),(1,0),(1,1)),\ ((0,0),(0,1),(1,1)),\ ((1,1),(2,1),(2,2)),\ ((1,1),(1,2),(2,2))$$

La méthode que nous avons présentée associe un complexe simplicial à toute image bidimensionnelle, mais ce processus peut être généralisé en dimensions supérieures [D. ORDEN et F. SANTOS, 2003].

D'un point de vue formel, nous définissons tout d'abord les deux triangles associés à un pixel :

Les variables `m` *et* `n` *désignent respectivement le nombre de lignes et de colonnes de l'image.*

```
Definition ultriangle (i : 'I_m.+1) (j : 'I_n.+1) :=
  [::(i, j) ; (inord i.+1, j) ; (inord i.+1, inord j.+1)].

Definition drtriangle (i : 'I_m.+1) (j : 'I_n.+1) :=
  [::(i, j) ; (i, inord j.+1) ; (inord i.+1, inord j.+1)].
```

Puis, nous écrivons une fonction parcourant une ligne et générant des triangles pour tous les pixels noirs rencontrés :

```
Definition row_facets init (x : 'I_m * (seq bool)) :=
  let r := foldl (fun acc b => let:(s,j) := acc in
    let i := x.1 in
    let j := inord j in
    if b then ([::ultriangle i j, drtriangle i j & s],j.+1)
    else (s,j.+1)) (init,0) x.2
  in r.1.
```

Enfin, la fonction `facets` suivante itère `row_facets` sur toutes les lignes pour générer l'ensemble des facettes du complexe simplicial associé à une image :

```
Definition facets (s : seq (seq bool)) :=
  foldl row_facets [::] (zip (ord_enum m) s).
```

7.2 GROUPES D'HOMOLOGIE

Nous nous concentrons maintenant sur la vérification du calcul des groupes d'homologie à partir d'un complexe simplicial et de ses matrices d'adjacence. Nous définissons pour ce faire en Coq la notion de groupe d'homologie. Soit K un corps, V_1, V_2, V_3 des K-espaces vectoriels et $f : V_1 \to V_2$, $g : V_2 \to V_3$ des applications linéaires ; alors l'homologie de f, g est le quotient du noyau de g par l'image de f :

```
Variable (K : fieldType) (V1 V2 V3 : vectType K)
         (f : linearApp V1 V2) (g : linearApp V2 V3).
Definition Homology := (lker g) :\: (limg f).
```

Puisque nous travaillons sur un corps, il nous suffit de calculer la dimension de ce groupe d'homologie (vu comme espace vectoriel), que l'on appelle *nombre de Betti* :

```
Definition Betti := \dim Homology.
```

Néanmoins, nous ne travaillons habituellement pas avec des applications linéaires lorsqu'il s'agit de calculer l'homologie, mais avec des matrices représentant ces applications. En particulier, comme nous travaillons sur un corps K, étant données deux matrices `M` et `N` à coefficients dans ce corps, de tailles respectives `v1` × `v2` et `v2` × `v3` telles que leur produit est nul, la dimension du groupe d'homologie correspondant est donnée par la définition suivante :

```
Definition mxBetti (M : 'M[K]_(v1,v2)) (N : 'M[K]_(v2,v3)) :=
  v2 - \rank M - \rank N.
```

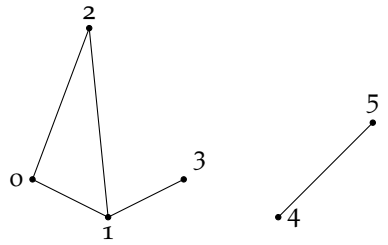
La correction de `mxBetti` peut être montrée en prouvant qu'étant données deux matrices `M` et `N` telles que leur produit est nul (`M *m N = 0`), le résultat obtenu en utilisant `mxBetti` est bien le même qu'en utilisant la définition `Betti` précédente sur les applications linéaires représentées par `M` et `N`, notées respectivement `LinearApp M` et `LinearApp N`. C'est ce qu'exprime le lemme suivant :

```
Lemma mxBettiE : M *m N = 0 ->
  mxBetti M N = Betti (LinearApp M) (LinearApp N).
```

Pour obtenir une définition qui nous permette de calculer ces nombres de Betti sur des entrées concrètes, nous utilisons la fonction `elim_rank` définie au chapitre 4, dont la correction est bien prouvée par rapport à la fonction `\rank` de SSReflect. Nous obtenons ainsi un programme exécutable pour les calculs d'homologie, dont la correction a été vérifiée en Coq :

```
Definition elim_Betti M N :=
  v2 - elim_rank M - elim_rank N.
```

Montrons maintenant sur un exemple comment le calcul des groupes d'homologie peut être concrètement effectué. Considérons le complexe simplicial de la partie haute de la figure 7.3. Si nous ordonnons lexicographiquement les simplexes de même dimension de ce complexe, la matrice d'incidence en dimension 1 est celle présentée sur la partie basse de la figure 7.3. Notons que les autres matrices d'incidences sont vides, en particulier nous ne considérons pas l'ensemble vide comme un élément de dimension −1.



	(0,1)	(0,2)	(1,2)	(1,3)	(4,5)
(0)	1	1	0	0	0
(1)	1	0	1	1	0
(2)	0	1	1	0	0
(3)	0	0	0	1	0
(4)	0	0	0	0	1
(5)	0	0	0	0	1

FIGURE 7.3 – Un complexe simplicial et sa matrice d'incidence

La procédure pour le calcul d'homologie du complexe simplicial de la figure 7.3 (en dimensions 0 et 1) se déroule de la manière suivante. Tout d'abord, nous définissons les matrices d'incidence :

```
Definition d0 := [::].
Definition d1 := [::[::1;1;0;0;0];
                    [::1;0;1;1;0];
                    [::0;1;1;0;0];
                    [::0;0;0;1;0];
                    [::0;0;0;0;1];
                    [::0;0;0;0;1]].
Definition d2 := [::].
```

Nous pouvons alors directement évaluer les nombres de Betti :[20]

```
Eval compute in elim_Betti 0 6 5 d0 d1.
Eval compute in elim_Betti 6 5 0 d1 d2.
```

[20] *Sur des exemples de taille importante, nous utilisons la stratégie de calcul* `native_compute` *développée au chapitre 1.*

Nous obtenons alors respectivement 2 et 1. L'interprétation de ces résultats est que le complexe simplicial de la figure 7.3 a 2 composantes connexes et 1 trou dans le plan.

De la même façon, nous pouvons calculer l'homologie à partir des matrices d'incidence associées à n'importe quel complexe simplicial généré à partir d'une image numérique. Afin d'évaluer la généralité et les performances de notre approche, nous l'appliquons dans la prochaine section à des images issues d'un contexte biomédical.

7.3 APPLICATION AUX IMAGES MÉDICALES

Le développement de preuves formelles pose systématiquement la question de l'applicabilité des définitions choisies à un problème concret. Pour nous assurer que c'est bien le cas du travail présenté dans ce chapitre, nous avons choisi de valider nos choix de conceptions sur un problème apparu dans une application industrielle, à savoir le comptage des synapses sur des images numériques biomédicales.

Les synapses sont les points de connexion entre neurones. Leur importance tient au fait qu'ils sont vecteurs des capacités calculatoires du cerveau. En particulier, l'évolution du nombre de synapses est un indicateur important dans l'étude de certaines maladies neurologiques, comme Alzheimer [D. J. SELKOE, 2002]. Il est donc important de disposer d'une méthode automatique fiable et efficace pour les compter.

Afin d'implémenter une telle méthode, une extension pour l'environnement de traitement et d'analyse d'images ImageJ [W. RASBAND, 2003] a été développée à l'université de La Rioja. Cette extension, appelée SynapCountJ [G. MATA, 2011] applique aux images des neurones une procédure qui peut être divisée en deux étapes.

D'abord, à partir de trois images d'un neurone, à savoir le neurone avec deux marqueurs d'anticorps différents et la structure du neurone, SynapCountJ produit une bitmap dont les composantes connexes représentent les synapses, comme sur la figure 7.4. Ensuite, la seconde étape consiste à compter les composantes connexes de la bitmap.

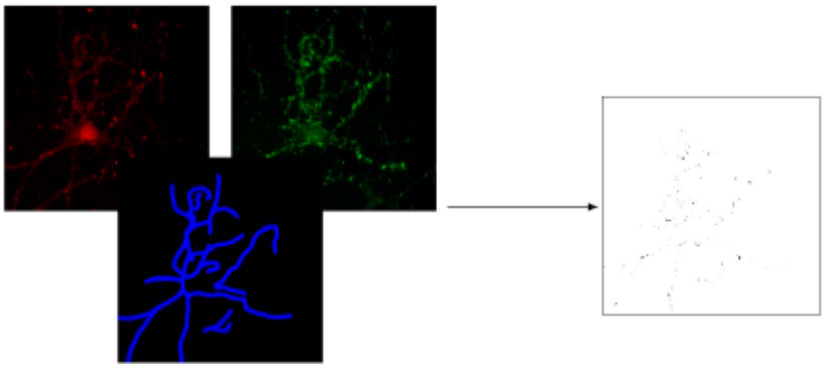


FIGURE 7.4 – Exemple d'extraction de synapses par SynapCountJ

Nous nous intéressons dans la suite à la vérification formelle de cette seconde étape, consistant à compter les composantes connexes. Le calcul de groupes d'homologie est un outil surdimensionné pour atteindre cet objectif, car il existe des méthodes plus élémentaires. Cependant, ce comptage constitue tout de même un test intéressant pour notre développement, que nous appliquons au cas particulier du calcul de l'homologie en dimension 0.

Mais en pratique, la taille des images obtenues en sortie de l'extension SynapCountJ est trop importante pour pouvoir être traitées en temps raisonnable à l'intérieur de Coq. Pour contourner ce problème, il est possible de définir des algorithmes réduisant la taille d'une image tout en préservant l'homologie associée. C'est l'objet de la prochaine section.

7.4 RÉDUCTION DES MATRICES D'INCIDENCE

La méthode que nous allons décrire pour le processus de réduction est basée sur la *théorie de Morse discrète* [R. FORMAN, 1998]. Le cadre algébrique exact dans lequel nous travaillons est décrit dans [A. ROMERO et F. SERGERAERT, 2010].

Pour schématiser, l'objectif de la théorie de Morse discrète consiste à trouver des « effondrements simpliciaux » qui transforment un complexe simplicial $\mathcal{K}$ en un autre plus petit en conservant ses propriétés homologiques. Dans ce contexte, l'outil important est le concept de *champ de vecteurs discret*

admissible. Commençons par définir la notion de champ de vecteurs associé à une matrice d'incidence d'un complexe simplicial :

Définition 7.1. *Soit $\mathcal{K}$ un complexe simplicial et $M_k(\mathcal{K})$ sa k-ième matrice d'incidence, de taille $m \times n$. Un champ de vecteurs discret pour cette matrice est un ensemble de paires d'entiers $\{(a_i, b_i)_{i\in I}\}$ satisfaisant les conditions suivantes :*

1. $\forall i, 1 \leq i \leq m$ *et* $1 \leq i \leq n$
2. $M_k(a_i, b_i) = \pm 1$
3. *Les a_i (resp. b_i) sont deux-à-deux distincts.*
   C'est-à-dire : $\forall i\, \forall j, a_i = a_j \implies i = j$ *(resp.* $\forall i\, \forall j, b_i = b_j \implies i = j$*)*

Dans notre développement formel, nous définissons les champs de vecteurs discrets comme des séquences de paires d'ordinaux, ce qui permet d'assurer la condition 1 vue plus haut :

```
Definition vectorfield := seq ('I_m * 'I_n).
```

Les deux autres propriétés sont exprimées formellement par le prédicat suivant :

L'expression booléenne `all P s` *est vraie si tous les éléments de la séquence* `s` *vérifient le prédicat* `P`*. En outre,* `uniq s` *exprime que la séquence* `s` *est sans répétition.*

```
Definition dvf (vf : vectorfield) (M : 'M_(m,n)) :=
  all [pred p | M p.1 p.2 == 1] vf && uniq (map (@fst _ _) vf)
  && uniq (map (@snd _ _) vf).
```

L'intérêt de ce concept est qu'un champ de vecteurs discret correspond à une sélection de simplexes qui peuvent être effacés du complexe (nous ne rentrons pas dans les détails de ce processus qui sont décrits dans [A. Romero et F. Sergeraert, 2010]). Pour garantir que cette transformation ne change pas l'homologie, il suffit que le champ de vecteurs discret satisfasse la condition d'admissibilité suivante :

Définition 7.2. *Un champ de vecteurs $V = \{(a_i, b_i)_{i\in I}\}$ associé à une matrice M est admissible si la relation $>$ suivante est une relation d'ordre (strict) partielle : Pour $a \neq a'$, on pose $a > a'$ si $(a, b) \in V$ et $M_{a',b} \neq 0$.*

Cette définition exclue les champs de vecteurs contenant des boucles, qui ne pourraient pas être enlevés sans changer l'homologie. Dans notre développement formel, nous définissons la notion d'admissibilité en la paramétrant par une relation, qui sera construite en même temps que le champ de vecteurs. Nous assurons l'absence de boucle en vérifiant qu'aucun élément n'est en relation avec lui-même par la clôture transitive de la relation passée en argument :

La fonction `connect` *prend en argument une relation et construit sa clôture transitive. Le symbole* `~~` *représente la négation booléenne.*

```
Definition admissible (vf : vectorfield) (M : 'M_(m,n)) (ords : ↙
    ↳ rel 'I_m) :=
  dvf vf M
  && all [pred i | ~~ (connect ords i i)] (map (@fst _ _) vf).
```

La relation qui nous intéresse pour l'admissibilité est définie comme suit :

```
Definition gen_orders (M : 'M_(m,n)) i j :=
  [rel x y | (x != i) && (y == i) && (M x j != 0)].
```

La construction d'un champ de vecteurs discret à partir d'une matrice, et de la relation d'admissibilité associée est réalisée par la fonction qui suit :

```
Fixpoint gen_adm_dvf_rec M vf (ords : rel _) k :=
  if k is l.+1 then
    let P := [pred ij | admissible (ij::vf) M (relU ords ↙
        ↳ (gen_orders M ij.1 ij.2))] in
    if pick P is Some (i,j) then
      gen_adm_dvf_rec M ((i,j)::vf) (relU ords (gen_orders M i ↙
          ↳ j)) l
    else gen_adm_dvf_rec M vf ords l
  else (vf, ords).
```

L'opérateur `relU` *réalise l'union de deux relations passées en argument.*

Enfin, la fonction `gen_adm_dvf_rec` est appelée initialement avec un champ de vecteurs discret et une relation d'admissibilité vides :

```
Definition gen_adm_dvf M :=
  gen_adm_dvf_rec M [::] [rel x y | false] (minn m n).
```

Le lemme suivant exprime que le champ de vecteurs renvoyé par la fonction `gen_adm_dvf` satisfait la condition d'admissibilité pour la relation qui est renvoyé avec lui :

```
Lemma admissible_gen_adm_dvf m n (M : 'M[R]_(m,n)) :
  let (vf,ords) := gen_adm_dvf M in admissible vf M ords.
```

Ce lemme n'apporte que peu de garanties : la relation pourrait rester vide, et le champ de vecteurs discret avoir une forme quelconque. Mais il permet de vérifier que nos définitions ont un minimum de cohérence entre elles.

L'intérêt de cette technique de réduction est que plus le champ de vecteurs discret admissible construit est gros, plus la taille de la matrice en sortie sera petite par rapport à celle en entrée. Il faut noter cependant que l'algorithme que nous avons décrit ne spécifie aucune stratégie (l'opérateur `pick` est utilisé). Il n'est pas destiné à être exécuté, mais montre que la construction d'un tel champ de vecteurs est possible.

En pratique, María Poza a utilisé les mêmes définitions pour implémenter dans Coq un autre algorithme, exécutable celui-ci [M. Poza López de Echazarreta, 2013]. Les résultats rapportés sont satisfaisant car ils montrent que le processus de réduction permet de calculer des groupes d'homologies sur des images médicales qui demandaient trop de ressources sans réduction.

7.5 PERFORMANCES

Pour évaluer les performances de l'ensemble de notre chaîne certifiée, depuis l'image numérique jusqu'au calcul des groupes d'homologie, nous remarquons que la complexité est dominée par le calcul du rang des matrices d'incidence. Nous nous concentrons ici sur le premier groupe d'homologie (H_0), mais le calcul de H_1 donne des performances très similaires.

Notre étude du chapitre 4 a montré que notre algorithme de calcul du rang a un comportement général en $\mathcal{O}(n^3)$, pour une matrice carrée de taille n. En fait, il est possible d'être plus précis : pour une matrice de taille $m \times n$ et de rang r, notre algorithme a une complexité $\mathcal{O}(mnr)$.

Dans le cas du calcul de H_0, la matrice auquel le calcul de rang est appliqué a m lignes et n colonnes, où m est le nombre de sommets et n le nombre d'arêtes du complexe simplicial issu de la triangulation de l'image

de départ. Or, le nombre de triangles d'où sont issus ces sommets et ces arêtes dépend du nombre de pixels noirs de l'image (et non directement de sa taille).

Nous choisissons donc de mesurer le temps de calcul total (comprenant la triangulation, le calcul des matrices d'incidences et celui de leur rang) en faisant varier le nombre de pixels noirs. Nous répétons cette expérience sur trois jeux de tests qui correspondent à différentes densités de pixels noirs. Ainsi, la figure 7.5 montre les temps obtenus pour une image entièrement noire (« Image pleine »), une image dont un pixel sur deux en moyenne est noir (« Densité 1/2 ») et enfin une image dont seulement un pixel sur seize est noir. Dans tous les cas, les images sont générées aléatoirement.

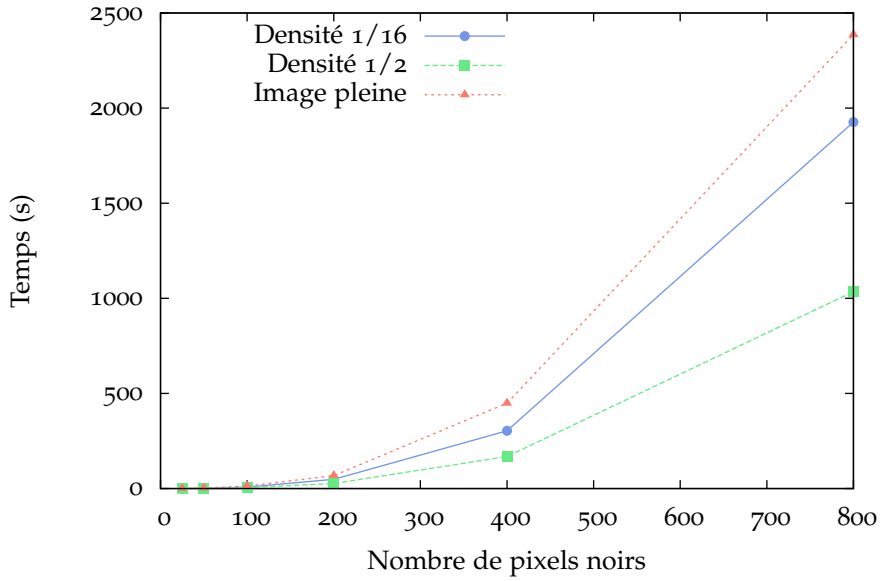


FIGURE 7.5 – Temps de calcul de la dimension du premier groupe d'homologie (H_0) d'images numériques générées aléatoirement.

On observe que des densités de pixels noirs fortes ou faibles entraînent des calculs plus coûteux qu'une densité moyenne (1/2). Ceci est attendu : lorsque la densité est très faible, les triangles sont disjoints et le nombre de sommets et d'arêtes du complexe est maximal. Les matrices d'incidence sont donc de plus grande taille. Quand au contraire la densité est très forte, les matrices d'incidences sont plus petites mais leur rang augmente. Dans le cas extrême de l'image pleine, la première matrice d'incidence est de rang maximum. La complexité de notre algorithme étant sensible au rang, le calcul est plus coûteux.

La table 7.1 reprend les données de la figure 7.5 sous forme numérique. Les performances que nous obtenons sont satisfaisantes car il s'agit pour nous essentiellement d'un exercice d'application des techniques vues dans cette thèse. Cependant, on voit que les temps de calcul augmentent très rapidement avec le nombre de pixels noirs dans l'image d'entrée. Ceci s'explique en partie par la croissance rapide du nombre de simplexes après triangulation de l'image. L'incorporation d'un algorithme de réduction tel que décrit à la section 7.4 permettrait de limiter fortement ce phénomène. Il serait également possible d'aller plus loin en améliorant la représentation des complexes simpliciaux.

Pixels	Temps (s)		
	Densité 1/16	Densité 1/2	Image pleine
25	0,15	0,13	0,52
50	0,9	0,56	2,54
100	7,37	4,15	13,16
200	48,46	26,41	68,12
400	303,64	168,39	448,86
800	1926,13	1034,78	2386,11

TABLE 7.1 – Temps de calcul de la dimension du premier groupe d'homologie (H_0) d'images numériques générées aléatoirement.

7.6 CONCLUSION

Dans ce chapitre, nous avons présenté une application concrète de techniques de topologie algébrique vérifiées formellement à l'étude d'images numériques. En particulier, le traitement certifié d'images biomédicales faisait partie des motivations initiales du projet FORMATH, et nous paraît être une application originale des méthodes formelles.

Nos contributions telles que présentées dans ce chapitre sont :

- Une implémentation en Coq d'une procédure de triangularisation qui génère un complexe simplicial à partir d'une image numériques monochromes.
- Un algorithme vérifié formellement et exécutable pour le calcul des nombres de Betti qui, dans le cas des groupes d'homologie sur $\mathbb{Z}/2\mathbb{Z}$, caractérisent ces groupes.
- Une spécification d'un algorithme de réduction des matrices d'incidence des complexes simpliciaux.

Ce travail est issu d'une collaboration avec Jónathan Heras, Gadea Mata, Anders Mörtberg, María Poza et Vincent Siles ayant fait l'objet d'une publication [J. HERAS, M. DÉNÈS et al., 2012].

Comme pour le chapitre 6, un prolongement important de nos algorithmes de calcul de groupes d'homologie est leur extension à $\mathbb{Z}$. Dans ce contexte, un groupe d'homologie n'est plus simplement caractérisé par le rang des matrices d'incidence, mais par leur forme normale de Smith. Au moment de l'écriture de cette thèse, un travail est en cours, en collaboration avec Anders Mörtberg et Cyril Cohen, pour réutiliser l'algorithme de calcul de la forme normale de Smith présenté au chapitre 5 à cette fin, en vérifiant formellement l'ensemble de la chaîne.

Un autre point d'amélioration possible est l'utilisation d'une représentation creuse pour les matrices d'incidence, notamment lors des calculs de rang. Nous pensons qu'une amélioration très franche des performances pourrait ainsi être réalisée, car les matrices d'incidence sont très creuses.

CONCLUSION ET PERSPECTIVES

L'objectif initial de cette thèse était d'étudier formellement des algorithmes en algèbre linéaire qui soient exécutables efficacement. Plus précisément, il s'agissait de vérifier la correction de ces algorithmes à l'aide de Coq tout en permettant d'observer en pratique des temps d'exécution qui soient en adéquation avec la complexité théorique de ces algorithmes. Typiquement, nous avons cherché à avoir un comportement asymptotique de même exposant que le modèle théorique (des améliorations sur le facteur constant sont probablement encore possibles).

Nous pensons avoir rempli cet objectif, en proposant d'une part des techniques pour l'amélioration de l'efficacité des programmes vérifiés formellement (partie I) et d'autre part l'étude de certains algorithmes en particulier (partie II) avec à chaque fois une preuve formelle et une mesure de performances. L'application de ces algorithmes et techniques à l'homologie des images numériques (partie III) nous a permis d'ouvrir le champ des applications, et de voir que nos contributions ont pu être mises à profit pour des lignes de travaux variées, notamment au sein du projet européen ForMath.

La problématique que nous avons traitée nous paraît originale car le souci de performances est souvent peu présent dans les mathématiques formalisées, où la priorité est donnée à l'abstraction et aux approches facilitant la preuve de propriétés élaborées, si possible d'une manière proche de la pratique mathématique habituelle sur papier. On retrouve davantage cet aspect du côté de la vérification de programmes, car le produit final est un programme extrait auquel les garanties formelles s'appliquent. Le comportement calculatoire de ce programme, notamment son efficacité, est alors un facteur important. En revanche, l'abstraction sur les concepts manipulés au cours des preuves est souvent moins critique que lorsque l'on manipule par exemple des structures algébriques.

Ce point de vue intermédiaire est très proche des préoccupations du calcul formel, où des expressions mathématiques sont manipulées par des procédures dont l'efficacité est souvent critique. Néanmoins, travaillant dans un cadre formel strict, nous avons dû être précis sur la sémantique des objets et des transformations qu'ils subissent. Ceci ouvre la voie à l'utilisation de techniques de calcul formel comme outils totalement intégrés d'aide à la preuve formelle.

Nous avons clairement fait le choix de préférer la réutilisabilité à l'exhaustivité. Ainsi, cette thèse n'est pas un catalogue de preuves formelles de tous les algorithmes utiles en algèbre linéaire. Au contraire, nous nous sommes concentrés sur quelques exemples que nous jugions les plus fondamentaux et utiles. Cependant, une grande partie de notre effort a été dirigée vers la pérennité et l'impact de nos développements. La première manifestation de cette volonté est que nous avons tenu nous-mêmes à réutiliser l'existant autant que possible. L'utilisation de la bibliothèque SSReflect notamment nous a permis d'avoir rapidement à disposition une base importante de concepts et de preuves formalisés pour l'algèbre linéaire, ce qui a grandement facilité la preuve de nos algorithmes et nous a permis de nous concentrer sur les aspects d'efficacité calculatoire.

Toujours dans l'optique de rendre nos développements pérennes, nous avons veillé à intégrer à Coq les technologies présentées au chapitre 1, au lieu de nous contenter d'un prototype. De même, nous avons organisé notre système de raffinements du chapitre 2 ainsi que les preuves de nos algorithmes en une bibliothèque pour Coq, destinée à être utilisée en complément de SSReflect. Ainsi, l'utilisateur dispose de toutes les théories disponibles dans SSReflect, mais également de la possibilité de basculer à tout moment vers des représentations lui permettant de mener des calculs si les données en entrées sont connues. Nous avons baptisé cette bibliothèque CoqEAL, pour « Coq Effective Algebra Library ».

Le fait d'organiser ainsi notre développement, destiné à être distribué et utilisé, nous a conduit à une plus grande rigueur notamment pour la clarté et la concision de nos preuves formelles que si nous avions simplement voulu certifier un résultat une fois pour toutes. Nous insistons donc sur le fait que les développements accompagnant cette thèse forment une part importante, sinon essentielle, de nos contributions. Des indicateurs de leur taille ainsi que des instructions pour les télécharger sont décrits dans la section 0.5 en introduction.

Bien sûr, notre étude est limitée par plusieurs aspects qu'il serait intéressant de développer dans des travaux futurs. En premier lieu, nous nous sommes penchés sur des problèmes d'efficacité en *temps*. Or en pratique, la limitation de la consommation des ressources *mémoire* est au moins aussi importante, car très bloquante : si l'exécution d'un programme est plus longue, il est possible d'attendre plus longtemps. Si par contre elle consomme plus de mémoire, les limites du système peuvent être atteintes et le calcul ne sera jamais mené.

Cependant, bien que nous n'ayons pas directement prêté attention à ces problèmes de consommation mémoire, nous ne les avons pas non plus complètement ignorés. En effet, les programmes souffrant d'une très mauvaise gestion de la mémoire sont également gourmands en temps, surtout dans les langages que nous utilisons où l'allocation et la libération de la mémoire sont faites automatiquement et peuvent être coûteuses. Nous avons par exemple évoqué la question des représentations de données, comme les nombres ou les matrices, avec des contraintes qui découlaient finalement assez largement d'une problématique d'économie d'espace, tout autant que de temps.

Sur un autre plan, nous nous sommes également cantonnés à de l'arithmétique exacte. Ainsi, notre approche est immédiatement applicable aux entiers, aux corps finis, aux nombres rationnels ou encore aux réels exacts. Dans des contextes où l'efficacité est primordiale et où il est possible de faire des approximations numériques, l'utilisation d'arithmétique flottante serait toute indiquée. Malheureusement, de tels calculs numériques introduisent leurs propres difficultés (notamment la question de la stabilité) que nous n'avons pas étudié. Des travaux dans ce sens sont menés dans Coq [G. Melquiond, 2012], il serait intéressant de voir comment les réutiliser dans notre contexte.

Nous avons également fait le choix de ne pas traiter d'algorithmes probabilistes, qui sont pourtant souvent utilisés en calcul formel. L'intégration de tels algorithmes dans un contexte de vérification formelle est un sujet de recherche à part entière [P. Audebaud et C. Paulin-Mohring, 2009], nous n'avons donc pas jugé réaliste d'aborder cette question en première approche.

Enfin, la multiplication des architectures multi-cœurs ou distribuées pose la question du parallélisme. Certains algorithmes courants en algèbre linéaire admettent une formulation tirant parti de la possibilité d'exécution simultanée de programmes. En théorie, nous aurions pu les étudier et espérer améliorer les performances obtenues à moindre frais car le langage fonctionnel intégré au formalisme de Coq est purement fonctionnel, ce qui simplifie grandement la problématique du parallélisme. Cependant, nous aurions rencontré deux obstacles technologiques : tout d'abord, ce formalisme ne donne en l'état aucun moyen d'exprimer le parallélisme. Ensuite, l'environnement d'exécution du langage OCaml ne permet pas, à l'heure actuelle, d'exploiter les architectures multi-cœurs. Le système Coq hérite de ces limitations, même si des efforts sont en cours pour paralléliser certains aspects de la vérification de preuves [B. Barras, L. D. C. González-Huesca et al., 2013].

Pour conclure, signalons que nous menons des travaux dans le prolongement direct du matériel présenté dans cette thèse. Notamment, nous développons un module formellement vérifié d'opérations sur les matrices creuses, qui permettra vraisemblablement d'améliorer les performances des calculs de groupes d'homologie. Également, nous étendons en collaboration avec Cyril Cohen et Anders Mörtberg le développement formel sur l'homologie à des coefficients entiers, mettant à profit l'algorithme de calcul de la forme normale de Smith.

BIBLIOGRAPHIE

Abdeljaoued, Jounaidi et Henri Lombardi

2003 *Méthodes matricielles - Introduction à la complexité algébrique*, Springer, t. 42.

Abrial, Jean-Raymond

2005 *The B-book - assigning programs to meanings*, Cambridge University Press, p. I–XXXIV, 1–779.

Aehlig, Klaus, Florian Haftmann et Tobias Nipkow

2008 « A Compiled Implementation of Normalization by Evaluation », in *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada*, Lecture Notes in Computer Science, Springer, t. 5170, p. 39–54.

Ahrens, B., C. Kapulkin et M. Shulman

2013 « Univalent categories and the Rezk completion », *ArXiv e-prints*, arXiv : 1303.0584.

Altenkirch, Thorsten, Conor McBride et Peter Morris

2006 « Generic Programming with Dependent Types », in *Datatype-Generic Programming - International Spring School, SSDGP 2006, Nottingham, UK, Revised Lectures*, sous la dir. de Roland Carl Backhouse, Jeremy Gibbons, Ralf Hinze et Johan Jeuring, Lecture Notes in Computer Science, Springer, t. 4719, p. 209–257.

Aransay, Jesús, Clemens Ballarin et Julio Rubio

2008 « A Mechanized Proof of the Basic Perturbation Lemma », *J. Autom. Reasoning*, 40, 4, p. 271–292.

Armand, Michaël, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry et Benjamin Werner

2011 « A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses », in *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan.* Lecture Notes in Computer Science, Springer, t. 7086, p. 135–150.

Armand, Michaël, Benjamin Grégoire, Arnaud Spiwack et Laurent Théry

2010 « Extending Coq with Imperative Features and Its Application to SAT Verification », in *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK.* Lecture Notes in Computer Science, Springer, t. 6172, p. 83–98.

Armstrong, Joe

2010 « Erlang », *Commun. ACM*, 53, 9, p. 68–75.

Audebaud, Philippe et Christine Paulin-Mohring

2009 « Proofs of randomized algorithms in Coq », *Sci. Comput. Program.*, 74, 8, p. 568–589.

Ayala, Rafael, Eladio Domínguez, Angel R. Francés et Antonio Quintero

2003 « Homotopy in digital spaces », *Discrete Applied Mathematics*, 125, 1, p. 3–24.

Back, Ralph-Johan

1978 *On the Correctness of Refinement Steps in Program Development*, thèse de doct., Åbo Akademi, Department of Computer Science, Helsinki, Finland.

Back, Ralph-Johan et Joakim von Wright

1999 *Refinement calculus - a systematic introduction*, Undergraduate texts in computer science, Springer, p. I–XV, 1–519.

Baker, Henry G.

1978 « Shallow Binding in LISP 1.5 », *Commun. ACM*, 21, 7, p. 565–569.

1991 « Shallow binding makes functional arrays fast », *SIGPLAN Notices*, 26, 8, p. 145–147.

Barendregt, Henk et Erik Barendsen

2002 « Autarkic Computations in Formal Proofs », *J. Autom. Reasoning*, 28, 3, p. 321–336.

Barras, Bruno, Lourdes Del Carmen González-Huesca, Hugo Herbelin, Yann Régis-Gianas, Enrico Tassi, Makarius Wenzel et Burkhart Wolff

2013 « Pervasive Parallelism in Highly-Trustable Interactive Theorem Proving Systems », in *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK. Proceedings*, Lecture Notes in Computer Science, Springer, t. 7961, p. 359–363.

Barras, Bruno et Benjamin Grégoire

2005 « On the Role of Type Decorations in the Calculus of Inductive Constructions », in *Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK*, Lecture Notes in Computer Science, Springer, t. 3634, p. 151–166.

Barras, Bruno, Jean-Pierre Jouannaud, Pierre-Yves Strub et Qian Wang

2011 « CoQMTU : A Higher-Order Type Theory with a Predicative Hierarchy of Universes Parametrized by a Decidable First-Order Theory », in *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, Toronto, Ontario, Canada*, IEEE Computer Society, p. 143–151.

Barthe, Gilles, Venanzio Capretta et Olivier Pons

2003 « Setoids in type theory », *J. Funct. Program.*, 13, 2, p. 261–293.

Barthe, Gilles, Mark Ruys et Henk Barendregt

1995 « A Two-Level Approach Towards Lean Proof-Checking », in *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, Selected Papers*, Lecture Notes in Computer Science, Springer, t. 1158, p. 16–35.

Bernardy, Jean-Philippe, Patrik Jansson et Ross Paterson

2012 « Proofs for free - Parametricity for dependent types », *J. Funct. Program.*, 22, 2, p. 107–152.

Bertot, Yves, Georges Gonthier, Sidi Ould Biha et Ioana Pasca

2008 « Canonical Big Operators », in *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada*, Lecture Notes in Computer Science, Springer, t. 5170, p. 86–101.

Besson, Frédéric, Pierre-Emmanuel Cornilleau et David Pichardie

2011 « Modular SMT Proofs for Fast Reflexive Checking Inside Coq », in *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan.* Lecture Notes in Computer Science, Springer, t. 7086, p. 151–166.

Bilton, Nick

2011 *IPhone Alarm Glitch Greets New Year - NYTimes.com*, http://www.nytimes.com/2011/01/03/technology/03iphone.html?_r=1&.

Boespflug, Mathieu

2010 « Conversion by Evaluation », in *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain*, Lecture Notes in Computer Science, Springer, t. 5937, p. 58–72.

Boespflug, Mathieu, Maxime Dénès et Benjamin Grégoire

2011 « Full Reduction at Full Throttle », in *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan.* Lecture Notes in Computer Science, Springer, t. 7086, p. 362–377.

Boulmé, Sylvain

2000 *Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel*, thèse de doct.

Boutin, Samuel

1997 « Using Reflection to Build Efficient and Certified Decision Procedures », in *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan*, Lecture Notes in Computer Science, Springer, t. 1281, p. 515–529.

Boyer, Brice, Jean-Guillaume Dumas, Clément Pernet et Wei Zhou

2009 « Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm », in *Symbolic and Algebraic Computation, International Symposium, ISSAC 2009, Seoul, Republic of Korea, Proceedings*, ACM, p. 55–62.

Brisebarre, Nicolas, Mioara Joldes, Érik Martin-Dorel, Micaela Mayero, Jean-Michel Muller, Ioana Pasca, Laurence Rideau et Laurent Théry

2012 « Rigorous Polynomial Approximation Using Taylor Models in Coq », in *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA. Proceedings*, Lecture Notes in Computer Science, Springer, t. 7226, p. 85–99.

Bunch, James R et John E Hopcroft

1974 « Triangular factorization and inversion by fast matrix multiplication », *Mathematics of Computation*, 28, 125, p. 231–236.

Cano, Guillaume et Maxime Dénès

2013 « Matrices à blocs et en forme canonique », in *JFLA - Journées francophones des langages applicatifs*, Aussois, France.

Chemla, Karine et Guo Shuchun

2004 *Les neuf chapitres le classique mathématique de la Chine ancienne et ses commentaires*, French, Dunod.

Chrzaszcz, Jacek

2003 « Implementing Modules in the Coq System », in *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rom, Italy, Proceedings*, sous la dir. de David A. Basin et Burkhart Wolff, Lecture Notes in Computer Science, Springer, t. 2758, p. 270–286.

Church, Alonzo

1940 « A Formulation of the Simple Theory of Types », *J. Symb. Log.*, 5, 2, p. 56–68.

Cohen, Cyril

2012 *Formalized algebraic numbers : construction and first order theory*, thèse de doct., École Polytechnique.

2013 « Pragmatic Quotient Types in Coq », in *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France. Proceedings*, Lecture Notes in Computer Science, Springer, t. 7998, p. 213–228.

Cohen, Cyril, Maxime Dénès et Anders Mörtberg

2013 « Refinements for free ! », in *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, Australia.* Lecture Notes in Computer Science, Springer.

Conchon, Sylvain et Jean-Christophe Filliâtre

2007 « A persistent union-find data structure », in *Proceedings of the ACM Workshop on ML, 2007, Freiburg, Germany*, ACM, p. 37–46.

Coppersmith, Don et Shmuel Winograd

1990 « Matrix multiplication via arithmetic progressions », *Journal of Symbolic Computation*, 9, 3, p. 251–280.

Coquand, Thierry et al.

2010 *ForMath : Formalisation of Mathematics*, http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ForMath.

Coquand, Thierry et Gérard P. Huet

1988 « The Calculus of Constructions », *Inf. Comput.*, 76, 2/3, p. 95–120.

Coquand, Thierry et Henrik Persson

1998 « Gröbner Bases in Type Theory », in *Types for Proofs and Programs, International Workshop TYPES '98, Kloster Irsee, Germany, Selected Papers*, Lecture Notes in Computer Science, Springer, t. 1657, p. 33–46.

Cormen, Thomas H, Charles E Leiserson, Ronald L Rivest et Clifford Stein

2001 *Introduction to algorithms*, MIT press.

Cornilleau, Pierre-Emmanuel

2013 *Certification of static analysis in many-sorted first-order logic*, thèse de doct., École normale supérieure de Cachan-ENS Cachan.

D'Alberto, Paolo et Alexandru Nicolau

2009 « Adaptive Winograd's matrix multiplications », *ACM Trans. Math. Softw.*, 36, 1.

Danielsson, Nils Anders et Thierry Coquand

2013 *Isomorphism is Equality*, Preprint. http://www.cse.chalmers.se/~nad/publications/coquand-danielsson-isomorphism-is-equality.html.

Danvy, Olivier

1996 « Type-Directed Partial Evaluation », in *Conference Record of POPL'96 : The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA*, ACM Press, p. 242–257.

Dargaye, Zaynah

2009 *Vérification formelle d'un compilateur optimisant pour langages fonctionnels*, thèse de doct., Université Paris-Diderot-Paris VII.

De Bruijn, Nicolaas Govert

1970 « The mathematical language AUTOMATH, its usage, and some of its extensions », in *Symposium on Automatic Demonstration*, Lecture Notes in Mathematics, Springer, t. 125, p. 29–61.

Dénès, Maxime, Anders Mörtberg et Vincent Siles

2012 « A Refinement-Based Approach to Computational Algebra in Coq », in *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA*, Lecture Notes in Computer Science, Springer, t. 7406, p. 83–98.

Dey, Tamal K., Herbert Edelsbrunner et Sumanta Guha

1999 « Contemporary Mathematics », in AMS, Providence, t. 223, chap. Computational topology. Advances in Discrete and Computational Geometry, p. 190–143.

Domínguez, César et Julio Rubio

2011 « Effective homology of bicomplexes, formalized in Coq », *Theor. Comput. Sci.*, 412, 11, p. 962–970.

Dousson, X., F. Sergeraert et Y. Siret

1998 *The Kenzo program*, http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo, Institut Fourier, Grenoble.

Dowek, Gilles, Thérèse Hardin et Claude Kirchner

2003 « Theorem Proving Modulo », *J. Autom. Reasoning*, 31, 1, p. 33–72.

Filinski, Andrzej et Henning Korsholm Rohde

2004 « A Denotational Account of Untyped Normalization by Evaluation », in *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004, Barcelona, Spain*, Lecture Notes in Computer Science, Springer, t. 2987, p. 167–181.

Forman, Robin

1998 « Morse theory for cell complexes », *Advances in mathematics*, 134, 1, p. 90–145.

Gamboa, Ruben, John Cowles et Jeff Van Baalen

2003 « Using ACL2 Arrays to Formalize Matrix Algebra », in *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '03)*.

Geuvers, Herman, Freek Wiedijk et Jan Zwanenburg

2000 « A Constructive Proof of the Fundamental Theorem of Algebra without Using the Rationals », in *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, Selected Papers*, Lecture Notes in Computer Science, Springer, t. 2277, p. 96–111.

Giesbrecht, Mark

1995 « Nearly Optimal Algorithms for Canonical Matrix Forms », *SIAM J. Comput.*, 24, 5, p. 948–969.

Gonthier, Georges

2007 « The Four Colour Theorem : Engineering of a Formal Proof », in *Computer Mathematics, 8th Asian Symposium, ASCM 2007*, Lecture Notes in Computer Science, Springer, t. 5081, p. 333.

2011 « Point-Free, Set-Free Concrete Linear Algebra », in *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands*, Lecture Notes in Computer Science, Springer, t. 6898, p. 103–118.

Gonthier, Georges, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi et Laurent Théry

2013 « A Machine-Checked Proof of the Odd Order Theorem », in *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France. Proceedings*, Lecture Notes in Computer Science, Springer, t. 7998, p. 163–179.

Gonthier, Georges, Assia Mahboubi et Enrico Tassi

2008 *A Small Scale Reflection Extension for the Coq system*, Research Report RR-6455, INRIA, http://hal.inria.fr/inria-00258384.

Gonthier, Georges, Beta Ziliani, Aleksandar Nanevski et Derek Dreyer

2011 « How to make ad hoc proof automation less ad hoc », in *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan*, ACM, p. 163–175.

González-Díaz, Rocío, Belén Medrano, Pedro Real et Javier Sánchez-Peláez

2005 « Algebraic Topological Analysis of Time-Sequence of Digital Images », in *Computer Algebra in Scientific Computing, 8th International Workshop, CASC 2005, Kalamata, Greece*, Lecture Notes in Computer Science, Springer, t. 3718, p. 208–219.

González-Díaz, Rocío et Pedro Real

2011 « On the Cohomology of 3D Digital Images », *CoRR*, abs/1105.4477.

Grégoire, Benjamin et Xavier Leroy

2002 « A compiled implementation of strong reduction », in *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA*, ACM, p. 235–246.

Grégoire, Benjamin et Assia Mahboubi

2005 « Proving Equalities in a Commutative Ring Done Right in Coq », in *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK*, Lecture Notes in Computer Science, Springer, t. 3603, p. 98–113.

Hales, Thomas C.

2005 « Introduction to the Flyspeck Project », in *Mathematics, Algorithms, Proofs*, Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, t. 05021.

Harrison, John

1995 *Metatheory and Reflection in Theorem Proving : A Survey and Critique*, Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK.

2005 « A HOL Theory of Euclidean Space », in *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK*, Lecture Notes in Computer Science, Springer, t. 3603, p. 114–129.

Harrison, John et Laurent Théry

1998 « A Skeptic's Approach to Combining HOL and Maple », *J. Autom. Reasoning*, 21, 3, p. 279–294.

Hendrix, Joe

2003 « Matrices in ACL2 », in *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '03)*.

Heras, Jónathan, Maxime Dénès, Gadea Mata, Anders Mörtberg, María Poza et Vincent Siles

2012 « Towards a Certified Computation of Homology Groups for Digital Images », in *Computational Topology in Image Context - 4th International Workshop, CTIC 2012, Bertinoro, Italy*, Lecture Notes in Computer Science, Springer, t. 7309, p. 49–57.

Heras, Jónathan, María Poza, Maxime Dénès et Laurence Rideau

2011 « Incidence Simplicial Matrices Formalized in Coq/SSReflect », in *Intelligent Computer Mathematics - 18th Symposium, Calculemus 2011, and 10th International Conference, MKM 2011, Bertinoro, Italy*, Lecture Notes in Computer Science, Springer, t. 6824, p. 30–44.

Herbelin, Hugo et al.

2013 « The Coq proof assistant reference manual », http://coq.inria.fr/distrib/current/refman/.

Hoare, C. A. R.

1972 « Proof of Correctness of Data Representations », *Acta Inf.*, 1, p. 271–281.

Hopcroft, John E et Leslie R Kerr

1971 « On Minimizing the Number of Multiplications Necessary for Matrix Multiplication », *SIAM Journal on Applied Mathematics*, 20, 1, p. 30–36.

Howe, Douglas J.

1988 « Computational Metatheory in Nuprl », in *9th International Conference on Automated Deduction, Argonne, Illinois, USA, Proceedings*, Lecture Notes in Computer Science, Springer, t. 310, p. 238–257.

Hudak, Paul, John Hughes, Simon L. Peyton Jones et Philip Wadler

2007 « A history of Haskell : being lazy with class », in *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA*, sous la dir. de Barbara G. Ryder et Brent Hailpern, ACM, p. 1–55.

Huss-Lederman, Steven, Elaine M. Jacobson, J. R. Johnson, Anna Tsao et Thomas Turnbull

1996 « Implementation of Strassen's Algorithm for Matrix Multiplication », in *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing, Pittsburgh, PA, USA*, IEEE, p. 32.

Kaliszyk, Cezary et Freek Wiedijk

2007 « Certified Computer Algebra on Top of an Interactive Theorem Prover », in *Towards Mechanized Mathematical Assistants, 14th Symposium, Calculemus 2007, 6th International Conference, MKM 2007, Hagenberg, Austria, Proceedings*, Lecture Notes in Computer Science, Springer, t. 4573, p. 94–105.

Keller, C.

2013 *A Matter of Trust : Skeptical Communication Between Coq and External Provers*, thèse de doct., École Polytechnique.

Keller, Chantal et Marc Lasson

2012 « Parametricity in an Impredicative Sort », in *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, Fontainebleau, France*, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, t. 16, p. 381–395.

Keller-Gehrig, Walter

1985 « Fast Algorithms for the Characteristic Polynomial », *Theor. Comput. Sci.*, 36, p. 309–317.

Klein, Gerwin, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch et Simon Winwood

2009 « seL4 : formal verification of an OS kernel », in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA*, ACM, p. 207–220.

Knight, Philip A.

1995 « Fast rectangular matrix multiplication and QR decomposition », *Linear Algebra and its Applications*, 221, p. 69–81.

Knuth, Donald E.

1997 *The art of computer programming, volume 2 (3rd ed.) : seminumerical algorithms*, Addison-Wesley.

Krebbers, Robbert et Bas Spitters

2011 « Type classes for efficient exact real arithmetic in Coq », *Logical Methods in Computer Science*, 9, 1.

Lammich, Peter

2013 « Automatic Data Refinement », in *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France. Proceedings*, Lecture Notes in Computer Science, Springer, t. 7998, p. 84–99.

Leroy, Xavier

1990 « The ZINC experiment : an economical implementation of the ML language ».

2009 « Formal verification of a realistic compiler », *Commun. ACM*, 52, 7, p. 107–115.

Leroy, Xavier, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy et Jérôme Vouillon

2012 « The OCaml system release 4.00 ».

Letouzey, Pierre

2008 « Extraction in Coq : An Overview », in *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, Proceedings*, sous la dir. d'Arnold Beckmann, Costas Dimitracopoulos et Benedikt Löwe, Lecture Notes in Computer Science, Springer, t. 5028, p. 359–369.

Lindley, Sam

2005 *Normalisation by evaluation in the compilation of typed functional programming languages*, thèse de doct., University of Edinburgh.

Lombardi, Henri et Claude Quitté

2011 *Algèbre commutative : méthodes constructives*, Calvage et Mounet.

Luo, Zhaohui

1989 « ECC, an Extended Calculus of Constructions », in *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA*, IEEE Computer Society, p. 386–395.

Mackenzie, Dana

2003 « Topologists and Roboticists Explore and Inchoate World », *Science*, 8, p. 756.

Magaud, Nicolas

2005 *Programming with Dependent Types in Coq : a Study of Square Matrices*, Unpublished. A preliminary version appeared in Coq contributions.

Magaud, Nicolas et Yves Bertot

2001 « Changement de représentation des structures de données en Coq : le cas des entiers naturels », in *Journées francophones des langages applicatifs (JFLA'01), Pontarlier, France*, Collection Didactique, INRIA, p. 1–16.

Martin-Dorel, Erik

2012 *Contributions à la vérification formelle d'algorithmes arithmétiques*, thèse de doct., Ecole normale supérieure de Lyon.

Martin-Lof, Per et Giovanni Sambin

1984 *Intuitionistic type theory*, Bibliopolis Naples,, Italy, t. 17.

Mata, Gadea

2011 *SynapCountJ*, http://imagejdocu.tudor.lu/doku.php?id=plugin:utilities:synapsescountj:start, Université de La Rioja.

McBride, Conor et James McKinna

2004 « The view from the left », *J. Funct. Program.*, 14, 1, p. 69–111.

McCarthy, John

1960 « Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I », *Commun. ACM*, 3, 4, p. 184–195.

Melquiond, Guillaume

2012 « Floating-point arithmetic in the Coq system », *Inf. Comput.*, 216, p. 14–23.

Milner, Robin, Mads Tofte et Robert Harper

1990 *Definition of standard ML*, MIT Press, p. I–XI, 1–101.

Mines, Ray, Fred Richman et Wim Ruitenburg

1988 *A course in constructive algebra*, Springer.

Munkres, James R

1984 *Elements of algebraic topology*, Addison-Wesley Reading, t. 2.

Obua, Steven

2005 « Proving Bounds for Real Linear Programs in Isabelle/HOL », in *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK*, Lecture Notes in Computer Science, Springer, t. 3603, p. 227–244.

Oostdijk, Martijn et Herman Geuvers

2002 « Proof by computation in the Coq system », *Theor. Comput. Sci.*, 272, 1-2, p. 293–314.

Orden, David et Francisco Santos

2003 « Asymptotically Efficient Triangulations of the d-Cube », *Discrete & Computational Geometry*, 30, 4, p. 509–528.

Ould Biha, Sidi

2008 « Formalisation des mathématiques : une preuve du théorème de Cayley-Hamilton », French, in *JFLA (Journées Francophones des Langages Applicatifs)*, Etretat, France, p. 1–14.

Palomo-Lozano, Francisco, Inmaculada Medina-Bulo et José A. Alonso-Jiménez

2001 « Certification of Matrix Multiplication Algorithms. Strassen's Algorithm in ACL2 », in *Supplemental Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, Edinburgh, t. Informatics Research Report EDI-INF-RR-0046.

Paulin-Mohring, Christine

1989a « Extracting F(omega)'s Programs from Proofs in the Calculus of Constructions », in *POPL*, ACM Press, p. 89–104.

1989b *Extraction de programmes dans le Calcul des Constructions*, thèse de doct., Université Paris-Diderot-Paris VII.

Potet, Marie-Laure et Yann Rouzaud

1998 « Composition and Refinement in the B-Method », in *B'98 : Recent Advances in the Development and Use of the B Method, Second International B Conference, Montpellier, France, Proceedings*, Lecture Notes in Computer Science, Springer, t. 1393, p. 46–65.

Pottier, Loic

1999 *User contributions in Coq : Algebra*, http://coq.inria.fr/pylons/contribs/view/Algebra/trunk.

Pous, Damien

2012 « Untyping Typed Algebras and Colouring Cyclic Linear Logic », *Logical Methods in Computer Science*, 8, 2.

Poza López de Echazarreta, María

2013 *Certifying homological algorithms to study biomedical images*, thèse de doct., Universidad de La Rioja.

Probert, Robert L.

1976 « On the Additive Complexity of Matrix Multiplication », *SIAM Journal on Computing*, 5, 2, p. 187–203.

Rasband, Wayne

2003 *ImageJ : Image Processing and Analysis in Java*, http://rsb.info.nih.gov/ij/.

Reynolds, John C.

1983 « Types, Abstraction and Parametric Polymorphism », in *IFIP Congress*, p. 513–523.

Rijke, Egbert et Bas Spitters

2013 « Sets in homotopy type theory », *CoRR*, abs/1305.3835.

Romero, Ana et Francis Sergeraert

2010 « Discrete Vector Fields and Fundamental Algebraic Topology », *CoRR*, abs/1005.5685.

Rubio, Julio et Francis Sergeraert

2006 *Constructive Homological Algebra and Applications, Lecture Notes Summer School on Mathematics, Algorithms, and Proofs*, University of Genova.

Schönhage, Arnold

1981 « Partial and Total Matrix Multiplication », *SIAM J. Comput.*, 10, 3, p. 434–455.

Ségonne, Florent, W. Eric L. Grimson et Bruce Fischl

2003 « Topological Correction of Subcortical Segmentation », in *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2003, 6th International Conference, Montréal, Canada*, Lecture Notes in Computer Science, Springer, t. 2879, p. 695–702.

Selkoe, Dennis J

2002 « Alzheimer's disease is a synaptic failure », *Science*, 298, 5594, p. 789–791.

Siles, Vincent

2012 *A formal proof of the Cauchy-Binet formula*, http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ProofExamples.

Soubiran, Elie

2010 *Développement modulaire de théories et gestion de l'espace de nom pour l'assistant de preuve Coq.* French, thèse de doct., École Polytechnique, http://tel.archives-ouvertes.fr/tel-00679201.

Sozeau, Matthieu

2010 « A New Look at Generalized Rewriting in Type Theory », *Journal of Formalized Reasoning*, 2, 1.

Sozeau, Matthieu et Nicolas Oury

2008 « First-Class Type Classes », in *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada*, Lecture Notes in Computer Science, Springer, t. 5170, p. 278–293.

Spitters, Bas et Eelis van der Weegen

2011 « Type classes for mathematics in type theory », *Mathematical Structures in Computer Science*, 21, 4, p. 795–825.

Spiwack, Arnaud

2011 *Verified Computing in Homological Algebra, A Journey Exploring the Power and Limits of Dependent Type Theory*, thèse de doct., PhD thesis, École Polytechnique.

Stein, Jasper

2001 *Documentation of my formalization of Linear Algebra*, rapp. tech., http://www.cs.ru.nl/~jasper/WWW/documentation.dvi.

Storjohann, Arne

2001 « Deterministic Computation of the Frobenius Form », in *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, Las Vegas, Nevada, USA*, IEEE Computer Society, p. 368–377.

Strassen, Volker

1969 « Gaussian elimination is not optimal », *Numerische Mathematik*, 13, 4, p. 354–356.

Théry, Laurent

2001 « A Machine-Checked Implementation of Buchberger's Algorithm », *J. Autom. Reasoning*, 26, 2, p. 107–137.

2008 « Proof Pearl : Revisiting the Mini-rubik in Coq », in *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada*, Lecture Notes in Computer Science, Springer, t. 5170, p. 310–319.

Verma, Kumar Neeraj, Jean Goubault-Larrecq, Sanjiva Prasad et S. Arun-Kumar

2000 « Reflecting BDDs in Coq », in *Advances in Computing Science - ASIAN 2000, 6th Asian Computing Science Conference, Penang, Malaysia*, Lecture Notes in Computer Science, Springer, t. 1961, p. 162–181.

Wadler, Philip

1987 « Views : A Way for Pattern Matching to Cohabit with Data Abstraction », in *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany*, ACM Press, p. 307–313.

1989 « Theorems for free ! », in *Functional Programming Languages and Computer Architecture*, ACM Press, p. 347–359.

Wedderburn, Joseph Henry Maclagan

2002 *Lectures on matrices*, American Mathematical Society, t. 17.

Williams, Virginia Vassilevska

2012 « Multiplying matrices faster than coppersmith-winograd », in *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA*, ACM, p. 887–898.

WINOGRAD, Shmuel

1971 « On multiplication of 2× 2 matrices », *Linear algebra and its applications*, 4, 4, p. 381–388.

WOOD, Jay A.

1989 « Spinor groups and algebraic coding theory », *J. Comb. Theory, Ser. A*, 51, 2, p. 277–313.

ZIOU, Djemel et Madjid ALLILI

2002 « Generating cubical complexes from image data and computation of the Euler number », *Pattern Recognition*, 35, 12, p. 2833–2839.