Changements de Représentation des Données dans le Calcul des Constructions

Nicolas Magaud

UNIVERSITE DE NICE - SOPHIA ANTIPOLIS UFR Sciences

École Doctorale STIC

THÈSE

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice - Sophia Antipolis

Spécialité : Informatique

prsente et soutenue par

Nicolas MAGAUD

Équipe d'accueil : LEMME – INRIA Sophia-Antipolis

Changements de Représentation des Données dans le Calcul des Constructions

Thse dirige par Yves BERTOT

Soutenue publiquement le mardi 21 octobre 2003 à 14 heures devant le jury composé de :

Mme. :	Laurence	Pierre	I3S, Université de Nice	Présidente
MM. :	James	McKinna	Université de St. Andrews, Royaume-Uni	Rapporteurs
	Christine	Paulin	Université Paris Sud	
MM. :	Pierre	Castéran	LaBRI, Université de Bordeaux	Examinateurs
	Jean-Michel	Muller	CNRS, École Normale Supérieure de Lyon	
M. :	Yves	Bertot	INRIA Sophia-Antipolis	Directeur

Remerciements

Je tiens tout d'abord à remercier James McKinna et Christine Paulin d'avoir accepté d'être les rapporteurs de cette thèse. Je remercie également Pierre Castéran et Jean-Michel Muller d'avoir accepté d'être examinateurs de cette thèse et Laurence Pierre d'avoir accepté de présider ce jury. Enfin, je remercie Yves Bertot de m'avoir encadré durant mon stage de DEA, puis tout au long de cette thèse.

Jean-Christophe Filliâtre et Christine Paulin m'ont permis de découvrir le métier de chercheur au cours de mon stage de licence en 1998. Leur enthousiasme communicatif m'a conforté dans mon envie de travailler dans le domaine des preuves formelles. Je tiens à les remercier pour leur accueil et le soutien permanent qu'ils m'ont apporté depuis.

Au cours de cette thèse, j'ai eu l'occasion de rencontrer et de collaborer avec de nombreuses personnes. Outre mes co-auteurs Yves Bertot et Paul Zimmermann, je tiens à remercier Didier Bondyfalat pour son aide au cours du développement de la preuve formelle de correction de la racine carrée de GMP et Jean-Christophe Filliâtre pour les nombreuses mises à jour de Correctness puis de Why qu'il a faites dans ce cadre. Je remercie également Laurent Théry, Gilles Barthe, Olivier Pons, Conor McBride et James McKinna pour les discussions intéressantes que nous avons eu sur la question du changement de représentation des données en théorie des types.

Les membres du projet Lemme m'ont chaleureusement accueilli au début de mon stage de DEA et m'ont fourni un cadre de travail très agréable tout au long de cette thèse. Je les en remercie. Je remercie également mes cobureaux Claude, Frédérique et Kuntal de m'avoir supporté certains jours. Merci aussi à Laurent C. pour m'avoir attendu ;-)

Parce qu'il y a une vie en dehors du boulot, je remercie les squasheurs (Francis, Franck, Pierre, Christian, Jérôme, Felipe...) pour m'avoir patiemment appris à jouer au squash, et les grimpeurs (Guillaume, Nicolas D., Ludo...) pour la découverte de nombreux sites d'escalade de la région. Je remercie également tous les gens qui m'ont entouré durant ma thèse pour leur enthousiasme et leur bonne humeur : Antonia, Eric, Fabrice, Pascal, Olivier, Jean-Ferdy (si,si) et tous les autres. Merci à Xavier T. et Oscar pour quelques mémorables soirées.

Finalement, je remercie mes parents et mon frère pour leur soutien et leurs encouragements tout au long de mes études.

Structure du manuscrit

Introduction

Le premier chapitre de cette thèse présente les motivations et le contexte dans lequel ont été menées ces recherches.

Preuve formelle de l'implantation de la racine carrée de GMP

Nous présentons une preuve de correction de l'implantation en C du calcul de la racine carrée, telle qu'elle est faite dans la bibliothèque de calcul en multi-précision Gmp. Dans le chapitre 2, nous décrivons formellement en Coq les opérations de base sur les entiers de la bibliothèque Gmp. Dans le chapitre 3, nous présentons l'algorithme original de calcul de la racine carrée considéré. Dans le chapitre 4, nous donnons trois descriptions formelles du programme de calcul de la racine carrée de Gmp. La première exprime les propriétés mathématiques de l'algorithme. La deuxième fournit un procédé correct de calcul fonctionnel (dans Coq) de la racine carrée. La troisième description montre que le code C fourni dans la bibliothèque Gmp calcule bien la racine carrée.

Changements de représentation des données

La seconde partie de ce manuscrit traite du changement de représentation concrète des données dans une théorie développée formellement avec un outil comme Coq [Coq02]. Nous expliquons, dans le chapitre 5, ce que l'on entend par *changement de représentation*. Nous présentons à cet effet les types inductifs et leurs propriétés dans Coq, ainsi que de nombreux travaux reliés à notre étude. Dans le chapitre 6, nous montrons comment mettre en évidence les étapes de preuve dans lesquelles on utilise implicitement les propriétés calculatoires de la représentation concrète (sous la forme d'un type inductif) des données. Dans le chapitre 7, nous étudions comment remplacer une représentation concrète par une autre dès lors que l'on a mis en évidence les étapes de preuve dépendantes de la représentation initiale. Finalement, dans le chapitre 8, nous présentons une extension aux types dépendants de notre méthode ainsi qu'une approche alternative.

Conclusions

Le chapitre 9 nous permet de revenir sur les contributions apportées par ces travaux. Finalement, nous présentons rapidement les perspectives de recherche ouvertes par ces travaux.

Table des matières

Chapitre 1

Introduction

Une communauté de chercheurs de plus en plus large considère les preuves formelles comme le meilleur moyen de s'assurer de la correction de développements logiciels et de développements mathématiques. Les preuves formelles permettent d'établir, de manière très rigoureuse, si une application est conforme ou non à sa spécification. Dans cette optique, de nombreux systèmes de preuve ont été développés ; parmi eux on peut citer Coq [Coq02], LEGO [LP92], Nuprl [CAB+86], HOL [GM93], PVS [SOR93], Isabelle-HOL [NPW02], ACL2 [KMM00b, KMM00a] et PhoX [RR03].

Dans cette thèse, nous travaillons avec le système Coq. Le système Coq est l'implantation d'un formalisme logique d'ordre supérieur : le Calcul des Constructions Inductives [CH88, PM93]. Il s'agit d'un λ-calcul typé avec types dépendants [Tho91, NPS90], permettant en outre la définition de constructions inductives.

Dans ce cadre, nous nous intéressons à deux problèmes reliés aux changements de point de vue sur les objets manipulés dans des développements formels : la preuve de correction du programme de calcul de la racine carrée de Gmp et le passage d'une représentation à une autre d'une notion mathématique (les entiers naturels par exemple) dans une théorie développée formellement.

1.1 La racine carrée

Les outils de calcul formel comme Maple [Map03] ou Mathematica [Wol03] sont de plus en plus utilisés pour faire du calcul symbolique. De nombreuses applications, parfois critiques, sont conçues à l'aide de ces outils. Il est donc important de s'assurer de la correction des calculs effectués par ces systèmes. Dans la première partie de cette thèse, nous avons choisi de démontrer la correction de l'implantation du calcul de la racine carrée telle qu'elle est faite dans la bibliothèque de calcul sur les grands entiers Gmp. Depuis, et indépendamment, les algorithmes de calcul sur les nombres entiers de Gmp sont utilisés dans Maple (depuis la version 9).

La bibliothèque Gmp est une bibliothèque de calcul sur les grands entiers. Le programme de calcul de la racine carrée de Gmp a été développé par P. Zimmermann et est l'un des plus efficaces à l'heure actuelle. Dans ce programme, les entiers sont représentés par des segments contigus de la mémoire. Ce programme est écrit en C dans un style impératif. La mémoire est mise à jour de façon destructive à chaque étape de l'algorithme.

Habituellement, les systèmes d'aide à la preuve sont bien adaptés à la preuve de programmes fonctionnels. Au cours de ces dernières années, de nouveaux outils comme Correctness [Fil99] sont apparus et permettent maintenant de faire des preuves de programmes impératifs. Les programmes sont fournis annotés par des formules logiques dans le style de la logique de Floyd-Hoare [Hoa69]. Une pré-condition assure la bonne formation des entrées et une post-condition assure que

le calcul effectué est bien celui qu'on attendait. De plus des annotations peuvent être ajoutées dans le corps du programme pour assurer la propagation correcte des pré- et post-conditions des fonctions mises en jeu. A partir de la donnée de ce programme annoté, un outil comme Correctness engendre une série de conditions de vérification, c'est-à-dire un certain nombre de théorèmes que l'utilisateur devra démontrer. Le formalisme proposé permet de gérer les appels de procédures à l'intérieur du code à certifier. La correction de cette méthode repose sur la modélisation formelle du programme impératif par un programme fonctionnel équivalent. Ce programme prend en arguments des données et des preuves que ces données vérifient bien certaines conditions. Il retourne des données et des certificats exprimant les propriétés attendues pour les données retournées.

En plus des modèles fonctionnel et impératif de calcul de la racine carrée, on peut décrire l'algorithme de calcul de la racine carrée par un modèle abstrait où les données intermédiaires sont simplement reliées les unes aux autres sans donner de procédé de calcul. Ce modèle utilise les entiers $\mathbb{Z}$ pour représenter les grands entiers. Ce choix permet l'utilisation des procédures de décision de Coq pour l'arithmétique linéaire [Coq02, Chap. 15]. Il permet aussi de décomposer le travail de preuve en deux parties. On associe une approche mathématique (où tous les calculs se font sur les entiers relatifs $\mathbb{Z}$, et donc dans laquelle on ne rencontre aucun problème de débordement) avec une description informatique qui prend en compte les difficultés relatives à la représentation concrète (bornée) des entiers.

Cela nous conduit à distinguer deux représentations différentes des entiers utilisés dans le calcul de la racine carrée :

- une représentation mathématique : les entiers relatifs $\mathbb{Z}$ de Coq sans limite (théorique) de taille,
- une représentation informatique : par des segments de tableaux d'entiers bornés comme c'est le cas dans la bibliothèque Gmp.

Cette étude sur la racine carrée fait apparaître le besoin de considérer un même algorithme sous plusieurs points de vue correspondant à plusieurs manières de représenter les données. C'est ce que nous avons étudié dans la deuxième partie de cette thèse.

1.2 Changements de représentation des données dans les théories mathématiques

Dans un système de preuves comme Coq, les types inductifs jouent un rôle majeur dans la description des données. Le choix d'un type inductif particulier pour représenter une notion mathématique donne une manière *canonique* de calculer et de raisonner sur ces données, dans l'esprit du système T de Gödel [Göd58, Göd86]. En effet, les structures de contrôle permettant de programmer dans Coq, à savoir, le filtrage et la récursion structurelle sont construites à partir des définitions inductives des objets considérés.

Nous souhaitons, idéalement, pouvoir dissocier la structure des données et la structure des calculs sur ces données. Dans le cadre du Calcul des Constructions Inductives tel qu'il est implanté dans Coq, il n'est pas possible de proposer des structures de données et des structures de calcul différentes pour une même notion mathématique. Cependant, les outils de haut niveau disponibles dans Coq pour définir des fonctions récursives générales permettent de contourner cette difficulté et de construire des fonctions suivant une structure de calcul différente de la structure des données qu'elles manipulent. Ainsi on peut construire des fonctions dont la structure de calcul est distincte de la structure inductive des données sur lesquelles calculent ces fonctions. Pour cela, ces fonctions sont définies en utilisant un principe de récursion générale sur un ordre bien-fondé, c'est-à-dire un

ordre pour lequel il n'existe pas de chaînes infinies décroissantes. La terminaison de la fonction est assurée par le typage ; en effet, à chaque appel récursif, en plus de la nouvelle entrée sur laquelle se fait la récursion, on doit fournir une preuve que cette nouvelle entrée est bien strictement inférieure à l'entrée précédente pour l'ordre bien fondé considéré.

La récursion bien fondée permet de résoudre le problème posé par la récursion structurelle, mais il faut aussi proposer une solution pour le filtrage. Pour cela, nous définissons un type dépendant de filtrage. Ce type décrit les différentes formes que peut prendre la valeur. Le type est dépendant de l'objet mathématique manipulé, il en donne la forme en fonction des éléments de base de la théorie considérée. Obtenir des équations de point-fixe décrivant le comportement des fonctions nécessite de montrer certaines propriétés d'unicité des habitants pour chacune des instances de ce type de filtrage. Cela peut se faire en utilisant la notion d'égalité dépendante de Coq. C'est une notion d'égalité difficile à manipuler, mais pour laquelle nous décrivons une méthode de travail systématique.

D'autre part, tous les détails d'une preuve n'apparaissent pas dans le terme de preuve correspondant. Il est nécessaire de proposer des outils pour mettre en évidence les étapes de raisonnement implicites (calculs par réduction) dans les termes de preuve. Une fois que l'on dispose de tels outils, on peut changer la représentation concrète d'une donnée (par exemple la représentation *à la Peano* des entiers) dans une théorie développée formellement. Par exemple, on peut construire à partir d'une théorie arithmétique sur les entiers de Peano une nouvelle théorie équivalente sur les entiers binaires.

1.3 Résultats obtenus

Cette thèse apporte deux contributions principales. Elle démontre que les outils de formalisation actuels comme Coq et Correctness sont bien adaptés pour faire des preuves formelles de correction de programmes impératifs. Ces outils permettent de gérer à la fois les programmes ayant des traits impératifs, récursifs et destructifs sur des données comme les tableaux. La preuve de correction du programme de calcul de la racine carrée présentée dans ce document en est un exemple caractéristique. C'est la première fois, à notre connaissance, que le degré de détail de la vérification formelle est si grand. Tous les détails de l'implantation effective en C de l'algorithme sont pris en compte dans cette démonstration, y compris la correction de l'arithmétique de pointeurs utilisée ainsi que celle de la délimitation des zones mémoire affectées par les calculs. Ce travail a servi d'exemple-test conséquent pour l'outil Correctness et a même contribué à son amélioration et au *design* de son successeur Why.

La seconde contribution de ce travail est l'étude du changement de représentation des données dans une théorie développée formellement. Cette étude a permis de proposer un nouvel algorithme de mise en évidence des étapes de raisonnement calculatoires habituellement invisibles dans les termes de preuve. Cet algorithme ainsi que les outils de définition de fonctions récursives non structurelles dans Coq ont permis de transformer (quasi-automatiquement) la bibliothèque Arith de Coq en une nouvelle bibliothèque arithmétique utilisant la représentation binaire des entiers. L'utilisateur n'a qu'à redéfinir les fonctions comme plus, etc. dans la représentation binaire et montrer les propriétés de base. En plus de ces résultats, nous avons étudié comment généraliser les techniques proposées dans le cas non-dépendant au cas dépendant.

Première partie

Preuve formelle de l'implantation de la racine carrée de GMP

Dans cette partie, nous nous intéressons au calcul efficace sur les grands entiers. Calculer efficacement sur de grands entiers a de nombreuses applications. Cela permet, entre autres, de faire du calcul scientifique haute-précision et est utile dans le cadre de la cryptographie. Dans ce cas, on souhaite avoir des opérations efficaces pour pouvoir générer des clés, coder et décoder un message, ou encore casser un code. Par exemple, pour décrypter un message codé avec RSA, il suffit de trouver un moyen de factoriser un grand nombre (constitué de plusieurs dizaines de milliers de chiffres) en deux nombres premiers entre eux. Les bibliothèques de calcul sur les grands entiers servent aussi de support pour le développement de bibliothèque de calcul sur les nombres flottants (par exemple la bibliothèque MPFR [Pol99] qui est basé sur Gmp [Gra02]). Ces bibliothèques sont utilisées dans de nombreuses applications embarquées.

La nécessité d'avoir une arithmétique flottante *fiable* est apparue avec les incidents survenus sur des systèmes basés sur l'arithmétique. On peut citer entre autres :

- les accumulations d'erreurs d'arrondis pour un missile Patriot (qui a conduit cet antimissile à rater le missile qui venait en face ...) [Inf92],
- le bug de l'algorithme de division du Pentium [Mul95],
- ou l'échec du vol 501 d'Ariane 5 [L$^+$96].

Un des moyens utilisés pour s'assurer de la fiabilité de l'arithmétique utilisé dans les systèmes embarqués est de formaliser les algorithmes et les programmes les implantant dans des systèmes d'aide à la preuve comme Coq [Coq02], HOL [GM93], PVS [SOR93] ou ACL2 [KMM00b, KMM00a].

Dans la suite de ce document, nous présentons une preuve formelle de correction de l'implantation d'un programme de calcul efficace de la racine carrée pour les grands entiers [Zim99]. Ce programme, qui fait partie de la bibliothèque de calcul en précision arbitraire Gmp, est entièrement prouvé correct dans le système Coq. Les preuves sont réalisées en utilisant l'outil Correctness pour gérer les traits impératifs du programme.

Dans le chapitre 2, nous présentons une description formelle de la bibliothèque Gmp de calcul sur les grands entiers. Cette spécification est décrite en utilisant le formalisme de l'outil Correctness distribué avec le système Coq.

Dans le chapitre 3, nous décrivons l'algorithme classique d'extraction à la main d'une racine carrée. Nous présentons ensuite l'algorithme efficace de calcul de la racine carrée pour les grands entiers proposé par P. Zimmermann. Nous présentons également les implantations mpn_dq_sqrtrem et mpn_sqrtrem de cet algorithme dans la bibliothèque de calcul en précision arbitraire Gmp. Enfin, nous décrivons une première version de la spécification et de la preuve de correction de cet algorithme.

Dans le chapitre 4, nous présentons la spécification finalement adoptée pour l'algorithme de calcul de la racine carrée ainsi que la preuve formelle de correction correspondante. Nous décrivons ensuite la preuve formelle de correction de son implantation (la fonction mpn_dq_sqrtrem en utilisant les outils Coq et Correctness. Le développement est assez gros (de l'ordre de 13000 lignes) et nécessite l'utilisation de techniques avancées de maintenance et de réutilisation des preuves. En particulier, nous utilisons une approche à trois niveaux qui permet une décomposition du travail de preuve en plusieurs étapes distinctes, plus facile à gérer.

Table des matières

Chapitre 2

Spécification formelle de la bibliothèque GMP

Dans ce chapitre, nous présentons la bibliothèque de calcul sur les grands entiers Gmp. Nous donnons les spécifications formelles des opérations de base de cette bibliothèque en utilisant le formalisme de Coq et de l'outil Correctness. Cette description formelle est réalisée à partir de la documentation informelle de Gmp [Gra02]. Elle sert de base à la preuve de correction de l'implantation du calcul de la racine carrée de Gmp présentée dans le chapitre 4. Elle peut néanmoins être réutilisée sans difficulté pour prouver la correction d'autres programmes utilisant les fonctions de la classe mpn.

2.1 La bibliothèque GMP

La bibliothèque Gmp[1] [Gra02] est une bibliothèque de calcul arithmétique multiprécision sur les entiers, les rationnels et les nombres flottants. Ses opérations permettent de calculer sur des nombres arbitrairement grands et sont conçues pour s'exécuter le plus rapidement possible. La bibliothèque Gmp est reconnu comme une des meilleures bibliothèques de calcul sur les grands entiers à l'heure actuelle.

Elle comporte plusieurs couches. En plus des opérations arithmétiques de haut niveau sur les nombres flottants, elle contient une classe de fonctions de bas niveau ; les fonctions de cette classe ont un nom commençant par mpn et traitent les entiers naturels codés dans des tableaux de mots-mémoire. Un mot-mémoire (limb dans la terminologie de Gmp) ne correspond pas nécessairement à un mot-machine ; par exemple sur les processeurs MIPS, où un mot-machine est composé de 64 bits, un bloc peut être de longueur 32.

Pour des raisons d'efficacité, la majorité de ces fonctions sont écrites en assembleur (par exemple sur les plate-formes x86). Cependant les deux fonctions mpn_sqrtrem et mpn_dq_sqrtrem dont nous démontrerons la correction dans le chapitre 4 sont écrites en C.

[1] *GNU Multiple Precision Arithmetic Library*

2.2 L'outil Correctness

2.2.1 Preuves formelles de programmes impératifs

L'outil Correctness permet de développer, dans le système Coq, des preuves de correction totale de programmes ayant des traits impératifs. Prouver la correction totale d'un programme consiste à montrer, d'une part que ce programme termine toujours, et d'autre part que le résultat retourné vérifie bien la spécification du programme.

L'outil Correctness a été développé comme une commande interne à Coq par Jean-Christophe Filliâtre [Fil99, Fil01b] au cours de sa thèse de doctorat. Depuis, cet outil a été dissocié de Coq et est devenu Why [Fil03] : un système qui permet de faire des preuves de programmes C ou ML et de générer des obligations de preuves pour différents démonstrateurs, parmi eux Coq [Coq02], PVS [SOR93], HOL-Light [Har98], haRVey [DR02].

Du point de vue de l'utilisateur, l'outil Correctness prend en entrée un programme écrit dans un langage dédié proche de ML, et pouvant contenir des constructions impératives (des affectations, des structures de contrôle comme les boucles par exemple). Ce programme peut contenir des annotations dans le style de la logique de Floyd-Hoare [Hoa69] et est accompagné d'une spécification de son comportement attendu.

L'outil produit, à partir de ces données, une série d'énoncés qui correspondent aux propriétés logiques à vérifier pour s'assurer de la correction du programme. Ces énoncés peuvent alors être démontrés par l'utilisateur dans le système Coq.

D'un point de vue plus technique, le système fonctionne de la manière suivante : l'outil Correctness analyse statiquement les effets de bord du programme fourni en entrée. A partir de cette information, il construit une traduction fonctionnelle. Par exemple, si le programme considéré accède à la référence x en lecture seule et la référence y en lecture/écriture, et retourne une valeur v, alors la traduction fonctionnelle sera une fonction que prend en entrée les valeurs de x et y et retourne en plus de la valeur v calculée, la nouvelle valeur de y.

De manière générale, la spécification d'un programme p peut s'écrire de la façon suivante :

$$\forall x.\ P(x) \Rightarrow\ \exists y.\ Q(x, y) \tag{2.1}$$

où x représente le vecteur des entrées et y le vecteur des sorties. P est la pré-condition de ce programme, c'est-à-dire la propriété que doit vérifier le vecteur d'entrées x pour que le programme calcule effectivement un vecteur de sorties y tels que x et y soient reliés par la relation Q. On dit que Q est la post-condition du programme p.

Prouver la correction totale d'un tel programme p consiste à montrer que la traduction fonctionnelle $\hat{p}$ de p vérifie bien la spécification donnée en (2.1). Un méta-théorème, présenté dans [Fil99] permet de déduire de la correction de la traduction fonctionnelle par rapport à la spécification initiale que le programme initial vérifie bien la spécification qui lui est associée.

Pour une description plus précise du fonctionnement interne de la commande Correctness, le lecteur intéressé pourra se reporter à la thèse de doctorat [Fil99] de Jean-Christophe Filliâtre.

2.2.1.1 Différences sémantiques entre C et ML

Le langage de description de programme de Correctness est un langage spécifique proche de ML. Les fonctions de la bibliothèque de Gmp que nous souhaitions étudier sont écrites en C. Cependant, pour ces fonctions, la sémantique à la ML est assez proche de la sémantique du langage C. C'est pourquoi nous avons pu utiliser cet outil pour prouver les programmes de Gmp.

Afin d'éviter tout problème, quelques subtilités de la sémantique de C ont été évitées. En particulier, nous avons transformé toutes les instructions de la forme `c+=a` en des instructions de la forme `c=c+a` dont la sémantique est plus claire. De plus, les opérations bit-à-bit ont seulement été modélisées au niveau arithmétique plutôt qu'au niveau binaire. Par exemple, l'opération de décalage vers la gauche ou la droite a été modélisée uniquement en terme de division ou de multiplication par une puissance de 2. D'une certaine manière, on peut dire que les opérations de décalage ne sont pas prouvées formellement, mais que l'on suppose leur correction.

2.2.2 Exemples de développement

Bien qu'étant un outil récent, CORRECTNESS a été utilisé pour développer des preuves de programmes non triviaux ; parmi eux, nous pouvons citer la preuve formelle du programme Find [Fil01a] et les preuves de programmes de tri en place — tri par insertion, tri rapide (quicksort) et tri en tas — [FM99].

Dans le but de montrer comment s'utilise l'outil CORRECTNESS, nous étudions un exemple *jouet* : la preuve de correction d'un programme naïf de recherche de l'indice du minimum dans un tableau d'entiers.

Le programme considéré consiste simplement à parcourir le tableau de la gauche vers la droite en gardant en mémoire l'indice du plus petit élément rencontré jusqu'ici et en le comparant avec l'élément courant du tableau. Le code de ce programme ainsi que les annotations nécessaires pour obtenir des obligations de preuve démontrables est donné à la figure 2.1.

Ce programme prend en argument un entier N et un tableau v de longueur N ; il retourne un entier m. L'unique pré-condition $\{0 < N\}$ de ce programme assure que le tableau v a au moins un élément. La post-condition exprime que tous les éléments du tableau sont bien supérieurs ou égaux à l'élément d'indice m.
Le corps de ce programme est une boucle `while` qui itère sur le tableau de 0 à $N - 1$. Un invariant de cette boucle est donné par la formule suivante :

$$0 \leq i \leq N \quad \wedge \quad 0 \leq m < N \quad \wedge \quad \forall j : \mathbb{Z}.\ 0 \leq j < i \rightarrow v[m] \leq v[j]$$

Cette formule spécifie que i et m restent bien dans les limites du tableau. i peut atteindre N (sinon le test de sortie de boucle ne s'évaluerait jamais à `false`. Cette formule exprime aussi que l'élément d'indice m est bien plus petit que tous les éléments du sous-tableau dont les indices sont compris entre 0 et i. Le variant $N - i$ est une quantité qui diminue à chaque itération de la boucle `while`, ce qui assure que le programme termine.

L'exécution de la commande CORRECTNESS sur ce programme engendre 8 obligations de preuve. Nous donnons une traduction informelle des énoncés de ces obligations de preuve. Celles-ci correspondent aux règles classiques de preuves de programme en Logique de Hoare. Par exemple, la sixième obligation consiste à montrer, sachant que l'invariant est vérifié et que le test de la boucle `while` est `faux`, que la post-condition de la boucle `while` est bien vérifiée.

1. La première obligation de preuve est une simple vérification des conditions de borne pour l'accès à un élément d'un tableau.
2. Idem.
3. Ici, on doit montrer que la post-condition de l'instruction conditionnelle `if` peut bien être déduite de l'invariant de boucle quand le test s'évalue à `vrai`.

```
Correctness mymin fun (N:Z)(v:array N of Z) ->
{‘0<N‘}
let i = ref ‘0‘ in let m = ref ‘0‘ in
begin
  while (!i < N)
  do
    {invariant ‘0<=i<=N‘/\‘0<=m<N‘/\(j:Z)(‘0<=j<i‘)->‘(access v m)<=(access v j)‘
     variant ‘N-i‘}
    (if (v[!i] < v[!m]) then m := !i)
        {‘0<=i<=N‘/\‘0<=m<N‘/\(j:Z)(‘0<=j<=i‘)->‘(access v m)<=(access v j)‘};
    i := (Zs !i)
  done
  {‘N = i‘ /\(j:Z)(‘0 <= j < i‘ -> ‘(access v m) <= (access v j)‘)};
  !m
end
{(j:Z)(‘0 <= j < N‘) -> ‘(access v result) <= (access v j)‘}.
```

FIG. 2.1 – Programme annoté de recherche de l'indice du minimum d'un tableau d'entiers

4. Même obligation de preuve dans le cas où le test s'évalue à faux.
5. On doit montrer, à partir de la post-condition du if, que l'invariant est bien maintenu et que le variant diminue strictement à chaque itération de la boucle while.
6. On doit montrer que lors de la sortie de boucle, on peut bien établir la post-condition suivante : `{‘N = i‘/\(j:Z)(‘0 <= j < i‘ -> ‘(access v m) <= (access v j)‘)}`.
7. On doit montrer que l'invariant est vérifié au moment de l'entrée dans la boucle en utilisant la pré-condition du programme `{‘0<N‘}`.
8. Enfin, on doit montrer que la post-condition du programme peut se déduire de la post-condition de la boucle while.

2.3 Modélisation

Dans cette section, nous présentons la modélisation choisie pour décrire les opérations de la classe mpn de GMP. Nous avons choisi de modéliser les opérations au niveau arithmétique, sans prendre en compte le codage binaire des entiers. De plus, nous avons choisi d'avoir une représentation globale de la mémoire sous la forme d'un seul tableau représentant linéairement toute la mémoire. Ce modèle s'adapte bien à la forme des arguments des opérations de GMP où l'on décrit un nombre par la position de son mot-mémoire de poids faible et le nombre de mots-mémoire qui le compose.

Pour chacune des fonctions dont nous donnons la spécification formelle dans COQ, nous donnons à titre de comparaison sa spécification informelle telle qu'on la trouve dans la documentation [Gra02] de GMP.

2.3.1 Modélisation de la mémoire

Dans notre modèle d'exécution des programmes, la mémoire est représentée par un tableau global m de longueur fixe bound. Ses indices vont de 0 à bound $-$ 1. Les pointeurs de C sont

`mp_limb_t` **mpn_sub_n** (`mp_limb_t *`*rp*, `const mp_limb_t *`*s1p*, `const mp_limb_t *`*s2p*, `mp_size_t` *n*) Function

Subtract {*s2p*, *n*} from {*s1p*, *n*}, and write the *n* least significant limbs of the result to *rp*. Return borrow, either 0 or 1.

This is the lowest-level function for subtraction. It is the preferred function for subtraction, since it is written in assembly for most CPUs.

FIG. 2.2 – Extrait de la documentation de GMP

remplacés par des indices dans ce tableau ; les longueurs sont, quant à elles, de simples entiers naturels. Chaque case de ce tableau contient un mot-mémoire, c'est à dire un entier de taille bornée (compris entre 0 et β). Nous définissons le type $\mathsf{mod}\mathbb{Z}$ pour décrire de telles données.

$$\mathsf{mod}\mathbb{Z} \equiv \{v : \mathbb{Z} \mid 0 \leq v < \beta\}.$$

Les mots-mémoires ne peuvent donc contenir que des nombres positifs bornés par β. Nous fournissons une fonction $\mathsf{mod}\mathbb{Z}\_\mathsf{to}\_\mathbb{Z}$ qui injecte les éléments de $\mathsf{mod}\mathbb{Z}$ dans $\mathbb{Z}$.

2.3.2 Représentation des grands entiers

Nous définissons une fonction d'interprétation des segments de la mémoire : $m, pos, l \mapsto \{pos, l\}_m$. Cette fonction retourne l'entier représenté par un segment de la mémoire. Elle a trois arguments, le tableau global dénotant la mémoire, la position p de début du segment et sa longueur l. p et l sont des entiers naturels, ce qui permettra par la suite de faire facilement du raisonnement par récurrence sur ces données. La fonction d'interprétation est définie par ces deux équations :

$$\begin{aligned} \{pos, 0\}_m &= 0, \\ \{pos, l+1\}_m &= \mathsf{mod}\mathbb{Z}\_\mathsf{to}\_\mathbb{Z}(m[pos]) + \beta\{pos+1, l\}_m. \end{aligned}$$

A partir de cette définition, il est possible de prouver que déposer deux nombres a et b côte-à-côte dans la mémoire suffit à construire un nombre de la forme $a\beta^l + b$:

$$\mathsf{Impnumber\_decompose} : \forall m, h, l, p.\ \{p, l\}_m + \beta^l\{p+l, h\}_m = \{p, l+h\}_m.$$

Ce théorème est facilement démontré par induction sur l.

2.3.3 Comportement calculatoire des fonctions

Maintenant que nous disposons d'un modèle de la mémoire, nous allons décrire les propriétés des opérations de la classe mpn de GMP. L'outil CORRECTNESS permet de déclarer des signatures de fonctions sans fournir de définitions effectives. Par exemple, une version simplifiée (i.e ne prenant pas en compte la gestion de la mémoire) de la fonction de soustraction mpn_sub_n, dont la spécification informelle est donnée dans la figure 2.2, peut être définie de la manière suivante :

```
Global  Variable mpn_sub_n :
        fun (pos_r : nat)(pos_a : nat)(pos_b : nat)(l : nat)
        returns _ : unit
        reads m
        writes m, rb
        post {pos_r, l}_m = (rb · β^l + {pos_a, l}_m@) − {pos_b, l}_m@
end.
```

Cet extrait de code exprime que mpn_sub_n prend en entrée quatre arguments pos_r, pos_a, pos_b, et l, ne retourne aucune valeur et peut avoir des effets de bords sur la mémoire m et sur la variable booléenne rb. Les nombres fournis en entrée sont stockés à l'adresse $\{pos_a, l\}$ (resp. $\{pos_b, l\}$) pour le premier (resp. deuxième) argument. La valeur calculée est stockée à l'adresse $\{pos_r, l\}$. Dans cette description, il n'y a aucune pré-condition, cependant nous en ajouterons une plus tard lorsque nous traiterons la validité des accès à la mémoire. Une post-condition minimale pour mpn_sub_n indique que cette opération soustrait la valeur du nombre stocké dans la mémoire initiale (notée $m@$) à l'adresse pos_a et de longueur l à la valeur du nombre stockée à l'adresse pos_b de longueur l. Le résultat est rangé à l'adresse pos_r dans la mémoire finale (m) à l'exception de la retenue éventuelle qui est stockée dans la variable globale rb. Cette variable sera positionnée à true lorsque le second argument sera plus grand que le premier, et donc que le résultat du calcul sera un nombre négatif. Le résultat est rendu positif en lui ajoutant $rb \cdot \beta^l$, ce qui est cohérent avec la représentation habituelle des nombres négatifs en arithmétique des ordinateurs.

2.3.3.1 Gestion de la retenue

La manière de gérer la retenue peut paraître un peu étrange. Une solution plus élégante aurait été de retourner la retenue comme c'est fait dans le code C. Notre approche ne permet pas de décrire directement les propriétés algébriques dans lesquels apparaissent des appels de fonction. La raison principale de ce choix d'implantation est un problème technique que nous avons rencontré au début de notre travail. Nous avons cependant testé la première solution.

Dans la solution que nous avons adoptée dans la suite, une expression[2] comme

```
b = q + mpn_sub_n (np, np, np + n, 2 * l);
```

est transformée en les deux lignes suivantes :

```
mpn_sub_n (np, np, np + n, 2 * l);
b = q + (bool_to_Z rb);
```

La transformation est la suivante : une fonction C qui retourne une valeur est transformée en suivant les deux règles suivantes :

- on commence par exécuter les pas contenant des effets de bord,
- ensuite on évalue une expression ne contenant plus aucun effet de bord.
  Cette technique présente au moins un avantage : elle rend explicite l'ordre d'appel des fonctions ayant des effets de bords. C'est important dans notre étude puisque nous modélisons un programme C (où l'évaluation des arguments d'une fonction se fait généralement de gauche à droite) dans le langage utilisé par Correctness où l'évaluation se fait de droite à gauche.

2.3.4 Gestion de la mémoire

Pour les fonctions ayant des effets de bord, il n'est pas suffisant d'exprimer leurs propriétés arithmétiques. Il faut aussi s'assurer que tous les accès à la mémoire sont valides et que seule la zone de mémoire spécifiée est modifiée. Dans la post-condition de la fonction mpn_sub_n, $m@$ représente la mémoire avant l'exécution du code et m dénote la mémoire après la soustraction. La post-condition proposée pour la fonction mpn_sub_n n'est pas complète car elle ne caractérise pas les segments de la mémoire qui ne sont touchés par aucun effet de bord. Dans ce paragraphe, nous allons rajouter à la spécification les contraintes nécessaires pour assurer cette propriété.

[2] Cette expression correspond à la ligne 21 du programme de calcul de la racine carrée (voir figure 3.3) que nous présentons dans le chapitre 3.

2.3.4.1 Accès à la mémoire

La mémoire a été déclarée comme un tableau de taille fixe bound. Par conséquent, tout accès à une position inférieure strictement à 0 ou supérieure à bound n'est pas valide.

Dans notre exemple mpn_sub_n, nous devons vérifier les propriétés suivantes :

$$pos_a + l \leq \mathsf{bound}, \qquad pos_b + l \leq \mathsf{bound}, \qquad pos_r + l \leq \mathsf{bound}.$$

Comme pos_a, pos_b, pos_r, et l ont été déclarés de type $\mathbb{N}$, la condition de positivité est vérifiée par définition. Les conditions précédentes sont, quant à elles, ajoutées dans les pré-conditions de mpn_sub_n ainsi que dans la description formelle de la plupart des fonctions de GMP.

Par exemple, pour la fonction mpn_dq_sqrtrem, nous devons vérifier les pré-conditions suivantes :

$$n_p + 2n \leq \mathsf{bound}, \qquad s_p + n \leq \mathsf{bound}.$$

De ces deux pré-conditions, nous serons capables d'établir la validité de tous les accès à la mémoire dans la suite du programme. Vérifier la validité de ces accès consistera simplement à vérifier la validité d'inégalités linéaires sur les indices et les longueurs des segments de la mémoire. De tels inégalités peuvent être démontrées automatiquement dans COQ en utilisant la tactique Omega qui est un procédure de décision pour l'arithmétique linéaire avec inégalités. Nous avons construit, au dessus de la tactique Omega, une nouvelle tactique SolveBounds qui ajoute dans le contexte des informations utiles sur les indices des tableaux représentant la mémoire avant d'appeler la tactique Omega. Cette tactique nous permet de montrer automatiquement toutes les propriétés reliées à la gestion de la mémoire dans GMP.

```
Tactic Definition SolveBounds b :=
(Abstract (Try Unfold no_overlap ; Repeat Rewrite two_2;Simpl;
               Repeat Rewrite <- (plus_sym 0);Simpl;
               Repeat Rewrite inj_plus ; Repeat Rewrite inj_minus1 ;
               Repeat Rewrite inj_minus_div2;
               Generalize (div2_le b);Intros ; Omega)).
```

2.3.4.2 Chevauchement des données

La spécification de la plupart des fonctions de la classe mpn de GMP exige que les segments de la mémoire représentant les entrées et les sorties soient exactement les mêmes ou bien qu'elles ne se chevauchent absolument pas. Dans notre exemple mpn_sub_n, cela se traduit par les pré-conditions suivantes :

$$pos_r = pos_a \vee pos_r + l \leq pos_a \vee pos_a + l \leq pos_r$$
$$pos_r = pos_b \vee pos_r + l \leq pos_b \vee pos_b + l \leq pos_r$$
$$pos_a = pos_b \vee pos_a + l \leq pos_b \vee pos_b + l \leq pos_a$$

Toutes les contraintes issues de ces spécifications peuvent être démontrées automatiquement en utilisant la tactique SolveBounds.

2.3.4.3 Stabilité de la mémoire

La plupart des fonctions laissent une grande partie de la mémoire inchangée. Le segment de la mémoire qui reçoit le résultat est bien évidemment modifié, mais il faut s'assurer que les effets de bord de la fonction sont bien limités à ce(s) segment(s). En effet, il est important de s'assurer que les données utilisées à un instant donné n'ont pas été altérées aux cours des opérations précédentes.

La vérification de la stabilité de la mémoire est une propriété nouvelle qui apparaît avec le développement de preuves formelles. Dans les preuves sur papier, il est implicite que les zones de la mémoire pour laquelle nous n'avons aucune information ne sont pas modifiées.

Dans notre exemple mpn_sub_n, nous devons ajouter la post-condition suivante :

$$\forall p : \mathsf{nat}.\ 0 \leq p < \mathsf{bound} \Rightarrow p < pos_r \vee pos_r + l \leq p \Rightarrow m[p] = m@[p]$$

où $m[p]$ représente la valeur de la mémoire à l'indice p après l'exécution de l'opération et $m@[p]$ sa valeur avant l'exécution.

De telles conditions permettent de transformer une preuve d'égalité entre deux valeurs dans des états mémoires différents en un ensemble de comparaisons facilement prouvables au moyen de la tactique SolveBounds.

2.3.5 Représentation des opérations binaires

Les opérations binaires comme le *ou* logique et le décalage vers la gauche ou la droite sont modélisées au niveau arithmétique. Ce sont les interprétations des segments de la mémoire qui sont reliées entre elles et non les valeurs binaires effectives. Dans cette section, nous présentons la spécification formelle de l'opérateur logique or et de l'opérateur de décalage vers la droite mpn_rshift qui sont utilisés dans la preuve de correction de la racine carrée de GMP.

2.3.5.1 L'opérateur binaire or

Sous certaines conditions, l'opérateur binaire or appliqué à deux mots permet de les additionner. Lorsque l'une des données est de la forme $\beta' \cdot x$ et que l'autre est strictement inférieur à β', un simple *ou* logique permet de faire l'addition de ces deux nombres. En effet, pour chaque position dans les mots, au plus un des deux bits concernés est à 1 et l'autre à 0. Par conséquent, l'opérateur d'addition binaire et le *ou* logique coïncident dans ces conditions.

Nous avons donc spécifié l'opérateur or en exprimant directement la propriété arithmétique attendue pour cette opération. Cela conduit à la spécification suivante :

$$\begin{array}{ll}
\mathsf{Global} & \mathsf{Variable\ I_or} : \quad (\beta' = \beta/2) \\
& \mathsf{fun}\ (p : \mathsf{nat})(y : \mathsf{mod}\mathbb{Z}) \\
& \mathsf{returns}\ w : \mathsf{mod}\mathbb{Z} \\
& \mathsf{reads}\ m \\
& \mathsf{pre}\ (\mathsf{mod}\mathbb{Z}to\mathbb{Z}\ m[p]) < \beta' \\
& \quad \wedge\ \exists yb : \mathbb{Z} \mid 0 \leq yb \leq 1 \rightarrow (\mathsf{mod}\mathbb{Z}to\mathbb{Z}\ y) = yb \times \beta' \\
& \mathsf{post}\ (\mathsf{mod}\mathbb{Z}to\mathbb{Z}\ w) = (\mathsf{mod}\mathbb{Z}to\mathbb{Z}\ m[p]) + (\mathsf{mod}\mathbb{Z}to\mathbb{Z}\ y) \\
\mathsf{end}.
\end{array}$$

Cette spécification est partielle, elle ne décrit le comportement de l'opération que pour les entrées vérifiant une certaine pré-condition. Comme ces conditions seront réunies quand nous utiliserons cette spécification dans la preuve de l'implantation de la racine carrée, cela ne posera aucun problème.

`mp_limb_t` **mpn_rshift** (`mp_limb_t *`*rp*`, const mp_limb_t *`*sp*`, mp_size_t` *n*`, unsigned int` *count*) Function

Shift {*sp*, *n*} right by *count* bits, and write the result to {*rp*, *n*}. The bits shifted out at the right are returned in the most significant *count* bits of the return value (the rest of the return value is zero).

count must be in the range 1 to `mp_bits_per_limb`−1. The regions {*sp*, *n*} and {*rp*, *n*} may overlap, provided $rp \leq sp$.

This function is written in assembly for most CPUs.

FIG. 2.3 – Extrait de la documentation de GMP

2.3.5.2 L'opérateur de décalage à droite

L'opérateur de décalage vers la droite mpn_rshift permet de décaler un bloc (une séquence de mots) d'un certain nombre de bits vers la droite. Cette opération permet de faire des opérations de division par des puissances de 2 très efficacement. Au niveau arithmétique, l'opération de décalage vers la droite est donc interprétée comme une division par une puissance de 2. Nous allons utiliser cette interprétation pour décrire formellement mpn_rshift dans COQ/CORRECTNESS. Pour des raisons d'homogénéité de la spécification, nous avons considéré que, comme pour les autres fonctions, les entrées et les sorties de mpn_rshift ne se chevauchent jamais ; soit elles dénotent les mêmes segments de la mémoire, soit elles sont complètement disjointes.

Global Variable mpn_rshift :
fun $(pos_a : \mathsf{nat})(pos_b : \mathsf{nat})(l : \mathsf{nat})(count : \mathsf{nat})$
returns $lower : Z$
reads m
writes m
pre $pos_b \leq pos_a \wedge pos_a + l \leq \mathsf{bound} \wedge pos_b + l \leq \mathsf{bound}$
$\wedge(\mathsf{no\_overlap}\ pos_a\ pos_b\ l\ l) \wedge O < l$
post $\{pos_a, l\}_m = (\mathsf{iter}\ \mathbb{Z}\ \mathsf{Zdiv2}\ count\ \{pos_b, l\}_{m@})$
$\wedge\{pos_b, l\}_{m@} = 2^{count} \cdot \{pos_a, l\}_m + lower$
$\wedge\forall p : \mathsf{nat}.0 \leq p < \mathsf{bound} \rightarrow (p < pos_a \vee pos_a + l \leq p) \rightarrow m[p] = m@[p]$
end.

L'opération mpn_rshift est utilisée dans le code de la fonction de calcul de la racine carrée de GMP. Dans la preuve de correction de cette fonction C, nous avons utilisé une spécification plus spécialisée dans le cas où *count* vaut 1. L'opération mpn_rshift2 divise son entrée par 2 en effectuant un simple décalage de *un* bit vers la droite. Sa spécification est la suivante :

Global Variable mpn_rshift2 :
fun $(pos_a : \mathsf{nat})(pos_b : \mathsf{nat})(l : \mathsf{nat})$
returns $tt : \mathsf{unit}$
reads m
writes m
pre $pos_b \leq pos_a \wedge pos_a + l \leq \mathsf{bound} \wedge pos_b + l \leq \mathsf{bound}$
$\wedge(\mathsf{no\_overlap}\ pos_a\ pos_b\ l\ l)$
post $\{pos_b, l\}_{m@} = 2 * \{pos_a, l\}_m + (\mathsf{mod}\mathbb{Z}to\mathbb{Z}\ (\mathsf{isEvenOrOdd}\ m@[pos_b]))$
$\wedge\forall p : \mathsf{nat}.0 \leq p < \mathsf{bound} \rightarrow (p < pos_a \vee pos_a + l \leq p) \rightarrow m[p] = m@[p]$
end.

La fonction isEvenOrOdd prend en argument v un entier borné (de type mod$\mathbb{Z}$) et retourne 1 encodé dans le type mod$\mathbb{Z}$ si v est impair, 0 sinon. Cette fonction permet de déterminer le bit de poids faible du mot $\{pos_b, l\}_m$.

C'est la spécification mpn_rshift2 que nous utilisons dans la preuve de correction de la racine carrée de GMP. Cependant, nous avons démontré que la fonction mpn_rshift2 peut se coder à partir de la fonction mpn_rshift. Cela s'exprime en CORRECTNESS par la commande suivante :

```
Correctness mpn_rshift2
  fun (pos_a:nat)(pos_b:nat)(l:nat) ->
  {‘pos_b<=pos_a‘
      /\‘pos_a+l<=bound‘
      /\‘pos_b+l<=bound‘
      /\(no_overlap pos_a pos_b l l)/\(lt 0 l)}
  (mpn_rshift pos_a pos_b l (S 0))
{‘(I m@ pos_b l) = 2*(I m pos_a l)+(modZtoZ (isEvenOrOdd m@ pos_b))‘
 /\(p:nat)‘0<=p<bound‘->((lt p pos_a)\/(le (plus pos_a l) p))->
          (access m p)=(access m@ p)
}.
```

Dans cette spécification, le bit de poids fort expulsé du mot au moment du décalage n'est pas conservé, mais on peut le déterminer en cherchant la parité de l'entier codé dans le segment de la mémoire donné en entrée.

2.4 Travaux connexes

2.4.1 Algorithme de division de GMP

Dans le cadre de l'Action de Recherche Coopérative *Arithmétique des Ordinateurs Certifiée*, la correction du programme de division des grands entiers a été démontrée par Didier Bondyfalat [Bon02]. Cette formalisation a nécessité de spécifier les opérations de base de la bibliothèque GMP. Le niveau de détail choisi est beaucoup moins grand que celui-ci que nous avons utilisé pour décrire la bibliothèque GMP dans ce chapitre. Par exemple, l'opération de soustraction mpn_sub_n est spécifiée beaucoup plus simplement :
Pour tout a et b de longueur n, si

$$(r, c) = \mathsf{minuswc}(n, a, b)$$

alors

$$a - b = r - c\beta^n \quad \wedge \quad r < \beta^n \quad \wedge \quad c = 0 \vee c = 1$$

Ici, a et b sont des entiers naturels. Pour calculer la différence entre a et b, il suffit de comparer a et b et d'ajouter β^n à a si sa valeur est strictement inférieure à b avant de faire la soustraction. Une telle spécification permet de rendre compte des problèmes comme les débordements, mais ne permet pas de rendre compte de la consommation mémoire d'une fonction comme mpn_sub_n.

2.4.2 Arithmétique réelle exacte

Dans sa thèse [MM94], V. Ménissier-Morain a proposé une implantation en précision arbitraire de l'arithmétique réelle exacte. Elle a proposé une implantation de l'arithmétique rationnelle, puis deux implantations des nombres réels calculables en utilisant les nombres B-adiques ou les fractions continues. Récemment des expériences ont été faites en vue de prouver la correction des algorithmes proposés pour l'arithmétique réelle exacte avec les nombres B-adiques [Cre02].

2.4.3 Formalisation de la norme IEEE-754

De nombreux chercheurs ont travaillé sur le thème de la formalisation des opérations arithmétiques dans les assistants de preuve. La plupart se sont intéressés à l'arithmétique flottante et à la norme IEEE-754, parmi eux [Har99, Rus99, Min95, DRT01, Jac01]. La vérification d'algorithmes sur les nombres flottants est plus centrée sur la correction arithmétique des algorithmes. Les questions d'erreurs d'arrondis y sont plus importantes. Dans l'arithmétique entière, le risque d'erreur est moindre quant à la correction des calculs. Cependant, une telle description formelle permet de s'assurer de leur correction, de plus elle permet de vérifier que tous les détails de l'implantation n'introduisent pas d'erreurs dans l'algorithme. P. Loiseleur a mené la première expérience de formalisation en Coq de la norme IEEE-754 dans Coq en 1997 [Loi97]. Depuis, M. Daumas, L. Rideau, et L. Théry ont proposé une formalisation de l'arithmétique flottante dans le système Coq [DRT01]. Ils ont développé une bibliothèque pour une base quelconque (non nécessairement une puissance de 2).

2.5 Conclusion

Dans ce chapitre, nous nous sommes concentrés sur deux opérations de la bibliothèque Gmp, à savoir mpn_sub_n et mpn_rshift ainsi que sur la spécification d'une opération d'addition sous certaines conditions, implantée à l'aide de la construction logique *ou*. En fait, la plupart des opérations de la classe mpn ont été modélisées en utilisant le formalisme de Coq/Correctness. De plus cette spécification a été récemment porté vers le nouvel outil de vérification de programmes Why qui remplace Correctness.

Maintenant que nous disposons d'une spécification formelle de la bibliothèque Gmp, nous allons pouvoir montrer la correction totale de son programme de calcul de la racine carrée. Dans le chapitre suivant, nous revenons sur la façon de calculer efficacement la racine carrée d'un entier. Finalement, dans le chapitre 4, nous présentons la preuve du programme de calcul de racine carrée de Gmp telle qu'elle a été développée formellement avec les outils Coq et Correctness.

Chapitre 3

La racine carrée de GMP

Dans ce chapitre, nous commençons par présenter la méthode d'extraction à la main d'une racine carrée entière telle qu'elle était autrefois enseignée à l'école. Nous décrivons ensuite l'algorithme proposé par Paul Zimmermann [Zim99] pour l'extraction efficace de la racine carrée ainsi que son implantation dans la bibliothèque de calcul GMP. Enfin nous présentons la première version de la preuve de correction, au niveau abstrait, de cet algorithme de calcul de racine carrée.

3.1 Extraction d'une racine carrée à la main

Considérons un entier naturel N. Extraire sa racine carrée consiste à retourner un couple d'entiers (S,R) tel que $S = \lfloor\sqrt{N}\rfloor$ et $R = N - S^2$. S est la racine carrée entière de N et R le reste correspondant.

Nous reprenons l'algorithme d'extraction de la racine carrée tel qu'il était enseigné à l'école avant l'apparition des calculatrices dans les salles de classes. Nous présentons cette méthode sur un exemple : l'extraction de la racine carrée de 7421. Le calcul à la main se présente à la manière qu'une division.

$$\begin{array}{r|l} 7421 & 86 \\ \hline 1021 & 166 \cdot 6 = 996 \\ 25 & \end{array}$$

Extraire la racine carrée de $n = 7421$ consiste à effectuer les étapes de calcul suivantes :

1. On découpe n en blocs de deux chiffres : $n = 100n' + n''$ avec ($n' = 74$ et $n'' = 21$).
2. On calcule la racine carrée du nombre formé par les deux chiffres de poids fort n'. Ici n' vaut 74 ; cela donne donc $s' = 8$ et $r' = 10$. On place $s' = 8$ en haut à droite du schéma et $r' = 10$ en dessous de 74.
3. On abaisse le deuxième bloc de deux chiffres du reste $100r' + n''$; dans notre exemple, cela donne 1021.
4. On cherche un chiffre x tel que l'expression $p = (2s' \cdot 10 + x) \cdot x$ soit le plus grand nombre plus petit ou égal à $100r' + n''$. Dans notre exemple, on obtient $x = 6$.
5. On soustrait maintenant p de $100r' + n''$, cela donne le reste final $r'' = 25$ et la racine carrée est $10s' + x$, i.e. 86.

Si on considère l'identité remarquable $(a+b)^2 - a^2 = (2a+b)b$, la quatrième étape de l'algorithme peut être vue comme un moyen de terminer de retrancher $(a+b)^2$ quand on a déjà retranché a^2.

Cette méthode élémentaire d'extraction de la racine carrée est un cas particulier de l'algorithme que nous allons présenter dans ce chapitre (cas particulier où $L = \beta = 10$).

3.2 L'algorithme implanté dans GMP

L'algorithme dont nous prouvons l'implantation mpn_dq_sqrtrem dans la suite de ce document utilise le paradigme proche *diviser pour régner*. On va découper l'entrée en deux parties de même longueur (approximativement), puis calculer récursivement la racine carrée sur les mots de poids fort. A cette étape, il ne restera plus qu'à effectuer une correction du résultat obtenu en tenant compte de la partie de poids faible de l'entrée.

On peut voir cet algorithme comme une extension de la méthode élémentaire présentée plus haut en considérant des blocs de taille $2n$ au lieu de blocs de taille 2. On peut aussi l'interpréter comme une variante discrète de la méthode de Newton. Classiquement, l'itération de Newton pour calculer la racine carrée de a est :

$$x_{k+1} = x_k + \frac{a - x_k^2}{2x_k} = \frac{1}{2}(x_k + \frac{a}{x_k}).$$

Au lieu de calculer avec des nombres flottants, ce qui nécessite de recalculer la division de a par x_k à chaque étape, la variante discrète met à jour la valeur du reste $a - x_k^2$. Cet algorithme peut être utilisé avec n'importe quels algorithmes efficaces de multiplication et de division, avec une complexité de l'ordre de $cM(n)$, où $M(n)$ représente la complexité de la multiplication de deux nombres de n bits et où c est une petite constante [Zur94].

Algorithm `SqrtRem`.
Input : $N > 0$ such as $N > \frac{L^2}{4}$.
Output : (S, R) such that $N = S^2 + R$ with $S^2 \leq N < (S+1)^2$.
0. **if** N is small **then** return SqrtRemNaive(N)
1. write N as $N'L^2 + N_1 L + N_0$ where $N_1, N_0 < L$ and $\frac{L}{4} < N'$
2. $(S', R') \leftarrow$ `SqrtRem`(N')
3. $(Q, R'') \leftarrow$ `DivRem`$(R'L + N_1, 2S')$
4. $(S, R) \leftarrow (S'L + Q, R''L + N_0 - Q^2)$
5. **if** $R < 0$ **then** $(S, R) \leftarrow (S - 1, R + 2S - 1)$
6. return (S, R).

FIG. 3.1 – Pseudo-code de l'algorithme d'extraction de la racine carrée

L'algorithme [Zim99] présenté dans la figure 3.1 prend en entrée un nombre N, le décompose en trois parties N', N_1 et N_0 (étape 1). Ensuite il calcule récursivement la racine carrée S' et le reste associé R' de la partie plus significative N' (étape 2). Il recompose R' avec N_1 et divise le nombre obtenu par $2S'$ (étape 3). Dans cette ligne, `DivRem` représente l'opération de division avec reste. Ce calcul donne une approximation de la racine carrée. De ce calcul, on tire le reste (étape 4). Le signe du reste permet de déterminer si l'approximation est exacte ou trop grande d'une unité. Le fait que la surestimation de la racine carrée soit au plus de 1 est une des propriétés fondamentales de l'algorithme. Si c'est nécessaire, l'algorithme corrige la valeur de la racine carrée et du reste associé (étape 5).

Dans la suite, nous simulons le fonctionnement de l'algorithme sur l'entrée 2703.

2703	52→ 51
203	20/10 = 2
-1	
102	

1. On découpe l'entrée n en trois parties $n' = 27$, $n_1 = 0$, et $n_0 = 03$.
2. On calcule récursivement la racine carrée et son reste pour $n' = 27$; cela donne $s' = 5$ et $r' = 2$. On écrit r'=2 en dessous de $n' = 27$.
3. On abaisse n_1, *i.e.* 0, et on divise le nombre obtenu, 20, par le double de s' ($2s' = 10$). Cela donne $q = 2$ et $r'' = 0$. Ainsi l'estimation de la racine carrée est $10s' + q = 52$.
4. On calcule le reste associé à l'estimation de la racine carrée. Pour cela, on soustrait q^2 à $10r'' + n_0$; ce qui donne $3 - 2^2 = -1$. L'égalité $2703 = 52^2 + (-1)$ est vérifiée, mais le reste est négatif. Cela signifie que la racine carrée a été surestimée de 1.
5. Finalement, on soustrait 1 à l'estimation de la racine carrée. On obtient 51 et le reste est augmenté de $2 \cdot 52 - 1$ et vaut donc 102.

Dans la suite de ce chapitre, nous allons décrire l'implantation de ce programme telle qu'elle est faite dans la bibliothèque GMP.

3.3 Programme de calcul de la racine carrée de GMP

Dans les paragraphes précédents, nous avons uniquement considéré le calcul de la racine carrée d'un point de vue algorithmique. Dans cette section, nous nous intéressons à l'implantation effective de cet algorithme dans GMP.

Le programme de calcul de la racine carrée se décompose en deux fonctions : la fonction de base mpn_dq_sqrtrem et la fonction d'encapsulation mpn_sqrtrem. La première effectue le calcul récursif de la racine carrée telle que nous l'avons présenté dans les deux sections précédentes. Cette fonction ne calcule la racine carrée et le reste que pour une certaine classe d'entrées. La fonction d'encapsulation mpn_sqrtrem a pour unique rôle de transformer l'entrée n dont on cherche à calculer la racine carrée en une entrée acceptable n' pour mpn_dq_sqrtrem. Elle calcule ensuite, à partir de la racine carrée s' et le reste r' retournés par la fonction mpn_dq_sqrtrem, la racine carrée r et le reste s de l'entrée initiale n.

3.3.1 Fonction d'encapsulation

Comme nous l'avons vu dans la section précédente, l'algorithme calcule une approximation à 1 près (par excès) de la racine carrée. Au niveau du programme implanté dans la bibliothèque GMP, pour que l'erreur sur l'approximation ne dépasse pas 1, il faut s'assurer que l'entrée vérifie bien une certaine condition de normalisation.

Soit n le nombre de mots-mémoire composant l'entrée N. La condition de normalisation exige que n soit pair et que $\frac{\beta^n}{4} \leq N < \beta^n$. Si ce n'est pas le cas, la fonction d'encapsulation va construire une nouvelle entrée N' vérifiant la condition de normalisation et qui pourra être traitée par le programme de calcul avec pré-condition. En base 2, la condition de normalisation exprime que la racine carrée calculée sera pleine, c'est-à-dire que son bit de poids fort sera positionné à 1. Pour

que cette propriété soit vérifiée, il suffit que au moins l'un des deux bits de poids fort de l'entrée ne soit pas nul. C'est ce qu'exprime la condition $\frac{\beta^n}{4} \leq N < \beta^n$.

La fonction d'encapsulation mpn_sqrtrem reçoit en entrée un nombre N ainsi que le nombre n de mots-mémoire nécessaire pour le représenter. On appelle c la moitié des zéros situé au début du mot de poids fort et tn la valeur de $\lceil \frac{n}{2} \rceil$. Si N ne vérifie pas la condition de normalisation, c'est-à-dire si n est impair ou si c est strictement positif, on construit une entrée $N_1 = 2^{2c}\beta^{2tn-n}N$. Ce nombre est normalisé, on peut donc utiliser la fonction mpn_dq_sqrtrem pour calculer la racine carrée de N_1. On obtient S_1 et le reste associé R_1. Il ne reste plus qu'à dénormaliser S_1 et R_1 pour en déduire la racine carrée de N et le reste associé. Nous verrons les détails de la preuve de correction de ce calcul dans le chapitre suivant.

Le code de la fonction d'encapsulation est donné à la figure 3.2. On peut remarquer que les pré-conditions de la fonction sont données sous forme d'assertions dans le code C. Par exemple, la condition

```
ASSERT (! MPN_OVERLAP_P (sp, (nn+1)/2, np, nn));
```

permet de s'assurer qu'il n'y a pas de recouvrement entre $\{sp, (nn+1)/2\}$ et $\{np, nn\}$.

3.3.2 Programme de calcul de la racine carrée avec pré-condition

Lorsque l'entrée N est représentable par $2n$ mots-mémoire exactement et que N vérifie la condition de normalisation $\frac{\beta^{2n}}{4} \leq N < \beta^{2n}$, la fonction mpn_dq_sqrtrem permet de calculer la racine carrée de N ainsi que le reste qui lui est associé.

La principale caractéristique de cette implantation de mpn_dq_sqrtrem est qu'elle fonctionne *en place*. Le stockage de l'entrée nécessite $2n$ blocs de mémoire, la racine carrée et le reste nécessitent à eux deux $2n + O(1)$, cependant l'algorithme assure que seulement $3n$ blocs sont nécessaires pour stocker tous les calculs intermédiaires. L'entrée est partiellement détruite pour stocker les calculs intermédiaires et le reste est stocké dans la partie de poids faible de l'entrée. L'emplacement précis où est stockée le reste est fondamental pour s'assurer de la correction du programme.

La figure 3.3 donne le code C de la procédure telle qu'on peut la trouver dans la version 4.0.1 de la bibliothèque GMP [Gra02]. Le lecteur attentif aura remarqué les similitudes avec l'algorithme présenté dans la section précédente. On peut cependant noter d'importantes différences. Tout d'abord, le nombre que nous appelons h dans les paragraphes précédents s'appelle maintenant n. De plus, il est inutile de calculer des nombres $H = \beta^h$ ou $L = \beta^l$ ainsi que les valeurs N', N'', N_1, et N_0, puisque celles-ci se trouvent stockées à un endroit déterminé dans la mémoire. Si N est un nombre codé sur $2n$ blocs et dont le bloc de poids faible se trouve à la position n_p, alors N' est un nombre de longueur $2h$ dont le bloc de poids faible est à la position $n_p + 2l$, la longueur de N_1 est l et ce nombre est stocké à la position $n_p + l$. Finalement N_0 se trouve à la position n_p, sans avoir aucun calcul supplémentaire à faire. De même, quand on calcule la racine carrée de N' de la manière que le reste R' est stocké à la position $n_p + 2l$ (avec la longueur h) et que N_1 se trouve à la position $n_p + l$ (avec la longueur l), alors sans aucun calcul, on obtient $R'L + N_1$ à la position $n_p + l$, avec la longueur $h + l = n$. Cette astuce est utilisé à la ligne 14 pour construire $R'L + N_1$ et à la ligne 21 pour $R''L + N_0$.

Deuxièmement, la plupart des opérations binaires sur les grands entiers ont en fait 4 arguments : deux qui indiquent la position du bloc de poids faible des entrées (ces arguments sont des pointeurs), un argument qui indique où se trouvera le bloc de poids faible du résultat du calcul ; enfin le dernier argument donne la longueur (commune) de ces trois nombres. Par exemple, la fonction mpn_sub_n est la fonction utilisée pour faire la soustraction d'un nombre de n blocs avec un nombre de n blocs

```
mp_size_t mpn_sqrtrem (mp_ptr sp, mp_ptr rp, mp_srcptr np, mp_size_t nn)  {
  mp_limb_t *tp, s0[1], cc, high, rl;
  int c;
  mp_size_t rn, tn;
  TMP_DECL (marker);
  ASSERT (nn >= 0);
  if (nn == 0)   /* If OP is zero, both results are zero.  */
    return 0;
  ASSERT (np[nn-1] != 0);
  ASSERT (rp == NULL || MPN_SAME_OR_SEPARATE_P (np, rp, nn));
  ASSERT (rp == NULL || ! MPN_OVERLAP_P (sp, (nn+1)/2, rp, nn));
  ASSERT (! MPN_OVERLAP_P (sp, (nn+1)/2, np, nn));

  high = np[nn-1];
  if (nn == 1 && (high & MP_LIMB_T_HIGHBIT))
    return mpn_sqrtrem1 (sp, rp, np);
  count_leading_zeros(c, high);
  c = c / 2; /* we have to shift left by 2c bits to normalize {np, nn} */
  tn = (nn+1) / 2; /* 2*tn is the smallest even integer >= nn */

  TMP_MARK (marker);
  if ((nn % 2) || (c > 0))  {
      tp = TMP_ALLOC_LIMBS (2 * tn);
      tp[0] = 0; /* needed only when 2*tn > nn, but saves a test */
      if (c) mpn_lshift(tp + 2*tn - nn, np, nn, 2 * c);
      else MPN_COPY (tp + 2*tn - nn, np, nn);
      rl = mpn_dq_sqrtrem (sp, tp, tn);
      /* we have 2^(2k)*N = S^2 + R where k = c + (2tn-nn)*BITS_PER_MP_LIMB/2,
         thus 2^(2k)*N = (S-s0)^2 + 2*S*s0 - s0^2 + R where s0=S mod 2^k */
      c += (nn % 2) * BITS_PER_MP_LIMB / 2; /* c now represents k */
      s0[0] = sp[0] & (((mp_limb_t) 1 << c) - 1); /* S mod 2^k */
      rl += mpn_addmul_1 (tp, sp, tn, 2 * s0[0]); /* R = R + 2*s0*S */
      cc = mpn_submul_1 (tp, s0, 1, s0[0]);
      rl -= (tn > 1) ? mpn_sub_1(tp + 1, tp + 1, tn - 1, cc) : cc;
      mpn_rshift (sp, sp, tn, c);
      tp[tn] = rl;
      if (rp == NULL) rp = tp;
      c = c << 1;
      if (c < BITS_PER_MP_LIMB) tn++; else { tp++; c -= BITS_PER_MP_LIMB; }
      if (c) mpn_rshift (rp, tp, tn, c); else MPN_COPY_INCR (rp, tp, tn);
      rn = tn;
    }
  else  {
      if (rp == NULL)
        rp = TMP_ALLOC_LIMBS (nn);
      if (rp != np) MPN_COPY (rp, np, nn);
      rn = tn + (rp[tn] = mpn_dq_sqrtrem (sp, rp, tn));
    }
  MPN_NORMALIZE (rp, rn);
  TMP_FREE (marker);
  return rn;
}
```

FIG. 3.2 – Le code C de la fonction d'encapsulation mpn_sqrtrem

```
1 int mpn_dq_sqrtrem(mp_ptr sp, mp_ptr np, mp_size_t n){
2   mp_limb_t q; /* carry out of {sp, n} */
3   int c, b; /* carry out of remainder */
4   mp_size_t l, h;
5
6   ASSERT (np[2*n-1] >= MP_LIMB_T_HIGHBIT/2);
7
8   if (n == 1) return mpn_sqrtrem2(sp, np, np);
9
10  l = n / 2;
11  h = n - l;
12  q = mpn_dq_sqrtrem (sp + l, np + 2 * l, h);
13  if (q) mpn_sub_n (np + 2 * l, np + 2 * l, sp + l, h);
14  q += mpn_divrem (sp, 0, np + l, n, sp + l, h);
15  c = sp[0] & 1;
16  mpn_rshift (sp, sp, l, 1);
17  sp[l-1] |= q << (BITS_PER_MP_LIMB - 1);
18  q >>= 1;
19  if (c) c = mpn_add_n (np + l, np + l, sp + l, h);
20  mpn_sqr_n (np + n, sp, l);
21  b = q + mpn_sub_n (np, np, np + n, 2 * l);
22  c -= (l == h) ? b : mpn_sub_1 (np+2*l,np+2*l,1,b);
23  q = mpn_add_1 (sp + l, sp + l, h, q);
24
25  if (c < 0) {
26      c += mpn_addmul_1 (np, sp, n, 2) + 2 * q;
27      c -= mpn_sub_1 (np, np, n, 1);
28      q -= mpn_sub_1 (sp, sp, n, 1);
29    }
30  return c;
31}
```

FIG. 3.3 – Le code de la fonction `mpn_dq_sqrtrem` distribuée dans GMP 4.0.1

et le résultat est un nombre de n blocs. Les opérations comme l'addition ou la soustraction peuvent provoquer des débordements, cela correspond à la mise à 1 du bit retourné par l'opération.

La fonction de calcul de la racine carrée peut elle aussi conduire à un débordement. Si l'on calcule la racine carrée d'un nombre (dont le représentation tient dans $2n$ blocs, la racine carrée tiendra dans n blocs. Cependant, on sait seulement que le reste de la racine carrée est inférieur ou égal à deux fois la racine carrée, et pour cette raison il se peut qu'il ne tienne pas dans n blocs. Le bit de débordement est la valeur retournée par la fonction.

Troisièmement, il n'y a pas de division de $R'L+N_1$ par $2S'$ comme c'est spécifié dans l'algorithme 3.2. Une telle division nécessiterait de construire le nombre $2S'$ et donc de trouver un emplacement dans la mémoire où le stocker. Pour éviter cela, la division par $2S'$ se fait en deux étapes : la première consiste à diviser $R'L + N_1$ par S' ; la seconde étape consiste à diviser le quotient obtenu par 2 (ligne 16), en prenant soin de mettre à jour le reste si le quotient est impair (ligne 19). Ce

procédé se situe entre les lignes 13 à 19 du programme 3.3. Les calculs effectifs seront décrits plus en détail à la section 4.5.

3.3.2.1 Utilisation de la mémoire par mpn_dq_sqrtrem

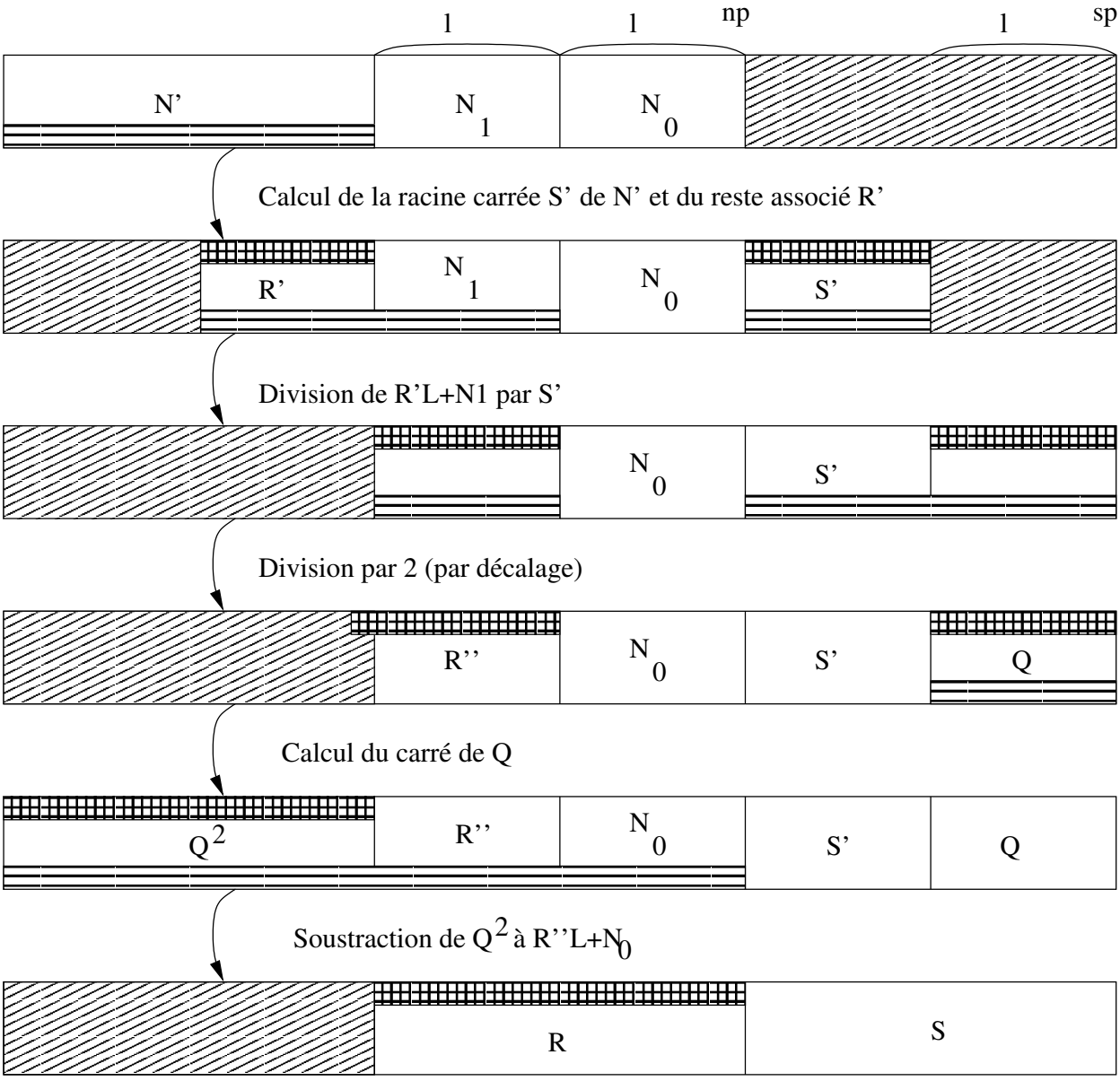

FIG. 3.4 – Utilisation de la mémoire au cours de l'exécution de l'algorithme : Les bits de poids fort se trouvent à gauche. Les hachures obliques représentent des données inutiles. Les hachures horizontales caractérisent les entrées de l'opération à venir. Les zones quadrillées représentent les sorties de l'opération précédente. Les bits de poids sont à gauche.

Initialement, l'entrée se trouve à l'adresse $\{np, 2n\}$. La première étape consiste à calculer la racine carrée de la partie de poids fort de l'entrée. Cela donne la partie de poids fort de la racine carrée et un reste que l'on va stocker dans la partie de poids faible de l'entrée ...

3.4 Preuve initiale de l'algorithme

Dans [Zim00], P. Zimmermann présente une preuve sur papier de la correction de l'implantation de cet algorithme de calcul de la racine carrée. Cette preuve détaillée consiste en la fourniture du code C de la fonction mpn_dq_sqrtrem et de la fonction d'encapsulation mpn_sqrtrem. La démonstration de la correction de la fonction mpn_dq_sqrtrem tient sur moins de deux pages [Zim00, pp. 8–9]. C'est cette démonstration formelle au sens mathématique du terme qui nous sert de base pour réaliser la preuve de correction de mpn_dq_sqrtrem aussi bien au niveau des calculs arithmétiques que de l'implantation en C.
Une première preuve (au niveau abstrait) a été développée par P. Zimmermann en collaboration avec des membres du projet Lemme. La manière de spécifier les propriétés de l'algorithme de calcul de la racine carrée était différente de celle finalement adoptée. En particulier, elle faisant apparaître dans la spécification du calcul la constante H.

$$(\mathsf{SqrtremProp}\ N\ H\ S\ R) \equiv \begin{cases} N = S^2 + R \\ 0 \leq R \\ R \leq 2 \times S \\ H \leq 2 \times S \\ S < H \end{cases}$$

où N est l'entrée fournie à l'algorithme, S la racine carrée entière, R le reste associée et H la constante utilisée pour la décomposition de l'entrée. Dans le chapitre suivant, nous reprenons cette preuve abstraite avant de présenter la preuve de correction de l'implantation elle-même. Cette preuve a été modifiée pour avoir une spécification de l'algorithme de calcul de la racine carrée qui soit indépendante de H.

3.5 Conclusion

Dans ce chapitre, nous avons présenté l'algorithme de calcul efficace de la racine carrée implanté dans GMP. L'objectif suivant est de montrer la correction de la fonction mpn_dq_sqrtrem de calcul de la racine carrée de GMP. C'est ce que nous allons étudier dans le chapitre suivant.

Chapitre 4

Preuve formelle de la racine carrée de GMP

Dans ce chapitre, nous présentons une preuve formelle de correction de la fonction mpn_dq_sqrtrem de GMP. Trois preuves correspondant à des niveaux d'abstraction différents sont présentées. Tout d'abord nous présentons la preuve de correction de l'algorithme en ne considérant que les calculs arithmétiques nécessaires pour extraire la racine carrée. Dans un deuxième temps, nous proposons une description fonctionnelle de l'algorithme. Finalement, nous prouvons la correction du code tel qu'il est fourni dans la bibliothèque GMP.

Au cours de ces trois étapes, nous sommes amenés à raffiner les structures de données considérées. Dans la première étape, l'algorithme manipule des entiers relatifs de type $\mathbb{Z}$ alors qu'au niveau de l'implantation, le programme ne manipule que des entiers bornés (encodés sous forme de tableaux de mots-mémoire).

Toutes ces preuves présentées ci-dessous ont été développées formellement dans l'assistant de preuves COQ. La structure du chapitre reprend l'ordre de présentation de la preuve de correction choisi dans l'article [BMZ02] écrit avec Y. Bertot et P. Zimmermann.

4.1 Description formelle et preuve de l'algorithme

La première étape de ce travail consiste à montrer la correction de l'algorithme à un niveau très abstrait. Nous commençons par décrire le fonctionnement de l'algorithme sur les entiers relatifs. A ce niveau d'abstraction, nous ne traitons pas les questions relatives à la gestion de la mémoire et aux problèmes de débordement arithmétique.

La spécification d'un algorithme de calcul de la racine carrée peut être décrite de la façon suivante : l'algorithme prend en entrée un entier N strictement positif et retourne deux entiers S et R. Ces entiers N, S, et R sont reliés entre eux par la propriété suivante :

$$\mathsf{SqrtremProp}(N, S, R) : \begin{cases} N = S^2 + R, \\ 0 < S, \\ 0 \leq R \leq 2S. \end{cases}$$

La propriété $R \leq 2S$ garantit que $N < (S+1)^2$, et donc que $S = \lfloor \sqrt{N} \rfloor$.

Comme nous l'avons vu dans le chapitre précédent, nous avons deux parties pour notre algorithme : un algorithme récursif avec une condition de normalisation sur ses entrées et un algorithme d'encapsulation. Dans un premier temps, nous ne considérons que l'algorithme de base

(c'est-à-dire l'algorithme récursif avec pré-condition dont l'implantation dans GMP est la fonction mpn_dq_sqrtrem).

4.1.1 Condition de normalisation

Soit β la base, c'est-à-dire le plus grand nombre qui peut être encodé dans un bloc (*limb* dans le jargon anglophone de GMP) plus un.

L'algorithme interne prend en entrée un nombre $N \in \mathbb{Z}$ tel que $N > 0$. On suppose que ce nombre est normalisé, c'est-à-dire que :

$$\frac{\beta^{2n}}{4} \leq N < \beta^{2n}.$$

Si β est une puissance de 2, cela signifie que les deux bits les plus significatifs de N ne sont pas simultanément nuls. Cette condition garantit que l'écriture de la racine carrée S de N nécessitera exactement n mots-mémoire (limbs) dans la base β.

L'algorithme commence par décomposer son entrée en deux parties, ses mots de poids forts et ses mots de poids faible. Cela correspond à une division par $\beta^{2\lfloor \frac{n}{2} \rfloor}$. Le diviseur effectif peut être vu de façon plus abstraite en considérant deux entiers H et L où L vaut $\beta^{\lfloor \frac{n}{2} \rfloor}$. Les entiers H et L doivent vérifier les conditions suivantes :

$$H \geq L \tag{4.1}$$

$$L > 0 \tag{4.2}$$

et

$$HL = \beta^n. \tag{4.3}$$

On verra plus tard que H doit nécessairement être pair. Les entiers H et L joueront un rôle très important dans la description formelle de l'algorithme. La condition de normalisation peut être reformulée en fonction de H et L de la manière suivante :

$$\frac{(HL)^2}{4} \leq N < (HL)^2.$$

Néanmoins, nous adoptons la formulation suivante qui évite d'utiliser l'opération de division sur les entiers :

$$N < (HL)^2 \leq 4N. \tag{4.4}$$

4.1.2 Étapes de calcul

Dans cette partie, nous décrivons, pas à pas, les calculs effectués par l'algorithme. Nous commençons par découper N en deux parties N' et N'', où N' est la partie la plus significative de N et N'' la partie la moins significative :

$$N = N'L^2 + N'', \qquad 0 \leq N'' < L^2.$$

La racine carrée de N' est calculée en appelant récursivement l'algorithme et retourne les valeurs S' et R' qui sont la racine carrée (resp. le reste) du calcul de racine carrée de N'. De façon formelle, on a la propriété $\mathsf{SqrtremProp}(N', S', R')$, c'est-à-dire :

$$(N' = S'^2 + R') \quad \wedge \quad (0 < S') \quad \wedge \quad (0 \leq R' \leq 2S').$$

Ensuite N'' est découpé en deux parties :

$$N'' = N_1 L + N_0, \qquad \text{avec } 0 \leq N_0, N_1 < L,$$

et $R'L + N_1$ est divisé par $2S'$. Cela donne un quotient Q et un reste R'' :

$$R'L + N_1 = Q(2S') + R'', \qquad \text{avec } 0 \leq R'' < (2S').$$

Toutes ces étapes de calcul conduisent au résultat suivant :

$$\begin{aligned} N &= N'L^2 + N_1 L + N_0 \\ &= (S'^2 + R')L^2 + N_1 L + N_0 \\ &= (S'L)^2 + (R'L + N_1)L + N_0 \\ &= (S'L)^2 + (2S'Q + R'')L + N_0 \\ &= (S'L + Q)^2 + (R''L + N_0 - Q^2). \end{aligned} \tag{4.5}$$

Ces équations font penser que $(S'L + Q)$ pourrait être la racine carrée entière de N. Nous allons montrer que ce nombre est en tout cas très proche de la valeur exacte de la racine carrée. Il s'agit soit de la racine carrée exacte, soit d'une surestimation d'une unité seulement. Les propriétés $L \leq 2S'$ et $Q \leq L$ sont fondamentales pour établir ce résultat. Intuitivement, cela signifie que la partie de poids fort de la racine carrée calculée au cours de l'appel récursif est suffisamment grande par rapport à la partie de poids faible de l'entrée. C'est assez logique sachant que l'algorithme `SqrtRem` est simplement une variante de la méthode de Newton, où la précision est doublée à chaque étape.

4.1.3 Une borne inférieure sur la racine carrée S'

La propriété de normalisation, l'équation de définition de N' et de N'' et la borne sur N'' permettent de déduire les deux inégalités suivantes :

$$H^2 L^2 \leq 4(N'L^2 + N'') < 4(N' + 1)L^2.$$

Supposons que H est pair. On peut considérer $H' = \frac{H}{2}$ et simplifier l'inégalité précédente :

$$H'^2 < N' + 1$$

elle-même équivalente à

$$H^2 \leq 4N'.$$

De plus, il est évident que N' est nécessairement plus petit que H^2 ; par conséquent les pré-conditions à l'appel récursif de l'algorithme sont bien vérifiées. Comme S' est la racine carrée de N' et que $L \leq H$, on en déduit l'inégalité suivante pour S' :

$$L \leq 2S'. \tag{4.6}$$

4.1.4 Une borne supérieure pour le quotient Q

Le nombre Q est le quotient d'une division ; on a donc $(2S')Q + R'' = R'L + N_1$. S' et R' proviennent du calcul récursif de la racine carrée, donc $R' \leq 2S'$. Comme $N_1 < L \leq 2S'$, on obtient

$$2S'Q + R'' < 2S'(L + 1).$$

Sachant que R'' est positif, on en déduit que

$$Q \leq L. \tag{4.7}$$

4.1.5 Pas de sous-estimation de la racine carrée

Nous montrons que $S'L + Q$ est plus grand ou égal à la racine carrée entière recherchée. Pour cela, il suffit de montrer que $S'L+Q+1$ est une surestimation, c'est-à-dire que $N < (S'L+Q+1)^2$. Cela revient à démontrer que $N - (S'L+Q)^2 \leq 2(S'L+Q)$; en utilisant l'équation (4.5), on arrive à l'inégalité :

$$R''L + N_0 - Q^2 \leq 2(S'L + Q).$$

On sait que $R'' \leq 2S' - 1$ et que $N_0 < L$, puisque R'' et N_0 sont les restes de division par $2S'$ et L. De là, on déduit que :

$$R''L + N_0 \leq (2S' - 1)L + L = 2S'L$$

En utilisant le fait que Q est positif, on établit la série d'inégalités suivante :

$$R''L + N_0 - Q^2 \leq R''L + N_0 \leq 2S'L \leq 2(S'L + Q).$$

4.1.6 Une borne sur la surestimation

Le candidat $S'L + Q$ peut toujours être une approximation par excès. Si c'est le cas, alors $S'L + Q - 1$ est un autre candidat pour la racine carrée et le reste associé est

$$R''L + N_0 - Q^2 + 2(S'L + Q) - 1. \tag{4.8}$$

Il suffit de montrer que cette expression est toujours positive ou nulle. Tout d'abord la borne inférieure (4.6) sur $2S'$ et la borne supérieure (4.7) sur L permet d'établir les inégalités suivantes :

$$Q - 1 \leq L \leq 2S'.$$

Nous savons donc que $Q-1 \leq 2S'$ et $Q-1 \leq L$. Si $Q \geq 1$, on peut en déduire l'inégalité suivante :

$$(Q - 1)^2 \leq 2S'L,$$

qui est aussi vraie pour $Q = 0$ puisque S' et L sont strictement positifs. L'inégalité précédente peut donc se réécrire en :

$$0 \leq 2(S'L + Q) - Q^2 - 1.$$

Comme N_0, L, et R'' sont positifs ou nuls, on peut ajouter $R''L+N_0$ au membre droit de l'inégalité, ce qui prouve que le reste corrigé est bien positif ou nul.

Les résultats des deux paragraphes précédents montrent que la racine carrée est soit $S'L + Q$, soit $S'L+Q-1$. Pour déterminer la bonne valeur, il suffit d'étudier le signe du reste $R''L+N_0-Q^2$. Si le reste est positif ou nul, on sait déjà que $R''L + N_0 - Q^2 \leq 2(S'L + Q)$; les deux conditions sur le reste sont donc bien vérifiées. Si le reste est négatif, on lui ajoute alors $2(S'L + Q) - 1$, ce qui donne bien une valeur positive ou nulle. Il ne reste plus qu'à vérifier que cette nouvelle valeur est bien inférieure ou égale à $2(S'L + Q - 1)$. C'est immédiat sachant que cette valeur est obtenue en additionnant une valeur strictement négative à $2(S'L + Q) - 1$.

4.1.7 Liens entre H, L, et la base β

La condition de normalisation impose que

$$\frac{\beta^{2n}}{4} \leq N < \beta^{2n}.$$

Les nombres H et L sont simplement utilisés pour abstraire la base durant la preuve abstraite de l'algorithme. Cependant, nous devons les relier à la base β et comprendre comment leurs propriétés peuvent être interprétées en termes de propriétés de la base.

Tout d'abord, H et L doivent vérifier les propriétés $HL = \beta^n$ et $L \leq H$. $L = \beta^l$ et $H = \beta^{n-l}$ avec $l \leq \frac{n}{2}$ sont des valeurs convenables, par exemple

$$l = \lfloor \frac{n}{2} \rfloor, \qquad L = \beta^l, \qquad H = \beta^{n-l}.$$

Pour les appels récursifs, cela assure que la longueur (nombre de mots-mémoire) du nombre décroît à chaque appel, excepté pour $n = 1$. Concrètement, cela signifie que l'algorithme nécessite un autre algorithme pour calculer la racine carrée des nombres N tels que $\frac{\beta^2}{4} \leq N < \beta^2$. Une telle fonction (mpn_sqrtrem2) est fournie par la bibliothèque GMP. Nous n'avons pas certifié cette fonction, mais seulement supposé qu'elle vérifie la spécification SqrtremProp.

La dernière contrainte que nous avons rencontré dans la preuve abstraite est que H doit être pair. Comme $HL = \beta^n$, cela impose que β soit pair. En pratique, cela signifie que l'on peut adapter facilement l'algorithme pour n'importe quelle base paire, en particulier la base 10.

C'est l'hypothèse que H est pair qui permet de montrer que $L \leq 2S'$; puis d'en déduire que $Q \leq L$. Ces deux inégalités permettent de montrer que le reste corrigé (4.8) est toujours positif ou nul, et donc qu'une seule correction au plus est nécessaire.

Notre algorithme ne fonctionne pas lorsque H est impair. En effet, supposons H impair. Les bornes établies précédemment (4.6) et (4.7) ne sont pas vraies lorsque $L = H = 2S'+1$ et $Q = L+1$, avec l'entrée $\frac{1}{4}L^4 + \frac{1}{2}L^3 + \frac{1}{4}L^2 - L$. Considérons par exemple $H = L = 3$, et $N = 33$. La première approximation de la racine carrée et du reste est $(7, -16)$, l'approximation corrigée vaut $(6, -3)$, par conséquent il faut au moins deux corrections dans ce cas pour obtenir la valeur correcte de la racine carrée.

4.2 Preuve abstraite de l'encapsulateur

La fonction de calcul de la racine carrée mpn_sqrtrem n'a pas de pré-condition demandant que l'entrée soit normalisée ou que sa longueur soit paire. En pratique, tous les nombres sont fournis avec leur longueur, la fonction mpn_sqrtrem prend donc en entrée un entier N et un nombre n tel que $0 < N < \beta^n$. n représente le nombre de mots-mémoire nécessaire pour écrire N en mémoire.

Si N vérifie la condition de normalisation de mpn_dq_sqrtrem, c'est-à-dire si n est pair et $\frac{\beta^n}{4} \leq N < \beta^n$, la racine carrée peut être calculée par l'algorithme de base. Sinon, une procédure d'encapsulation se charge de transformer N en une entrée correcte pour l'algorithme de base, d'effectuer le calcul de racine carrée avec l'algorithme de base, puis d'interpréter le résultat pour retourner la racine carrée de N et le reste associé.

Dans ce paragraphe, nous décrivons la preuve formelle (au niveau mathématique) de l'algorithme.

Soit c la moitié du nombre de zéros situés au début du bloc de poids fort de N. On définit tn par $tn = \lceil \frac{n}{2} \rceil$. On distingue deux cas, soit $even(n) \wedge c = 0$, soit $odd(n) \vee c > 0$. Dans le premier

cas, si c vaut 0, cela signifie que N est normalisé. En effet, cela signifie que N a au plus un zéro au début. Par conséquent, un des deux bits de poids fort vaut 1. Dans le cas où $odd(n) \vee c > 0$, c'est un peu plus technique. On commence par construire une nouvelle entrée N_1. N_1 sera normalisé par rapport à $2tn$:

$$N_1 = 2^{2c}\beta^{2tn-n}N$$

On définit k comme $c + (2tn - n)b/2$ où b est le nombre de bits d'un mot-mémoire (limb en langage GMP). $(2tn - nn)$ vaut 0 ou 1 en fonction de la parité de n. k doit être une valeur entière, par conséquent b doit être pair. Cette condition n'apparaît pas dans la preuve informelle donnée par P. Zimmermann [Zim00, pp. 9-10]. Au départ, nous nous sommes trouvés bloqués dans la preuve en COQ. En effet, cette propriété était fondamentale pour pouvoir conclure.

Nous avons donc $N_1 = 2^{2k}N$. Comme N_1 est normalisé, on peut calculer deux nombres S_1 et R_1 vérifiant les propriétés suivantes :

$$N_1 = S_1^2 + R_1 \qquad R_1 \geq 0 \qquad R_1 \leq 2S_1 \tag{4.9}$$

L'équation liant N_1, S_1, et R_1 peut se transformer en

$$2^{2k}N = (S_1 - s_0)^2 + 2S_1s_0 - s_0^2 + R_1$$

où $s_0 = S \bmod 2^k$. Il nous faut maintenant montrer que $S_1 - s_0$ (resp. $2S_1s_0 - s_0^2 + R_1$) peut être divisé par 2^k (resp. 2^{2k}). Cela signifie que nous devons calculer deux entiers S et R tels que

$$S_1 - s_0 = 2^kS$$

et que

$$2S_1s_0 - s_0^2 + R_1 = 2^{2k}R$$

Finalement, nous montrons simplement que N, S, et R vérifient bien la propriété SqrtremProp, à savoir :

$$N = S^2 + R \qquad 0 < S \qquad 0 \leq R \leq 2S$$

Nous décrivons ici comment montrer que $R \leq 2S$. Comme $R_1 \leq 2S_1$, on a $S_1^2 \leq N_1 < (S_1 + 1)^2$. Cette inégalité peut se réécrire de la manière suivante :

$$(2^kS + s_0)^2 \leq 2^{2k} \times N < (2^kS + s_0 + 1)^2$$

Comme $s_0 \geq 0$, on en déduit que $(2^kS)^2 \leq (2^kS + s_0)^2$. De même, on déduit de $s_0 < 2^k$ que $2^{2k}N \leq (2^k(S + 1))^2$. A ce stade, nous avons montré que

$$(2^kS)^2 \leq 2^{2k} \times N < (2^k(S + 1))^2$$

En divisant par 2^{2k}, on obtient le résultat attendu, à savoir $S^2 \leq N < (S + 1)^2$. De là, il est facile de montrer $R \leq 2S$ en développant $(S + 1)^2$ et en remplaçant N par $S^2 + R$.

Jusqu'ici, nous avons simplement établi des relations formelles entre des variables. Nous n'avons pas décrit d'algorithme effectif de calcul de la racine carrée. Par exemple, nous avons seulement relié N et N_1 par une égalité, mais nous n'avons pas donné de moyens de calculer N_1 à partir de N. Dans la prochaine section, nous allons construire en COQ un programme fonctionnel récursif de calcul effectif de la racine carrée. La description abstraite faite dans cette section servira à prouver la correction de ce programme.

4.3 Description fonctionnelle de l'algorithme

Dans cette section, nous donnons une description fonctionnelle de l'algorithme dans Coq. Cela consiste à fournir un programme fonctionnel calculant la racine carrée dont le type est exactement sa spécification.

La description abstraite de l'algorithme de base de calcul de racine carrée présentée dans la section précédente met en relation les résultats intermédiaires du calcul de la racine carrée, mais ne donne pas de moyen effectif de calculer à partir d'une entrée. Il ne donne aucune garantie sur l'adéquation des entrées avec les pré-conditions des différentes fonctions et de leurs appels récursifs.

Pour que la description de l'algorithme soit complète, il faut que l'on dispose d'un programme exécutable dans notre formalisme logique et une preuve que ce programme calcule bien la racine carrée de son entrée.

Une telle description fonctionnelle de l'algorithme va faire une utilisation intensive des *types dépendants*, ceci afin de garantir la correction et la terminaison de l'algorithme défini.

Tout d'abord, les types dépendants seront utiles pour restreindre le domaine des fonctions. Pour cela, on donne un argument supplémentaire aux fonctions, ces arguments sont des certificats attestant que les données sont des entrées acceptables pour l'algorithme. Par exemple, la fonction decompose reçoit en arguments un valeur de type $\mathbb{N}$, celle-ci devant être supérieure ou égale à 1. Dans Coq, cette pré-condition s'exprime en donnant à la fonction decompose un type dépendant de la forme suivante :

$$\forall n : \mathbb{N}.(1 < n) \to \ldots$$

Ce type exprime que la fonction reçoit deux arguments, une valeur n de type $\mathbb{N}$ et une preuve que n est strictement supérieur à 1. La proposition $1 < n$ est en fait le type de la pré-condition. Ce type dépend de n, c'est la raison pour laquelle on parle de types dépendants.

Les fonctions peuvent aussi produire des résultats et des propriétés sur ces résultats. Par exemple, la fonction decompose retourne une paire d'entiers naturels, h et l, tels que :

$$h + l = n \quad \wedge \quad h < n \quad \wedge \quad 0 < l \leq h.$$

Ces propriétés ne sont vérifiées que quand $1 < n$. C'est pourquoi la fonction decompose doit avoir un certificat attestant que $1 < n$.

Finalement, le type de la fonction decompose est de la forme suivante :

$$\forall n : \mathbb{N}.(1 < n) \to \{h, l : \mathbb{N} \mid l + h = n \wedge h < n \wedge 0 < l \wedge l \leq h\}.$$

Dans la description abstraite de l'algorithme, nous utilisions deux constantes H et L. Cette description n'imposait que des faibles contraintes sur ces valeurs, et leur relation avec la base effective β utilisée par le programme de Gmp n'était pas utilisée dans la preuve abstraite de l'algorithme. Nous allons montrer que H et L sont des puissances de la base β. La fonction decompose servira à construire les entiers h et l tels que $H = \beta^h$ et $L = \beta^l$. Par conséquent, la description fonctionnelle de l'algorithme n'acceptera que des nombres normalisés par rapport à une puissance paire de la base β. Nous définissons la propriété isNormalized, introduite dans le paragraphe 4.4, de la façon suivante :

$$(\mathsf{IsNormalized}\ n\ v) \equiv n < v \leq 4n.$$

Pour exprimer que le nombre n doit être normalisé par rapport à une puissance paire de la base, nous affirmons simplement qu'il existe un h tel que la propriété $(\mathsf{IsNormalized}\ n\ \beta^{2h})$ soit vérifiée. Plus précisément, h est la moitié du nombre de chiffres de l'entrée et donc le nombre prévisible de chiffres de la sortie.

Le nombre de chiffres est aussi utilisé par l'algorithme, la fonction représentant l'algorithme de base a donc le type suivant :

$\forall h : \mathbb{N}.\ \forall n : \mathbb{Z}.\ (\mathsf{IsNormalized}\ n\ \beta^{2h}) \rightarrow \{s, r : \mathbb{Z} \mid (\mathsf{SqrtremProp}\ n\ s\ r)\}$.

C'est le type d'une fonction à trois arguments : elle prend en entrée un entier naturel h, un entier n, et aussi une preuve que les valeurs h et n sont cohérentes. Cette fonction retourne trois éléments : deux entiers s et r ainsi qu'une preuve que ces deux entiers vérifient bien la spécification de la racine carrée donnée dans la section précédente (4.2).

Pour simplifier les équations présentées dans la suite de ce document, nous donnons un nom à ce type complexe :

$$\begin{aligned}(\mathsf{sqrt\_F\_type}\ h) \quad \equiv \quad & \forall n : \mathbb{Z}.\quad (\mathsf{IsNormalized}\ n\ \beta^{2h}) \rightarrow \\ & \{s, r : \mathbb{Z} \mid (\mathsf{SqrtremProp}\ n\ s\ r)\}.\end{aligned}$$

Il est important de noter que l'on va construire simultanément la fonction et la preuve qu'elle vérifie bien sa spécification. Le type de la fonction sera suffisamment riche pour contenir la spécification d'un calcul de racine carrée. Les données classiques (comme des entiers...) ainsi que les arguments de preuve sont transmis d'une fonction à l'autre à l'aide des constructions de filtrage du langage de spécification de Coq. Par exemple, l'appel à la fonction decompose se fait dans un fragment de code de la forme :

```
Cases (Zle_lt_dec h 1) of
  (left H_h_le_1) ⇒ ...
| (right H_1_lt_h) ⇒
  Cases (decompose h ?) of
    (h', l, Heqh, H_h'_lt_h, H_O_lt_l, H_l_le_h) ⇒ ...
```

Ce fragment contient un appel à la fonction decompose sur la valeur h ; son deuxième argument, qui doit être une preuve que h est supérieur à 1, est remplacé par un simple point d'interrogation. A partir de ce fragment de code, le système Coq produira une obligation de preuve demandant de prouver que h est supérieur à 1 dans le contexte adéquat. Pour cet exemple, la preuve est triviale. Il suffit d'utiliser l'hypothèse $H\_1\_lt\_h$ qui affirme que $1 < h$. Cette hypothèse provient de l'analyse par cas sur (le_lt_dec h 1) qui non seulement décide si $h \leq 1$ ou $1 < h$, mais retourne une preuve de l'inégalité valide dans chacun des cas possibles et transmet dans les branches du filtrage l'hypothèse correspondante. Dans le cas qui nous intéresse, l'hypothèse $H\_1\_lt\_h$ permet de conclure immédiatement. Cependant, dans d'autres exemples, des preuves plus complexes peuvent être nécessaires.

On peut voir les résultats retournés par decompose comme deux entiers naturels h et l et quatre preuves que $h' + l = h$ (nous l'appellerons $Heqh$), $h' < h$ ($H\_h'\_lt\_h$), $0 < l$ ($H\_O\_lt\_l$), et $l \leq h'$ ($H\_l\_le\_h$).

4.3.1 Description d'une fonction récursive par ordre bien-fondé

L'utilisation des types dépendants est aussi nécessaire pour décrire des fonctions récursives non structurelles. Des arguments de preuve doivent être fournis à chaque itération de la fonction pour s'assurer qu'elle termine bien. En pratique, une fonction récursive dont le type est de la forme $\forall x : A.(B\ x)$ peut être décrite dans le système Coq par une fonction de type

$$\forall x : A.(\forall y : A.(R\ y\ x) \rightarrow (B\ y)) \rightarrow (B\ x),$$

```
λh, sqrt, n, Hnorm.
Cases (le_lt_dec h 1) of
  (left Hhle) ⇒ (normalized_base_case h ? n ?)
| (right H1lth) ⇒
  Cases (decompose h ?) of
    (h', l,Heqh, H_h'_lt_h, H_O_lt_l, H_l_le_h) ⇒
    Cases (div n β^2l ? ?) of
      (n', n'', Hdiv) ⇒
      Cases (div n'' β^l ? ?) of
        (n1, n0, Hdiv') ⇒
        Cases (sqrt h' ? n' ?)of
          (s', (r', Hsqrtrem)) ⇒
          Cases (div r'β^l + n1 2s' ? ?) of
            (q, r'', Hdiv1) ⇒
            Cases (Z_le_gt_dec 0 (r''β^l + n0 − q^2)) of
              (left H_0_le_R) ⇒ (s'β^l + q,(r''β^l + n0 − q^2, ?))
            | (right HltR) ⇒
              (s'β^l + q − 1,(r''β^l + n0 − q^2 + 2(s'β^l + q) − 1, ?))
...
```

FIG. 4.1 – Structure générale de la fonction sqrt_F.

où R est une relation bien-fondée, c'est-à-dire, une relation pour laquelle il n'y a pas de chaîne infinie décroissante. Cette fonction a deux arguments ; le second décrit l'ensemble des appels récursifs possibles pour la fonction. Son type exprime que les seuls appels récursifs légaux doivent être faits sur des valeurs qui sont strictement inférieures à l'argument initial (où R est la relation "est strictement inférieur à"). L'absence de chaînes infinies décroissantes garantit que la fonction récursive terminera un jour. Pour l'algorithme de base, nous utilisons $\mathbb{N}$ comme type d'entrée au lieu de A et l'ordre strict $<$ sur $\mathbb{N}$ comme relation bien-fondée. L'algorithme récursif est donc décrit par la fonction sqrt_F qui a le type suivant.

$$\forall h : nat. \quad (\forall h' : \mathsf{nat}.\ h' < h \to (\mathsf{sqrt\_F\_type}\ h')) \to (\mathsf{sqrt\_F\_type}\ h).$$

On voit donc que les appels récursifs ne peuvent intervenir que sur des valeurs h' qui sont plus petites que l'entrée initiale h. La fonction decompose décrit plus haut nous donne un moyen de produire de tels valeurs, accompagné de la preuve que $h' < h$ dès que h est suffisamment grand.

La structure générale de la fonction sqrt_F est présentée dans la figure 4.1. Tous les points d'interrogation intervenant dans ce texte correspondent aux obligations de preuves que le système engendre au moment de l'acceptation de cette définition. Ces obligations doivent être démontrées par l'utilisateur comme de simples théorèmes. On peut remarquer que l'appel récursif à l'algorithme est décrit par l'expression suivante ($sqrt\, h' ?\, n' ?$). Cet appel requiert d'établir deux certificats, l'un exprime que h' est strictement plus petit que h et l'autre que n' est bien normalisé par rapport à $\beta^{2h'}$.

4.4 Preuve du programme de calcul de la racine carrée

Une fois que nous avons démontré l'algorithme de calcul de la racine carrée au niveau abstrait et que nous disposons d'une description fonctionnelle de l'algorithme dans Coq, il ne reste plus qu'à montrer que le programme donné dans la figure 3.3 calcule effectivement la racine carrée de son entrée. Pour cela, nous allons réutiliser les propriétés établies dans les deux sections précédentes.

4.4.1 Spécification modulaire du programme

Afin d'avoir une preuve plus facile à maintenir, nous avons décomposé le code de la fonction mpn_dq_sqrtrem en plusieurs parties. Chacune de ces parties dispose de ses propres pré- et post-conditions. Cette technique nous a permis de limiter le nombre d'informations inutiles dans le contexte pour la preuve de chaque obligation. De plus, elle permet de modifier localement le programme sans avoir à tout redémontrer si les pré- et post- conditions de la partie modifiée restent inchangées.

Le code est découpé en quatre parties qui correspondent aux principales phases du calcul de la racine carrée :

- l'appel récursif à la fonction mpn_dq_sqrtrem,
- la division par $2S'$,
- l'élévation au carré de Q et sa soustraction,
- la correction du résultat si le reste est négatif.

On montre la validité de l'appel récursif à mpn_dq_sqrtrem de la même façon que la validité de l'appel à la fonction sqrt dans la description fonctionnelle. On doit proposer une relation bien fondée et fournir une quantité (le variant) qui diminue à chaque appel récursif. Ici, on utilise la longueur (le nombre de mots-mémoire) de l'entrée. Cette valeur est, approximativement, divisée par 2 à chaque appel.

Toutes ces étapes de calcul peuvent être reliés à certaines lignes du programme mpn_dq_sqrtrem présenté à la figure 4.1. La correspondance s'établit comme suit :

appel récursif	Cases $(sqrt\ h'\ ?\ n'\ ?)$ of
division par $2S'$	Cases $(\mathsf{div}\ r'\beta^l + n_1\ 2s'\ ?\ ?)$ of
calcul du carré de Q ...	Cases $(\mathsf{Z\_le\_gt\_dec}\ 0\ (r''\beta^l + n_0 - q^2))$ of
correction ...	$(s'\beta^l + q - 1, (r''\beta^l + n_0 - q^2 + 2(s'\beta^l + q) - 1, ?))$

Dans la figure 4.2, nous donnons le code annoté du programme de calcul de la racine carrée. C'est exactement le programme qui a été construit à partir du code C et qui est fourni à l'outil Correctness afin d'engendrer les obligations de preuve. Les numéros des lignes utilisés dans les paragraphes suivants correspondent à celles de la figure 3.3.

4.4.2 Soustraction du carré de Q

La valeur Q est représentée par $q\beta^l + \{s_p, l\}$ à la ligne 20, où le carré de $\{s_p, l\}$ est calculé. Ce calcul est suffisant pour déterminer le carré de Q, puisque l'on sait que $Q \leq L$. A partir de ce résultat, on peut en déduire que q ou $\{s_p, l\}$ est nul et que le double produit (issu du développement du carré) est toujours nul. Cette propriété sera utile aux lignes 21 et 22, où $R''L + N_0$ est représenté par $c\beta^n + \{n_p, n\}$ avant la ligne 21 et Q^2 est représenté par $q\beta^{2l} + \{n_p + n, 2l\}$. A la ligne 21, on calcule seulement $\{n_p, 2l\} - Q^2$. Quand $h \neq l$, une soustraction supplémentaire de la retenue de la soustraction doit être effectuée sur le mot-mémoire de poids fort de $\{n_p, n\}$, tout en mettant à jour

```
Correctness sqrt_root
  let rec mpn_dq_sqrtrem (sp : nat)(np : nat)(nn : nat) : unit
          {variant nn for lt} =
{sp + nn ≤ bound ∧ np + 2 * nn ≤ bound
 ∧0_ℤ < {np, 2 * nn}_m ∧ (np + 2 * nn ≤ sp ∨ sp + nn ≤ np)
 ∧(IsNormalized {np, 2 * nn}_m β^(2*nn)) ∧ O_nat < nn}
  (let r = ref true in
   let q = ref 0_ℤ in let c = ref 0_ℤ in let b = ref 0_ℤ in
   let l = ref O_nat in let h = ref O_nat in
      begin
        if nn = (1)
           then begin (mpn_sqrtrem2 sp np np) end
           else begin
              l := (div2' nn);
              h := nn − !l;
              (mpn_dq_sqrtrem sp+!l np + 2∗!l !h);
              q := (bool2Z !rb);
assert{(SqrtremProp {np + 2 * l, 2 * h}_{m@_0}
                          {sp + l, h}_m   {np + 2 * l, h}_m + qβ^h)
        ∧(∀p : nat.0 ≤ p < bound ⇒
              ¬((np + 2l ≤ p ∧ p < np + 2nn) ∨ (sp + l ≤ p ∧ p < sp + nn)) ⇒
              m@0[p] = m[p])   ∧ (q = 0_ℤ ∨ q = 1)};
label after_recursive_call;
              (division_step np sp nn l h c q);
assert {(Zdivprop'
                    q_@after_recursive_call β^nn + {np + l, nn}_m@after_recursive_call
                        2 * {sp + l, h}_m@after_recursive_call
                        q * β^l + {sp, l}_m   cβ^h + {np + l, h}_m)
        ∧(∀p : nat.0 ≤ p < bound ⇒
           ¬((sp ≤ p ∧ p < sp + l) ∨ (np + l ≤ p ∧ p < np + 2l + h)) ⇒
           m[p] = m_@after_recursive_call[p])   ∧ 0 ≤ q ≤ 1}
              (square_s_and_sub np sp nn l h q c b);
              (mpn_add_1 sp+!l sp+!l !h !q);
              q := (bool2Z !rb);
              (correct_result np sp nn l h q c b);
              rb := (Z2bool !c)
         end
      end)
{(SqrtremProp {np, 2 * nn}_m@ {sp, nn}_m {np, nn}_m + (Zmulbool rb β^nn))
  ∧∀p : nat.0 ≤ p < bound ⇒
  ¬((np ≤ p ∧ p < np + 2nn) ∨ (sp ≤ p ∧ p < sp + nn)) ⇒  m_@0[p] = m[p]}.
```

FIG. 4.2 – Le code annoté de la racine carrée

la valeur de c. Finalement, la valeur $R''L + N_0 - Q^2$ est finalement représenté par $c\beta^n + \{n_p, n\}$ à partir de la ligne 22, où $-1 \leq c \leq 1$.

Avant la ligne 23, $\{s_p, n\}$ ne contient pas exactement le nombre $S'L+Q$. En fait, Q est représenté (depuis la ligne 18) par $q\beta^l + \{s_p, l\}$, de telle sorte que $\{s_p, n\}$ vaut $S'L + Q - qL$. Ceci est corrigé à la ligne 23 du programme.

4.4.3 Correction du résultat

Après la ligne 23, la valeur $R''L + N_0 - Q^2$ se trouve dans la mémoire à la localisation $c\beta^n + \{n_p, n\}$. Si cette valeur est négative, seul le signe c peut en être la cause, par conséquent il suffit de tester le signe de c pour savoir si une correction est nécessaire ou non.

La valeur $S'L+Q$ est représentée dans la mémoire par $q\beta^n+\{n_p, n\}$, cela signifie que la correction effectuée sur le reste aux lignes 26 et 27 comprend aussi l'ajout de $2q$ à c. La soustraction de la ligne 28 correspond à la correction de la racine carrée pour calculer $S'L + Q - 1$. La retenue retournée par cette soustraction est soustraite de q, mais cela est inutile puisque la valeur de q sera *jetée* à la fin de l'exécution de la fonction. De toute façon, nous savons que la valeur finale de q sera obligatoirement 0.

4.5 Division efficace pour le calcul de racine carrée

La division par $2S'$ s'effectue en plusieurs étapes (lignes 13-19 incluses). La technique utilisée permet d'éviter de multiplier S' par 2 et de stocker $2S'$ dans la mémoire. La première étape est de soustraire S' de R' si R' est plus grand que H. Cette soustraction correspond à la ligne 13 du programme.

Le calcul effectif est $(R' - H) - S'$, puisque seul le nombre $R' - H$ est présent dans la mémoire à l'adresse $\{n_p + 2l, h\}$. Cette soustraction retourne nécessairement une retenue. En effet, on sait que $R' \leq 2S' < H + S'$. Pour cette raison, la valeur stockée à l'adresse $\{n_p + 2l, h\}$ après la soustraction est

$$R' - H - S' + H = R' - S'.$$

Dans le cas où $(R' \geq H)$, c'est la valeur $(R' - S')L + N_1$qui est divisé par S'. Cela donne un quotient Q_0 et un reste R_0 et nous avons l'égalité suivante :

$$(R' - S')L + N_1 = Q_0 S' + R_0,$$

que l'on peut transformer en celle-ci :

$$R'L + N_1 = (L + Q_0)S' + R_0.$$

La division effective de $(R'-S')L+N_1$ par S' se fait en utilisant la fonction fonction mpn_divrem de GMP. Cette fonction est seulement spécifiée pour faire des divisions sur des nombres *normalisés*, c'est-à-dire dont le bit de poids fort vaut 1. Dans ce cas, le nombre de mots-mémoire nécessaire pour représenter le quotient peut être déterminé à l'avance : c'est la différence entre le nombre de mots-mémoire du dividende et le nombre de mots-mémoire du diviseur. Cependant, ce calcul peut conduire à un débordement d'au plus 1 bit. Dans notre cas, ce bit est ajouté à q à la ligne 14. q devient donc une valeur comprise entre 0 et 2. Après la ligne 14, le quotient de la division de $R'L + N_1$ par S' est $q\beta^l + \{s_p, l\}$.

A la ligne 15, on détermine le bit de parité de Q_0. Les lignes 16 à 18 calculent la division par 2 du quotient Q_0 et nous obtenons une valeur de q telle que $0 \leq q \leq 1$.

Cette division s'effectue par un simple décalage vers la droite, comme c'est le cas pour les machines utilisant le codage binaire des nombres. Dans notre développement formel, nous avons évité la tâche de montrer la correspondance entre les opérations de décalage et les opérations arithmétiques de division ou de multiplication par une puissance de 2. Pour la même raison, nous avons replacé l'opération

```
sp[l-1] |= q << (BITS_PER_MP_LIMB - 1)
```

par l'addition de $\frac{\beta}{2}$ si q est impair (dans ce cas, q vaut obligatoirement 1), et 0 sinon.

Si le quotient Q_0 est impair, la valeur du reste doit être corrigée en rajoutant S' à R_0. Cette correction est faite à la ligne 19. Elle peut retourner un débordement qui est alors stocké dans la variable locale c.

4.6 Ingénierie de preuves

Le graphe de dépendance entre les différents fichiers de la preuve formelle de correction du programme de calcul de la racine carrée de Gmp est donné à la figure 4.4.

4.6.1 L'approche à trois niveaux

Nous avons prouvé la correction de la racine carrée de Gmp à trois niveaux d'abstraction différents. Nous avons prouvé :

- une description mathématique,
- une description fonctionnelle (qui donne un moyen effectif de calculer la racine carrée),
- une description formelle de l'implantation réelle telle qu'elle est faite dans la bibliothèque Gmp.

La plupart des théorèmes démontrés dans la partie mathématique sont réutilisés dans la preuve de l'implantation. Réutiliser ces preuves demande de relier entre eux des segments de la mémoire avec les variables entières les représentant dans la description abstraite en utilisant la fonction d'interprétation $m, p, l \mapsto \{p, l\}_m$.

Par exemple le théorème QleL qui exprime que $Q \leq L$ a l'énoncé suivant :

$$\begin{array}{l}\forall N, L, H, H', N', N'', S', R', N_1, N_0, Q, R''.\\ H = 2H' \quad \wedge \quad 0 < L \quad \wedge \quad L \leq H \quad \wedge \quad (\mathsf{Zdivprop}\ N\ L^2\ N'\ N'') \quad \wedge\\ (\mathsf{Zdivprop}\ N''\ L\ N_1\ N_0) \quad \wedge \quad (\mathsf{Zdivprop}\ R'L + N_1\ 2S'\ Q\ R'') \quad \wedge\\ (\mathsf{SqrtremProp}\ N'\ S'\ R') \quad \wedge \quad (\mathsf{IsNormalized}\ N\ (HL)^2)\\ \Rightarrow Q \leq L.\end{array}$$

Toutes les valeurs N, N', ... sont représentées quelque part dans la mémoire, il s'agit parfois de combinaisons de plusieurs segments de la mémoire, ou de l'addition d'un segment de la mémoire avec une retenue multipliée par une puissance de la base β. Pour cette raison, on peut instancier le théorème avec certaines de ces valeurs, dans une commande de la forme suivante :

Apply QleL with
$N := \{n_{p_0}, 2n\}_{m_0}$ $H := \beta^h$ $H' := H'$
$N' := \{n_{p_0} + 2l, 2h\}_{m_0}$
$N'' := \{n_{p_0}, 2l\}_{m_0}$ $N_1 := \{n_{p_0} + l, l\}_{m_0}$ $N_0 := \{n_{p_0}, l\}_{m_0}$
$S' := \{s_{p_0} + l, h\}_{m_1}$ $R' := rb_1\beta^h + \{n_{p_0} + 2l, h\}_{m_1}$
$R'' := c_0\beta^h + \{n_{p_0} + l, h\}_{m_2}$.

```
Correctness division_step
  fun (np : nat)(sp : nat)(nn : nat)(l : nat ref)(h : nat ref)
        (c : ℤ ref)(q : ℤ ref) →
  {1 < nn ∧ (l = (div2' nn)) ∧ (h = nn − (div2' nn)) ∧ (q = 0_ℤ ∨ q = 1)
  ∧np + 2 ∗ nn ≤ bound ∧ sp + nn ≤ bound
  ∧(np + 2 ∗ nn ≤ sp ∨ sp + nn ≤ np)
  ∧(q = 1 → {np + 2l, h}_m < {sp + l, h}_m)
  ∧β^h ≤ 2 ∗ {sp + l, h}_m}
     begin
        (if !q ≠ 0_ℤ
           then (mpn_sub_n np + 2∗!l np + 2∗!l sp+!l !h)
           else tt)
        {{np + 2 ∗ l, h}_m@ + q ∗ β^h − q ∗ {sp + l, h}_m@ = {np + 2 ∗ l, h}_m
          ∧∀p : nat.0 ≤ p < bound ⇒
            ¬(np + 2 ∗ l ≤ p ∧ p < np + 2 ∗ l + h) ⇒ m[p] = m@[p]};
        (mpn_divrem sp np+!l nn sp+!l+!h);
        q :=!q + (bool2Z !rb);
        (single_and_1 sp);
        c := !rz;
        (mpn_rshift2 sp sp !l);
label before_assignment;
        (m[sp + (pred !l)] := (I_or sp + (pred !l) (shift_1 !q))
           {(modZtoZ result) =
                 (modZtoZ (m_@before_assignment[sp + (pred l)] + (aux_shift_1 q)))})
        {(modZtoZ m[sp + (pred l)]) =
             (modZtoZ m@[sp + (pred l)] + (aux_shift_1 q))
         ∧∀p : nat.0 ≤ p < bound ∧ p = sp + (pred l) ⇒ m[p] = m@[p]};
        q := (shift_2 !q);
        if !c ≠ 0_Z then
           begin
              (mpn_add_n np+!l np+!l sp+!l !h);
              c := (bool2Z !rb)
           end
     end
  {(Zdivprop' q@ ∗ β^nn + {np + l, nn}_m@   2 ∗ {sp + l, h}_m@
                q ∗ β^l + {sp, l}_m   c ∗ β^h + {np + l, h}_m)
  ∧∀p : nat.0 ≤ p < bound ⇒
    ¬((sp ≤ p ∧ p < sp + l) ∨ (np + l ≤ p ∧ p < np + 2l + h)) ⇒ m[p] = m@[p]
  ∧0_ℤ ≤ q ≤ 1}.
```

FIG. 4.3 – Le code annoté de la division par $2 * S'$

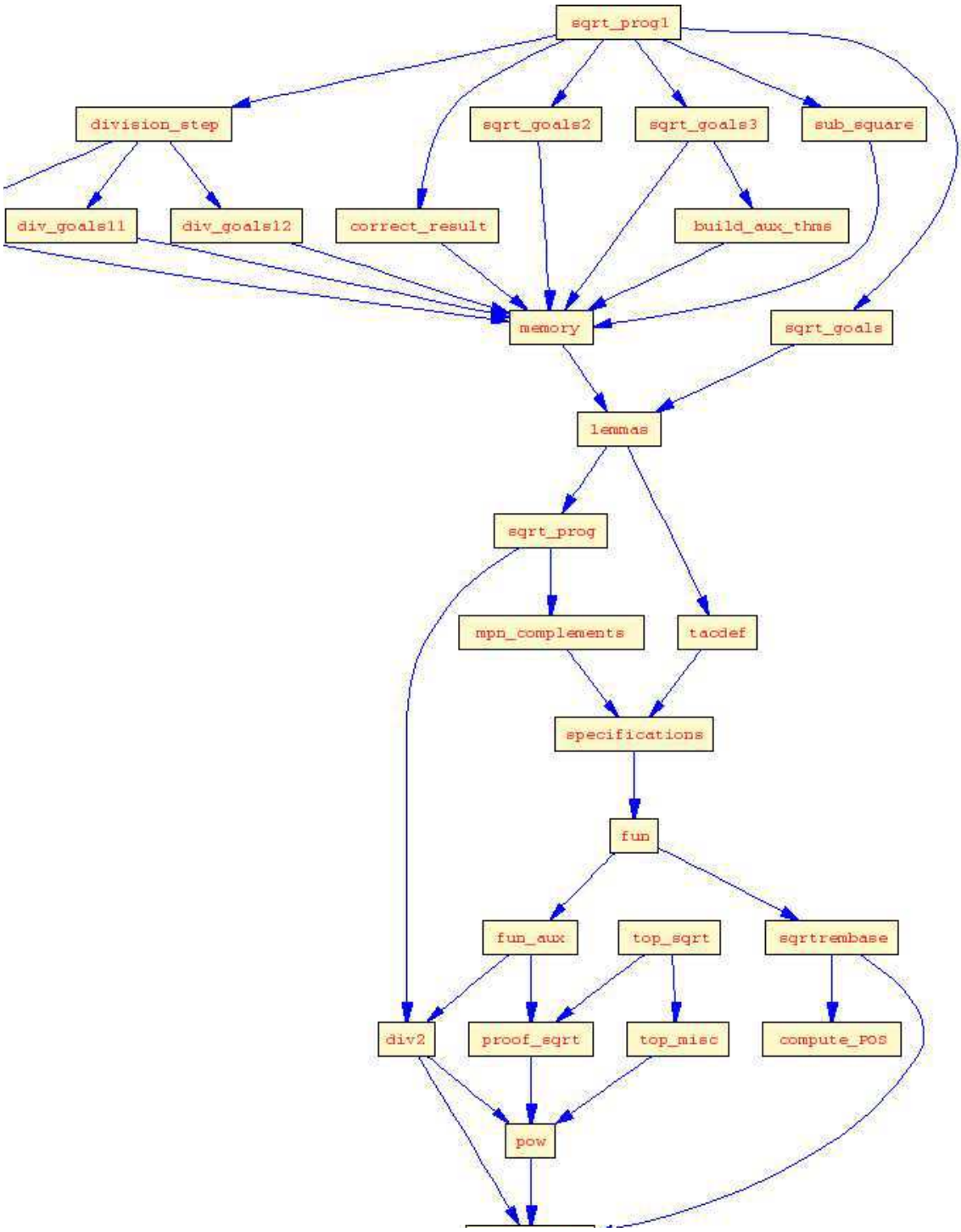

FIG. 4.4 – Graphe de dépendance de la preuve formelle de la racine carrée de GMP

De telles commandes apparaissent effectivement dans le développement formel. Dans cet exemple, les valeurs de Q et L ne sont pas fournies mais automatiquement inférées par l'outil de preuve à partir du contexte.
Trois états de la mémoire sont pris en compte. L'état m_0 correspond à la mémoire initiale au moment où l'appel récursif se produit, m_1 correspond à l'état de la mémoire après l'appel récursif à la fonction de calcul de la racine carrée. La variable m_2, quant à elle, dénote l'état après la division de $R'L + N_1$ par $2S'$. Les différentes prémisses du théorème QleL sont démontrées à l'aide des spécifications des fonctions de GMP, comme mpn_sub_n.

4.6.2 Évolution des preuves

L'évolution des preuves et leur maintenance s'est avérée une question importante dans ce travail. En effet, l'outil CORRECTNESS s'est amélioré au fur et à mesure des versions successives de COQ (de 6.3.1 à 7.4 pour la dernière). De plus la spécification formelle ainsi que les spécifications des fonctions de GMP ont été adaptées au cours du développement complet de la preuve.

4.6.2.1 Robustesse des preuves

Identification des pré- et post-conditions Au fil des développements, il est apparu intéressant de nommer les pré- ou post- conditions. Cela permet, au cours d'une modification mineure de la preuve formelle, de reconnaître rapidement le contexte dans lequel on se trouve lorsqu'un des scripts de démonstrations retourne une erreur.

Noms des hypothèses Les noms des hypothèses ne doivent pas être générés automatiquement par le système lorsqu'ils seront réutilisés plus tard dans la preuve. Par exemple, la séquence Intros ; Omega. ne pose pas de problème au niveau de la robustesse des scripts de preuves. Par contre, les deux tactiques suivantes Intros. Rewrite H. où H est un nom d'hypothèses engendré automatiquement au moment de l'application de la tactique Intros vont causer des problèmes lors de la maintenance. Pour décompose une hypothèse formée de plusieurs conjonctions, il est plus raisonnable d'éviter la tactique Decompose [and] Pre_1 et de lui préférer la suite de tactiques Generalize Pre_1 ; Intros (Pre_1a, Pre_2a...) qui permet de garder le contrôle sur les noms d'hypothèses au cours de la démonstration.

4.6.2.2 Amélioration des performances : GOmega

Au cours de ce développement, nous avons remarqué que la tactique Omega met de plus en plus de temps pour résoudre le but courant quand le contexte devient grand. Pour réduire ce problème d'efficacité, nous proposons un nouvel outil Gomega destiné à réduire la taille du contexte avant l'application de la tactique Omega. L'utilisateur fournit en paramètres à cette tactique la liste des hypothèses utiles à la résolution du but par Omega. Cette nouvelle tactique permet d'alléger le contexte de toutes les hypothèses ne traitant pas de la gestion de la mémoire au moment de prouver qu'un accès à la mémoire est légal par exemple. Associée à un outil d'analyse de dépendances entre théorèmes, une telle tactique permet de déterminer les conditions minimales à réunir pour que l'énoncé soit prouvable. Cela impose à l'utilisateur le travail de sélectionner les hypothèses utiles pour prouver le but courant.

4.6.2.3 Vue d'ensemble

Au total, le développement formel représente plus de 13000 lignes. 2700 lignes correspondent à la preuve au niveau arithmétique. 500 servent à la description fonctionnelle de l'algorithme. Finalement, plus de 10000 lignes sont consacrées à la preuve de l'implantation impérative de cet algorithme. La vérification des preuves par le système Coq nécessite une compilation de 10 minutes sur une machine bi-processeurs équipée de processeurs Pentium III 1 GHz et de 1 GByte de mémoire. Au cours de la compilation, Coq utilise au plus 300 MBytes de mémoire.

4.7 Travaux futurs

L'outil Correctness va être définitivement remplacé par Why [Fil03]. Afin d'assurer la pérennité de notre développement formel, nous avons porté notre preuve de correction vers Why. Le code de Gmp évolue fréquemment, il serait donc intéressant de développer des outils de maintenance des preuves permettant de mettre à jour facilement la preuve de correction à chaque changement de l'implantation du programme.
Finalement, l'outil Why permet de prendre en entrée des programmes C annotés. L'ultime étape serait donc de faire une preuve de correction directement sur le code C plutôt que de passer par un langage intermédiaire à la ML comme c'est le cas avec Correctness. Ainsi on disposerait d'un programme C de calcul de la racine carrée. Ce programme (contenant des annotations en commentaires) pourrait être envoyé soit vers le compilateur C, soit vers l'outil Why afin d'engendrer les obligations de preuves à démontrer pour s'assurer de la correction de ce programme.

4.8 Conclusion

Le travail de formalisation et de démonstration formelle effectué sur le programme de calcul de la racine carrée de Gmp a permis d'en prouver la correction totale au niveau arithmétique, de montrer que la gestion de la mémoire et l'arithmétique de pointeurs utilisée dans ce programme étaient corrects. De plus, ce travail de formalisation nous a permis de découvrir une propriété supplémentaire de ce programme. La base utilisée par le programme n'était initialement pas spécifiée dans la démonstration. Habituellement la base considérée lors de l'utilisation de la bibliothèque Gmp est $\beta = 2^{32}$ ou $\beta = 2^{64}$. En fait, pour que le programme de calcul de la racine carrée de Gmp soit correct, il suffit d'utiliser une base β qui soit paire et strictement supérieure à 2. Évidemment, les bases 2^{32} et 2^{64} conviennent. Cependant il est intéressant de remarquer que cet algorithme peut être utilisé pour calculer la racine carrée en base 10.

Récemment, J. Sawada et R. Gamboa [SG02] ont prouvé formellement la correction de l'algorithme de calcul de la racine carrée flottante du processeur IBM Power4$^{\text{TM}}$. Cet algorithme utilise des polynômes de Chebyschev pour affiner l'approximation de la racine carrée obtenu à chaque étape. La preuve est développée dans le système d'aide à la preuve ACL2(r) qui dispose d'une bibliothèque d'analyse réelle non standard ; elle a nécessité la définition et la preuve de nombreuses propriétés d'analyse sur les polynômes de Chebyschev et de Taylor ainsi que la démonstration du théorème de Taylor.

Deuxième partie

Changements de représentation des données

Le choix d'un type inductif particulier pour représenter une notion mathématique dans Coq influence fortement la manière dont les fonctions seront définies ainsi que la manière dont les preuves seront développées. Il est cependant intéressant de pouvoir privilégier l'une ou l'autre des représentations concrètes dans certaines preuves. Au niveau de la programmation, P. Wadler a proposé à la fin des années 80, la notion de *vues* [Wad87] (views en anglais) afin de pouvoir regarder une notion mathématique de plusieurs points de vue différents. D'autre part, une approche comme le Calcul des Constructions Algébriques proposé par F. Blanqui dans sa thèse [Bla01] peut être vue comme une alternative à notre travail. Elle permet en effet de voir les notions mathématiques et les fonctions de plusieurs points de vue différents. Cela donne une vue de plus haut niveau des données et peut éviter d'avoir à faire des changements de représentation.

Dans cette partie, nous étudions plus particulièrement comment réutiliser des preuves formelles développées dans un système d'aide à la preuve comme Coq. Nous nous intéressons plus particulièrement à la question suivante : comment rendre les preuves indépendantes de la représentation concrète des données ? Imaginons que l'on dispose d'une théorie développée formellement avec une représentation inductive particulière d'une notion mathématique. On souhaite construire, de manière la plus automatique possible, une nouvelle théorie formelle sur la même notion mathématique mais en utilisant une représentation différente.

Le système Coq permet d'associer aux fonctions définies par récursion structurelle un comportement calculatoire (la ι-réduction). Cette réduction rend implicite certaines étapes de raisonnement dans les termes de preuve. Afin de pouvoir transformer une théorie, il faut tout d'abord mettre en évidence ces étapes de raisonnement calculatoire (implicite) pour les remplacer par des étapes de raisonnement équationnel (explicite). Une fois ces étapes de calcul mises en évidence, il suffit de relier syntaxiquement les éléments homologues dans les représentations initiale et finale.

Dans le chapitre 5, nous décrivons ce que signifie *changement de représentation* dans une théorie développée formellement. Nous proposons ensuite, dans le chapitre 6, des moyens de faire abstraction de la représentation concrète des données dans une preuve formelle. Dans le chapitre 7, nous étudions comment réinstancier les preuves avec une nouvelle représentation concrète des données considérées. Finalement, dans le chapitre 8, nous présentons une approche alternative pour le changement de représentation des données ainsi qu'une extension de la méthode proposée dans les chapitres précédents en présence de types dépendants.

Table des matières

Chapitre 5

Passage d'une représentation à une autre

Dans ce chapitre, nous présentons les principales motivations de notre travail sur les changements de représentation des données dans le calcul des constructions inductives. Nous commençons par décrire les moyens que fournit Coq pour définir des types de données qu'ils soient concrets ou abstraits. Nous verrons aussi comment calculer dans Coq que ce soit en utilisant la récursion structurelle ou la récursion générale. Nous donnons ensuite, à travers quelques exemples de changements de représentations intéressants, les motivations de notre travail. Enfin, nous présentons quelques travaux connexes sur ces questions de réutilisation des preuves lors d'un changement de représentation.

5.1 Caractéristiques générales de Coq

Dans cette section, nous présentons les caractéristiques de Coq qui seront utilisés dans la suite de ce document.

Le système Coq est l'implantation d'un formalisme logique d'ordre supérieur : le Calcul des Constructions Inductives [Coq02, Chap. 4]. Il s'agit d'un λ-calcul typé avec types dépendants permettant en outre la définition de constructions inductives [PM93].

Coq propose un mode interactif de développement de preuve par chaînage arrière. On dispose d'un certain nombre de tactiques, applicables au but courant si certaines préconditions sont vérifiées. Elles permettent d'effectuer les pas élémentaires de preuve. Dans ce calcul, les preuves sont des termes du λ-calcul comme les autres. L'isomorphisme de Curry-Howard permet d'interpréter les types du calcul comme des formules logiques et les λ-termes comme des preuves. L'objet dénotant la preuve d'un théorème est conservé ; il est donc possible de vérifier une preuve par une simple opération de vérification de type.

Nous commençons par décrire comment définir des types de données dans Coq ; ensuite nous montrons comment calculer sur de tels types de données. Enfin nous décrivons les différentes formes d'égalité disponibles dans Coq et leurs propriétés.

5.1.1 Types de données dans Coq

Le système Coq offre la possibilité de construire des types de données concrets sous forme de définitions inductives [Gim98]. Néanmoins, il est aussi possible de décrire une notion mathématique

de façon complètement abstraite au moyen de paramètres et d'axiomes exprimant les propriétés que doivent vérifier ces paramètres.

5.1.1.1 Types concrets

Définitions inductives Dans Coq, un type inductif est défini par la donnée d'un ensemble de fonctions dont le codomaine est le type lui-même et dont les arguments peuvent être des éléments du type en cours de définition. De telles fonctions s'appellent des constructeurs. A titre d'exemple, nous présentons le type des entiers naturels nat. Ce type est défini de la façon suivante :

```
Inductive nat : Set := O : nat | S : nat -> nat.
```

O et S sont les constructeurs du type inductif nat. O est un constructeur sans argument, c'est-à-dire un élément de type nat. S est une fonction qui, à partir d'un élément de nat, construit un nouvel élément de nat.

Les constructeurs d'un type inductif sont injectifs. De plus, ils sont libres. Cela signifie que chaque élément d'un type inductif s'écrit d'une manière unique au moyen de ses constructeurs.

Calculer avec les définitions inductives Dans le système Coq, on dispose de moyens effectifs pour calculer sur les types inductifs. Dans sa présentation initiale des définitions inductives dans le calcul des constructions [PM93], C. Paulin utilise un système avec des opérateurs de récursion primitive. L'idée de dissocier récursion et analyse par cas est dû à T. Coquand. Cette idée a été adaptée par C. Paulin pour être implanté dans Coq. Dans la présentation du calcul des constructions telle qu'elle est faite dans son habilitation [PM96], on considère deux opérateurs généraux : l'opérateur de traitement par cas case et l'opérateur de récursion fix. La construction case permet de faire de l'analyse structurelle sur un élément d'un type défini inductivement. La construction fix permet d'écrire des fonctions récursives structurelles. Cette construction est associée à une condition de garde. Cette condition sert à assurer que tous les appels récursifs à la fonction définie avec fix se font bien sur des éléments (syntaxiquement) plus petits que l'argument courant. Cette condition de garde distingue un des arguments de la fonction, celui qui assure la terminaison de la fonction. Dans la suite, nous appelons cet argument, argument *principal* de la fonction.

Par exemple, l'addition sur les entiers de Peano définis plus haut peut se faire comme suit :

```
Fixpoint plus [n:nat] : nat -> nat :=
[m:nat] Cases n of O     => m
                | (S p) => (S (plus p m))
        end.
```

Le premier argument de plus est l'argument principal. On remarque que l'appel récursif à plus se fait sur p qui est bien un sous-terme strict de n.

A une telle définition sont associées des règles de ι-réduction. Dans le cas de plus, ces règles expriment que :

$$(\mathsf{plus}\ \mathsf{O}\ n) \xrightarrow{\iota} n \qquad (\mathsf{plus}\ (\mathsf{S}\ n)\ m) \xrightarrow{\iota} (\mathsf{S}\ (\mathsf{plus}\ n\ m))$$

Ce sont ces règles de réduction qui permettent de calculer dans Coq. C'est grâce à ces règles que la preuve de l'énoncé (plus (2) (3)=(5)) se fait simplement par réflexivité de l'égalité. Le calcul de (plus (2) (3)) vers (5) se fait par réduction et n'apparaît donc pas dans le terme de preuve. De même, ces règles assurent que les types $(\mathsf{plus}\ \mathsf{O}\ \mathsf{O}) = \mathsf{O}$ et $\mathsf{O} = \mathsf{O}$ sont convertibles.

Ces règles de calcul sur les types inductifs permettent de cacher une partie du raisonnement fait dans les démonstrations formelles ; cette caractéristique a été intensivement utilisée pour faire des

preuves par réflexion [BRB95, Bou97, GWZ00]. De telles preuves utilisent la capacité de calculer de Coq pour faire des preuves.

Principes d'induction associés Au moment de la définition de nat, le système Coq génère automatiquement des principes de récursion (un pour chaque sorte Prop, Set et Type). L'opérateur de récursion pour la sorte Set (nat_rec) s'énonce de la façon suivante :

$$\mathsf{nat\_rec} : \forall P : \mathsf{nat} \to \mathsf{Set}.\ (P\ \mathsf{O}) \to (\forall n : \mathsf{nat}.\ (P\ n) \to (P\ (\mathsf{S}\ n))) \to \forall n : \mathsf{nat}.\ (P\ n)$$

Principes dépendants et principes non-dépendants
Le système Coq permet la définition de deux sortes de principes de récursion : les principes dépendants (ou maximaux) et les principes non dépendants (aussi appelés minimaux) [BC03].

En plus du principe dépendant pour nat que nous venons de voir, on peut définir dans Coq un principe non-dépendant. Il correspond à l'opérateur de récursion dans le système T de Gödel. Ce principe minimal nat_min_rec a le type suivant :

$$\forall P : \mathsf{Set}.\ P \to (\mathsf{nat} \to P \to P) \to \mathsf{nat} \to P \tag{5.1}$$

Un tel opérateur permet de définir directement une opération comme plus.

$$\begin{array}{lll} \mathsf{plus'} & = & \lambda n : \mathsf{nat}. < A0 > (\mathsf{nat\_min\_rec}\ \ \mathsf{nat} \to \mathsf{nat} \\ & & \qquad\qquad\qquad\qquad\quad \lambda m : \mathsf{nat}.m \\ & & \qquad\qquad\qquad\qquad\quad \lambda\_ : \mathsf{nat}.\lambda vr : \mathsf{nat} \to \mathsf{nat}.\lambda m : \mathsf{nat}.(\mathsf{S}\ (vr\ m)) \\ & & \qquad\qquad\qquad\qquad\quad n) \end{array}$$

Règles de réduction associées aux principes de récursion
En fait, les principes de récursion sont des fonctions comme les autres. Ils sont définis dans Coq au moyen des constructions de traitement par cas case et de définition par point-fixe fix. Les principes dépendants et non dépendants sont définis de façon semblable. Par exemple, le principe de récursion dépendant sur les entiers de Peano est défini par :

```
nat_rec = [P:(nat->Set); f:(P (0)); f0:((n:nat)(P n)->(P (S n)))]
 Fix F {F [n:nat] : (P n) :=
          <P>Cases n of (0) => f
                      | (S n0) => (f0 n0 (F n0))
             end}
```

De même que les fonctions de base comme l'addition, les principes de récursion ont, à cause de leur définition, un comportement calculatoire. Les règles de réduction pour nat_rec peuvent être exprimées par :

$$(\mathsf{nat\_rec}\ P\ v0\ vr\ \mathsf{O}) \xrightarrow{\iota} v0$$
$$(\mathsf{nat\_rec}\ P\ v0\ vr\ (\mathsf{S}\ n)) \xrightarrow{\iota} (vr\ n\ (\mathsf{nat\_rec}\ P\ v0\ vr\ n))$$

Prédicats inductifs Les types inductifs permettent aussi de définir des prédicats. L'ordre sur les entiers naturels le est défini par un type inductif. Ce type est paramétré par un entier n. Le constructeur le_n exprime que $n \leq n$ et le_S indique que si l'on a $n \leq m$, alors on peut en déduire l'inégalité $n \leq (\mathsf{S}\ m)$.

```
Inductive le [n : nat]  : nat->Prop :=
      le_n : (le n n) | le_S : (m:nat)(le n m)->(le n (S m))
```

La définition d'un prédicatif inductif comme le déclenche la construction d'un opérateur de récursion le_ind. En effet, les règles d'élimination autorisées dans Coq ne permettent pas de construire des éléments de sorte Type ou Set par filtrage sur un élément de sorte Prop.

$$\begin{aligned}\mathsf{le\_ind}: \quad & \forall n : \mathsf{nat}.\ \forall P : \mathsf{nat} \to \mathsf{Prop}.\\ & (P\ n) \to (\forall m : \mathsf{nat}.\ (\mathsf{le}\ n\ m) \to (P\ m) \to (P\ (\mathsf{S}\ m))) \to\\ & \forall n0 : \mathsf{nat}.\ (\mathsf{le}\ n\ n0) \to (P\ n0)\end{aligned}$$

On notera que cet opérateur est non dépendant, c'est-à-dire que P est de type $\mathsf{nat} \to \mathsf{Prop}$ plutôt que $\forall n0 : \mathsf{nat}.\ (\mathsf{le}\ n\ n0) \to \mathsf{Prop}$.

Retour sur les principes de récursion Dans la figure 5.1, nous donnons les caractéristiques des opérateurs de récursion générés automatiquement à la suite de la définition d'un type inductif T. L'outil Coq définit différents opérateurs de récursion permettant de construire des expressions de sorte Set, Type ou Prop. Les colonnes de ce tableau représentent la sorte de l'élément sur lequel on applique l'opérateur de récursion alors que les lignes représentent la sorte des éléments que l'on peut construire.

	T : Set	T : Type	T : Prop
$P : T \to$ Set	dépendant	dépendant	interdit
$P : T \to$ Type	dépendant	dépendant	interdit
$P : T \to$ Prop	dépendant	dépendant	non dépendant

Fig. 5.1 – Les éliminations autorisées et leurs degrés de dépendance

Il est important de préciser que lorsque les éliminations sont autorisées, elles peuvent toujours se faire de manière dépendante bien qu'en pratique on ne les utilise quasiment pas. De plus, les éliminations de Set vers Type ne sont possibles que sur les petits types (cf. [PM93] pour une définition des petits types). Enfin, les éliminations de Prop vers Set ou Type ne sont autorisées que dans le cas des types inductifs vides ou à un seul constructeur (par exemple l'égalité de Leibniz eq).

Pour résumer, la définition du type inductif nat (de sorte Set) déclenche la définition automatique des trois principes d'élimination nat_rec, nat_ind et nat_rect. Par contre, la définition de le ne déclenche que la construction d'un seul principe d'élimination (qui est non-dépendant). Dans les deux cas, il est toutefois possible de définir les principes manquants, qu'ils soient dépendants ou non. Cela se fait au moyen de la commande Scheme :

$$\mathsf{Scheme}\ T_{(\mathsf{ind}|\mathsf{rec}|\mathsf{rect})} := (\mathsf{Induction} \mid \mathsf{Minimality})\ \mathsf{for}\ T\ \mathsf{Sort}\ (\mathsf{Prop} \mid \mathsf{Set} \mid \mathsf{Type}).$$

A titre d'exemple, l'opérateur de récursion non dépendant nat_min_rec (5.1) peut être généré par la commande suivante : `Scheme nat_min_rec := Minimality for nat Sort Set.`
De la même façon, le principe dépendant associé à le peut être généré par la commande :
`Scheme le_dep_ind := Induction for le Sort Prop.`

5.1.1.2 Types abstraits

Le choix d'une représentation inductive particulière pour un type de données détermine la manière de raisonner sur les objets de ce type. Pour éviter de contraindre la manière de raisonner sur un type de données, on peut choisir de le représenter comme un type abstrait.

On peut représenter un type abstrait dans Coq en le décrivant à l'aide de paramètres et d'axiomes. Par exemple le type nat présenté dans la section précédente peut être décrit par les paramètres suivants :

```
Parameter Anat:Set.
Parameter AO:nat.
Parameter AS:nat->nat.

Axiom AS_inj : (n,m:Anat)(AS n)=(AS m)->n=m.
Axiom AO_AS : (n:Anat)~AO=(AS n).
```

Les trois premières lignes permettent de décrire le type en donnant son nom et ses constructeurs. On donne ensuite les propriétés d'injectivité pour les constructeurs fonctionnels (ici AS). Finalement, on donne en axiome une propriété de non-confusion, dont on peut déduire qu'un nombre ne peut pas s'écrire de deux façons différentes avec ses constructeurs. Avoir une représentation axiomatique d'un type de données ne permet pas de calculer sur ce type. Par conséquent, dès lors que l'on veut programmer, on utilisera plutôt une représentation concrète sous forme de type inductif.

5.1.1.3 Passage d'une structure de données à une autre

Choisir une certaine représentation d'un type de données, c'est non seulement choisir une manière de calculer, mais aussi s'imposer une manière canonique de raisonner sur ces données. On peut pourtant être amené, pour des raisons d'efficacité, à changer de représentation concrète d'un type de données et vouloir conserver les principes de raisonnement qui sont attachés à la représentation initiale. Dans ce cas, il faudra pouvoir définir des fonctions non structurelles pour conserver la structure initiale des données dans les calculs tout en considérant une représentation concrète des données.

5.1.2 Calculs non structurels dans Coq

Comme nous l'avons vu dans la section précédente, il est facile de définir des fonctions en suivant la structure inductive des données sur lesquels elles calculent. Cependant, on veut souvent définir une fonction sans suivre la structure inductive des données qu'elle manipule. Pour cela, on a recours à la notion d'ordre bien fondé et à la définition de fonctions récursives générales.

5.1.2.1 Définitions par ordre bien fondé

Les définitions par ordre bien fondé reposent sur la notion d'accessibilité. Celle-ci est définie en Coq par le type inductif suivant :

```
Inductive Acc [A : Set; R : A->A->Prop]  : A->Prop :=
      Acc_intro : (x:A)((y:A)(R y x)->(Acc A R y))->(Acc A R x)
```

Une relation R sur un ensemble A est dite bien fondée si et seulement si tout élément de A est accessible via R. En termes plus formels, cela s'exprime par la définition suivante :

$$(\mathsf{well\_founded}\ A\ R) \equiv \forall a : A.\ (Acc\ A\ R\ a)$$

Fonction d'itération d'une étape de calcul Pour définir une fonction par ordre bien fondé, il faut commencer par construire une fonction qui décrive les calculs à chaque pas d'exécution de l'algorithme. Une telle fonction a un type de la forme suivante :

$$F : \forall n : A.\ (\forall m : A.\ (R\ m\ n) \to (B\ m)) \to (B\ n)$$

La fonction f de type $(\forall m : A.\ (R\ m\ n) \to (B\ m))$ permet de faire les appels récursifs. Il faut cependant noter qu'on ne peut pas l'appeler sur n'importe quelle valeur m ; il faut que m soit strictement inférieur à n suivant l'ordre R. Ce critère garantie la terminaison de la fonction.

Construction de la fonction A partir de la donnée de A, de R, d'une preuve $\mathsf{wf\_prf}$ que R est une relation bien fondée sur A, du type de retour $\lambda x : A.(B\ x)$, et de la fonction F, on obtient la définition :

$$\mathsf{wf\_F}\ \equiv (\mathsf{well\_founded\_induction}\ A\ R\ \mathsf{wf\_prf}\ \lambda x : A.(B\ x)\ F)$$

où $\mathsf{well\_founded\_induction}$ est une fonction qui a le type suivant :

$\forall A : \mathsf{Set}.\ \forall R : A \to A \to \mathsf{Prop}.$
$(\mathsf{well\_founded}\ A\ R) \to \forall P : (A \to \mathsf{Set}).\ (\forall x : A.\ (\forall y : A.\ (R\ y\ x) \to (P\ y)) \to (P\ x)) \to \forall a : A.\ (P\ a)$

5.1.2.2 Équations de point fixe associées

Maintenant que l'on a défini une fonction récursive générale, on souhaite montrer que cette fonction a bien le comportement de réduction que l'on attendait. Malheureusement les fonctions définies par ordre bien-fondé n'ont pas de règles de réduction associées à leur définition. Néanmoins, il est possible de démontrer une équation de point fixe qui exprime sous forme d'une égalité l'évolution pas-à-pas du calcul.

A. Balaa et Y. Bertot ont proposé dans [BB00] une méthode générale pour dériver une équation de point fixe à partir de la définition d'une fonction récursive générale. Cette méthode permet d'énoncer et de démontrer automatiquement que

$$\forall a : A.\ (\mathsf{wf\_F}\ \mathsf{a}) = (F\ a\ \lambda y : A.\ \lambda h : (R\ y\ x).\ (\mathsf{wf\_F}\ y))$$

à partir de la donnée de F et $\mathsf{wf\_F}$. A partir de cette équation de point fixe générale, on peut dériver une équation de point fixe simplifiée qui décrit le comportement de la fonction de manière plus concrète en fonction des constructeurs $c_1,\ldots,\ c_n$ du type inductif manipulé.

$$\begin{array}{lll} \forall a : A.\ (\mathsf{wf\_F}\ a) = \mathsf{Cases}\ a\ \mathsf{of} & (c\_1\ldots) \Rightarrow \ldots \\ & |\ (c\_2\ x) \Rightarrow (\ldots(\mathsf{wf\_F}\ x)\ldots) \\ & |\ \ldots \\ & \mathsf{end} \end{array}$$

5.1.3 Égalités dans Coq

Nous terminons notre tour d'horizon de Coq par la présentation des différentes égalités qui coexistent dans Coq. Nous commençons par donner les propriétés de base de l'égalité de Leibniz dans Coq. Ensuite nous présentons l'égalité dépendante eq_dep qui sera utile dans la suite de notre travail. Pour une description plus complète des différentes égalités de Coq et de leurs propriétés, nous renvoyons le lecteur intéressé au chapitre 4 de la thèse [Alv02] de C. Alvarado.

5.1.3.1 Égalité de Leibniz

Il n'existe pas de notion propre d'égalité propositionnelle dans Coq. L'égalité de Leibniz est simplement définie par un type inductif paramétré et dépendant. Cette égalité polymorphe est paramétrée par le type des éléments comparés.

Comme c'est le cas habituellement en théorie des types, on distingue deux égalités :

– l'égalité propositionnelle. Il s'agit de l'égalité de Leibniz de Coq (eq) :
```
Inductive eq [A : Set; x : A]  : A->Prop :=  refl_equal : (eq A x x).
```
– l'égalité définitionnelle. Il s'agit de la relation de $\beta\delta\iota$-conversion dans Coq.

Tous les objets égaux pour l'égalité définitionnelle sont égaux pour l'égalité propositionnelle. Par contre, la réciproque n'est pas vraie. Les termes n et ($\mathsf{plus}\ n\ \mathsf{O}$) ne sont pas définitionnellement égaux, par contre il est possible de montrer qu'ils sont propositionnellement égaux. Pour cela, on procède par induction et par réécriture avec l'hypothèse de récurrence.

A l'égalité de Leibniz est associé un principe de substitution des égaux par des égaux ; ce principe s'énonce comme suit :

$$\mathsf{eq\_ind} : \forall A : \mathsf{Set}.\ \forall x : A.\ \forall P : (A \to \mathsf{Prop}).\ (P\ x) \to \forall y : A.\ x = y \to (P\ y)$$

C'est grâce à ce principe de base qu'il est possible de faire de la réécriture dans Coq.

5.1.3.2 Égalité dépendante

L'égalité de Leibniz ne permet que de comparer des éléments dont on sait *a priori* qu'ils sont de même type. En présence de types dépendants, on peut vouloir parler de l'égalité entre deux éléments dont on ne sait pas encore s'ils sont dans la même instance du type inductif dépendant. Considérons par exemple le cas de vecteurs sur lequel nous reviendrons dans le dernier chapitre de ce manuscrit. La définition inductive des vecteurs est la suivante :

```
Inductive vect [A : Set]  : nat->Set :=
      vnil : (vect A (0)) | vcons : (n:nat)A->(vect A n)->(vect A (S n))
```

Reprenons l'exemple classique de l'associativité de la concaténation des vecteurs. La fonction de concaténation a le type suivant

$$\mathsf{concat} : \forall n : \mathsf{nat}.\ (\mathsf{vect}\ A\ n) \to \forall m : \mathsf{nat}.\ (\mathsf{vect}\ A\ m) \to (\mathsf{vect}\ (\mathsf{plus}\ n\ m))$$

Le premier réflexe serait d'énoncer la propriété d'associativité de la manière suivante :

$$\forall n, m, p : \mathsf{nat}.\ \forall v : (\mathsf{vect}\ A\ n); w : (\mathsf{vect}\ A\ m); z : (\mathsf{vect}\ A\ p).$$
$$(\mathsf{concat}\ n\ v\ (\mathsf{plus}\ m\ p)\ (\mathsf{concat}\ m\ w\ p\ z)) = (\mathsf{concat}\ (\mathsf{plus}\ n\ m)\ (\mathsf{concat}\ n\ v\ m\ w)\ p\ z)$$

Malheureusement ce terme n'est pas bien typé. En effet le type du membre gauche de l'égalité est ($\mathsf{vect}\ A\ (\mathsf{plus}\ n\ (\mathsf{plus}\ m\ p))$) alors que le type du membre droit est ($\mathsf{vect}\ A\ (\mathsf{plus}\ (\mathsf{plus}\ n\ m)\ p)$).

Comme (plus n (plus m p)) et (plus (plus n m) p) ne sont pas convertibles, les deux types sont différents. Par conséquent, pour pouvoir énoncer de telles propriétés il faudrait pouvoir parler de l'égalité entre objets de la même famille inductive, mais dont les types ne sont pas nécessairement convertibles. C'est ce que permet de faire l'égalité dépendante proposée par Thierry Coquand. L'égalité dépendante est définie de la façon suivante :

```
Inductive eq_dep [U : Set; P : U->Set; p : U; x : (P p)]
     : (q:U)(P q)->Prop :=  eq_dep_intro : (eq_dep U P p x p x)
```

L'énoncé précédent se réécrit en utilisant l'égalité dépendante comme :

$$\begin{array}{l}\forall n, m, p : \mathsf{nat}.\ \forall v : (\mathsf{vect}\ A\ n); w : (\mathsf{vect}\ A\ m); z : (\mathsf{vect}\ A\ p).\\ (\mathsf{eq\_dep}\ \mathsf{nat}\ (\mathsf{vect}\ A)\\ (\mathsf{plus}\ n\ (\mathsf{plus}\ m\ p))\ (\mathsf{concat}\ n\ v\ (\mathsf{plus}\ m\ p)\ (\mathsf{concat}\ m\ w\ p\ z))\\ (\mathsf{plus}\ (\mathsf{plus}\ n\ m)\ p)\ (\mathsf{concat}\ (\mathsf{plus}\ n\ m)\ (\mathsf{concat}\ n\ v\ m\ w)\ p\ z))\end{array}$$

Relation entre égalité de Leibniz et égalité dépendante Si l'on ajoute à Coq, l'axiome d'unicité des preuves d'égalité[1] (aussi appelé UIP - *Uniqueness of Identity Proofs* en anglais), on peut déduire l'égalité de Leibniz de l'égalité dépendante entre deux éléments de la même instance d'une famille inductive comme vect.

$$\forall p : U.\ \forall x, y : (P\ p).\ (\mathsf{eq\_dep}\ p\ x\ p\ y) \rightarrow x = y$$

L'égalité dépendante sera utile dans la suite de nos travaux pour démontrer certaines propriétés des opérations de filtrage non structurelles. Il existe une alternative à l'égalité dépendante dans Coq : l'égalité de John Major, proposée par C. McBride dans sa thèse [McB99]. Nous aurions également pu l'utiliser comme alternative à l'égalité dépendante de Coq.

5.2 Motivations

Dans la suite de ce document, nous nous intéressons aux différentes représentations concrètes possibles pour une notion mathématique donnée. Nous étudions plus particulièrement les moyens de changer la représentation concrète d'un type de données sans avoir à redémontrer à la main toute la bibliothèque de théories développée pour la représentation initiale.

5.2.1 Pourquoi considérer plusieurs représentations concrètes ?

Une notion mathématique peut avoir différentes représentations concrètes. Chacune de ces représentations concrètes présente des avantages et des inconvénients spécifiques. C'est pourquoi le choix d'une structure de données concrète dépend généralement de l'utilisation que l'on veut en faire. Prenons l'exemple des entiers naturels. Pour calculer efficacement sur ces entiers, on choisira une représentation binaire alors que si l'on veut démontrer des propriétés arithmétiques, il sera plus confortable d'utiliser la représentation unaire (à la Peano) des entiers. En effet, dans ce cas, le principe de récurrence habituel sur les entiers

$$P(0) \wedge (\forall n : \mathsf{nat}.\ P(n) \rightarrow\ P(n+1)) \rightarrow \forall n : \mathsf{nat}.\ P(n)$$

[1] L'axiome d'unicité des preuves d'égalité exprime que le type $x = x$ n'a qu'un seul habitant. En termes plus formels : $\forall x : A.\ \forall p, q : x = x.\ p == q$

se démontre simplement par récurrence sur la structure des entiers de Peano.

P. Wadler a introduit à la fin des années 80 [Wad87] la notion de *vues* (*views* en anglais). Cela consiste à considérer un type de données suivant différents points de vue. Par exemple, les nombres complexes peuvent être vus de deux manières différentes : soit par leur représentation cartésienne $z = a + ib$, soit par leur représentation polaire $z = \rho.e^{i\theta}$. Une liste peut être être lue, soit de la gauche vers la droite, soit de la droite vers la gauche. De même une liste de couples peut être vue comme un couple de listes...

Pour en revenir à l'exemple des entiers naturels, les deux représentations unaire et binaire sont utiles et devraient pouvoir coexister. Si pour des raisons d'efficacité, on se restreint à la seule représentation binaire des entiers naturels, on pourra définir facilement les opérations de base comme l'addition et la multiplication et prouver les propriétés qui leur sont associées. Cependant, on ne pourra pas définir simplement une fonction comme la factorielle. En effet, la structure de l'algorithme de calcul de la factorielle correspond exactement à la structure des entiers de Peano. Il n'est pas possible de programmer la factorielle en suivant la structure des entiers binaires. Dans le cadre des entiers binaires, on devra revenir à une définition par ordre bien fondé.

5.2.2 Mise en correspondance de deux types de données

La mise en correspondance de deux types de données doit être sémantiquement valide pour que les méthodes présentées dans les prochains chapitres soient valides. En particulier, il existe un isomorphisme entre $\mathbb{N}$ et $\mathbb{Z}$, cependant les éléments mis en correspondance ont des interprétations différentes.

Supposons que l'on veuille transformer la représentation A d'une notion mathématique en une représentation B qui lui est isomorphe. La première étape consiste à définir les homologues des constructeurs du type inductif A sous forme de fonctions. Ensuite, il faut associer les fonctions travaillant sur A avec des fonctions *équivalentes* travaillant sur B. Ensuite, il suffit de traduire syntaxiquement les énoncés des théorèmes en remplaçant A par B, les constructeurs de A par leurs homologues dans B et les fonctions sur A par les fonctions équivalentes sur B.

5.2.3 Changements de représentation d'une donnée dans un développement formel

On considère que l'on dispose d'une suite des théorèmes $t_1, t_2, \ldots, t_n$ ordonnés suivant l'ordre de leurs dépendances les uns par rapport aux autres. Ces théorèmes établissent des propriétés sur les éléments du type A. On suppose que $t'_1, t'_2, \ldots, t'_n$ sont les énoncés des théorèmes équivalents à $t_1, t_2, \ldots, t_n$ où le type A a été changé en B. Il existe deux chemins pour démontrer le théorème t'_n. On peut démontrer tous les théorèmes de la théorie basée sur A, puis prouver directement le dernier en utilisant t_n, ou bien reconstruire une théorie $t'_1, t'_2, \ldots, t'_n$ avec comme type de base B au lieu de A.

Le chemin $t_0 \rightarrow t_n \rightarrow t'_n$ correspond à la traduction par l'utilisation directe de l'isomorphisme reliant A et B. Nous étudions cette technique dans le dernier chapitre de cette thèse. Le chemin $t_0 \rightarrow t'_0 \rightarrow t'_n$ correspond à l'utilisation de la traduction structurelle des termes de preuve que nous allons présenter dans la suite de ce document. Cette traduction structurelle conduit à définir sur la représentation B les opérations suivant la structure de A.

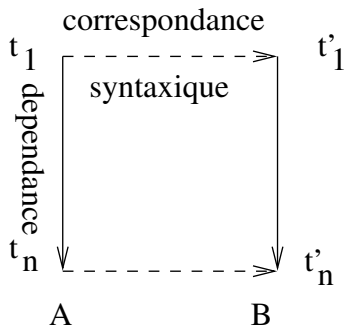

FIG. 5.2 – Mise en correspondance de deux développement formels. Les flèches verticales représentent les dépendances entre les théorèmes alors que les flèches horizontales représentent la correspondance syntaxique entre les énoncés des théorèmes dans les deux représentations.

5.2.4 Données affectées par un changement de représentations

Le changement de représentation peut se faire en remplaçant un type inductif par un autre, passant ainsi de la représentation unaire à la représentation binaire des entiers par exemple. Une fois la technologie bien maîtrisée dans ce cas, on s'intéressera au changement de représentation d'un prédicat défini inductivement. Finalement on pourra considérer les changements de représentations en présence de types dépendants.

Parallèlement à cette hiérarchisation des changements de représentations, des plus simples aux plus complexes, on peut imaginer de changer la représentation d'une fonction en conservant toute chose égale par ailleurs. Après tout, les fonctions sont des objets de première classe. On peut par exemple, vouloir remplacer la version récursive de l'addition sur les entiers de Peano par une version récursive terminale. De tels changements de représentation interviendront aussi si l'on transforme une définition par point-fixe en une définition équivalente basé sur une relation bien fondée.

5.3 Travaux reliés

5.3.1 Présentation universelle des données

Venanzio Capretta décrit dans [Cap99] une formalisation de la notion d'algèbre universelle en théorie des types. L'intérêt d'un tel développement est qu'il permet d'avoir un cadre générique pour la spécification de structures de données. En utilisant ce cadre, on peut facilement représenter les théories du premier ordre décrivant les entiers naturels et leurs opérations classiques sous forme d'une algèbre de termes (avec les symboles O, S, $\mathsf{plus}\ldots$). Il est ensuite envisageable d'exprimer dans ce même langage formel ce qu'est un théorème sur les entiers naturels (par un ensemble de règles de construction par exemple). A partir de là, il devient possible de construire deux interprétations de ces théorèmes dans les entiers de Peano et dans l'arithmétique binaire. On peut ensuite envisager d'établir un principe qui montre que tout théorème sur les entiers de Peano a un équivalent sur les entiers binaires.

Cette approche présente l'avantage d'être très formelle et générale, mais l'inconvénient d'être difficilement utilisable en pratique. Elle se rapproche des techniques de réflexion [Bou97] qui consistent à exprimer dans un système des propriétés du système lui-même. Ici, on raisonnerait sur les objets du système (les théorèmes sur les entiers naturels) en les modélisant à l'intérieur même du système

sous forme d'un type de données concret.

Nous n'avons pas retenu cette technique car le principe à établir demande un effort considérable pour décrire le cadre logique dans lui-même. De plus, le résultat d'incomplétude de Gödel montre que cet objectif ne peut être atteint que partiellement.

5.3.2 Cœrcions

Une manière de manipuler simultanément des représentations concrètes d'une même notion mathématique dans une théorie développée formellement consiste à utiliser des cœrcions implicites [Bar95, Saï97, Luo99]. Cette technique permet d'utiliser indifféremment des fonctions de type $A \to A$ ou de type $B \to B$ sur des objets de type A et B. Au moment du typage, le système de types va rajouter aux bons endroits les cœrcions entre B et A par exemple. Ainsi, dans le contexte $f : A \to A; x : B$, le terme $(f\ x)$ est transformé en $(f\ (c\ x))$ où $c : B \to A$ est la cœrcion de A vers B. La mise en œuvre des cœrcions fait intervenir un certain nombre de concepts difficiles dont les deux suivants.

- Il faut s'assurer de la cohérence du système avec cœrcions. Cela signifie que le graphe engendré par un ensemble de cœrcions par composition et instantiation doit avoir la propriété suivante : pour tout couple de types A et B, toutes les cœrcions de A vers B doivent être convertibles.
- Enfin les systèmes avec cœrcions ne prennent pas en compte les cœrcions aller-et-retour (*back-and-forth*) qui sont nécessaires pour passer d'une représentation à une autre des données et vice-versa.

Dans notre étude, nous ne nous trouvons pas dans la situation où les deux représentations doivent coexister. C'est pourquoi nous n'aurons pas besoin d'utiliser les cœrcions.

5.3.3 Transformations automatiques de preuves

Nous nous intéressons aux techniques de transformation automatique de preuves afin de réutiliser les preuves construites avec la représentation initiale des données pour établir les preuves correspondantes dans la nouvelle représentation. De nombreux travaux ont été effectués sur des sujets similaires aussi bien dans le cadre de la synthèse de programmes [Mad89, And94, Ric95] que dans celui des preuves par analogie [MW99] visant à réutiliser l'expérience acquise dans les preuves précédentes. D'autres travaux ont été menés sur le thème de la coopération et de la réutilisation dans un système d'aide à la preuve de démonstrations construites dans un autre [Den00, FH97].

5.3.3.1 Transformation de preuves et synthèse de programmes

Le paradigme *proofs-as-programs* consiste à extraire d'une preuve constructive de l'énoncé :

$$\forall x.P(x) \ \Rightarrow\ \exists y \mid Q(x, y)$$

un programme fonctionnel calculant y à partir de x. Plusieurs études ont été réalisées concernant la transformation ou l'optimisation de programmes par modification de la preuve à partir de laquelle ils sont synthétisés.

P. Madden [Mad89] propose de transformer automatiquement des programmes par la transformation automatique de la preuve de leur spécification. Il montre, en particulier, comment des transformations au niveau de la preuve permettent la spécialisation du programme extrait. P. Anderson [And94] présente un moyen d'encoder les transformations de preuve au sein du système Elf. Elle propose d'optimiser un programme en modifiant directement la structure de la preuve sous-jacente plutôt que d'utiliser les techniques standards (purement syntaxiques) d'optimisation. Elle montre,

entre autres, comment utiliser cette méthode pour transformer une définition de fonction récursive en une définition récursive-terminale.

J. Richardson [Ric95] utilise les transformations de preuve pour changer les structures de données mises en jeu dans la preuve et le programme qui en est extrait. Cela permet en disposant de structures de données sur lesquelles les opérations de base sont plus efficaces d'améliorer les performances des programmes extraits. Considérons par exemple un changement de représentation des entiers naturels : le passage des entiers de Peano aux entiers binaires. On se donne la définition des entiers de Peano et de l'addition sur ce type de données, la définition des entiers binaires et des fonctions de passage d'une représentation à l'autre. A partir de ces éléments, il est possible de construire automatiquement une preuve de la spécification

$$\forall x, y : \mathsf{bin}.\ \exists z : \mathsf{bin} \mid nat(z) = nat(x) + nat(y)$$

où *nat* est une fonction de traduction des entiers binaires vers les entiers de Peano. Le travail présenté dans [Ric95] permet d'obtenir une preuve de cette propriété à partir de laquelle il est possible de synthétiser un programme raisonnablement efficace de calcul de l'addition sur les entiers binaires. Ce résultat aurait pu être utilisé dans notre travail pour synthétiser les opérations de base sur les entiers binaires. Cependant, il ne permet pas d'obtenir l'algorithme le plus efficace de calcul de la somme de deux entiers binaires. Nous avons donc préféré considérer que la définition des opérations de base sur un nouveau type concret fait partie des données initiales du problème.

5.3.3.2 Preuves par analogie

Un autre sujet de recherche intéressant est l'utilisation du raisonnement par analogie dans les assistants de preuve. Il est en effet fréquent que des preuves mathématiques informelles contiennent des formules comme *"On démontrerait P de la même façon que l'on a établi Q"*.

On peut distinguer deux étapes dans la preuve par analogie :

- choisir, dans une base de données, un théorème Q sur lequel on va s'appuyer pour construire la démonstration du nouveau théorème P ; c'est le rédacteur de la preuve qui prend habituellement en charge cette partie ;
- adapter la preuve de Q pour que celle-ci soit une preuve de P ; c'est une tâche qui incombe au lecteur de la preuve.

Dans le cadre des systèmes d'aide à la démonstration de théorèmes, des travaux concernant l'automatisation de ces deux aspects du raisonnement par analogie ont été effectués. R. Curien propose dans [Cur95] une méthode pour rechercher dans une base de théorèmes un énoncé dont la preuve pourrait être réutilisée pour établir le nouvel énoncé considéré. Cette recherche se base sur la notion de filtrage d'ordre 2 sur l'énoncé du théorème dont on cherche à établir une preuve. Supposons que l'on cherche à établir une preuve de $\forall n, m : \mathsf{nat}.\ (\mathsf{mult}\ n\ m) = (\mathsf{mult}\ m\ n)$, on construit un motif dans lequel on peut avoir des variables qui représentent des objets du premier ordre et des fonctions sur ces objets (d'où le terme "2^e ordre"). Dans l'exemple précédent, le motif construit est le suivant $(f\ x\ y) = (f\ y\ x)$ avec f un symbole de fonction et x et y des variables du premier ordre. Le système cherche ensuite dans la base de données un énoncé qui est une instance de ce motif, par exemple $\forall n, m : \mathsf{nat}.\ (\mathsf{plus}\ n\ m) = (\mathsf{plus}\ m\ n)$.

E. Melis et J. Whittle proposent dans [MW99] une technique pour construire cette nouvelle preuve à partir du modèle. On commence par établir une correspondance entre les objets mis en jeu dans chacun des deux théorèmes, puis on rejoue en parallèle la construction des preuves de l'ancien et du nouveau théorème tant que le but (le théorème courant à établir) dans le théorème cible vérifie les critères autorisant l'application du même pas de preuve que dans l'original. Si ces

conditions ne sont pas vérifiées, on recourt soit à des heuristiques classiques résolvant le but, soit à la spéculation d'un lemme permettant l'établissement de la propriété recherchée. On tentera de prouver ce lemme séparément. La spéculation de lemme s'accompagne de l'utilisation d'un *disprover* qui, par la recherche de contre-exemples, vérifie si le lemme spéculé n'est pas trivialement faux.

Ces idées ont été implémentées avec succès au sein du démonstrateur de théorèmes CLAM. Ce système est basé sur des tactiques et des *proof-plans* (qui peut être considérés comme des super-tactiques permettant une large automatisation des preuves). Les calculs de recherche de preuve effectués par ces tactiques peuvent être arbitrairement compliqués. Dans notre approche, nous travaillons sur une description exacte de la preuve indépendamment de la façon dont elle a été construite. Le système Coq donne un moyen plus précis de contrôler la traduction des preuves. En effet, en cas d'échec de l'application d'une règle dans un *proof-plan*, il est difficile d'en déterminer les raisons et de résoudre le problème. Dans le cas d'un terme de preuve, les étapes de raisonnement considérées sont élémentaires et en cas d'échec, on peut plus facilement trouver quelle autre règle appliquer pour pouvoir poursuivre la traduction de la preuve.

5.3.3.3 Traduction d'un formalisme vers un autre

E. Denney [Den00] propose un mécanisme de traduction permettant de générer des scripts de preuve Coq à partir d'un enregistrement du déroulement d'une preuve dans le système HOL [GM93]. HOL (acronyme de *Higher-Order Logic*, logique d'ordre supérieur) est un système de preuves dans la tradition de LCF. Seuls les énoncés des théorèmes sont représentés dans un λ-calcul simplement typé. En particulier, il n'y a pas de représentation des preuves sous forme de λ-termes, celles-ci sont représentées sous forme d'une succession d'applications de règles d'inférence. Il n'existe par ailleurs pas de notion de convertibilité comme c'est le cas dans Coq.

Un mécanisme de traduction tel que celui décrit dans [Den00] permet la réutilisation des preuves développées en HOL pour construire de nouvelles théories dans Coq ; on évite ainsi de reformaliser les mêmes notions mathématiques dans des systèmes différents.
Le problème de la coopération entre les systèmes de preuves a déjà été étudié dans le cas des systèmes HOL et Nuprl [FH97]. A. Felty et D. Howe ont réalisé une formalisation hybride d'une preuve de normalisation de la logique combinatoire SK simplement typée en utilisant à la fois des bibliothèques de théorèmes dans les deux systèmes. De tels travaux ont pour objectif de montrer qu'il n'est pas nécessaire de redéfinir une théorie si elle est déjà formalisée dans un autre système. Elle peut, en effet, soit être utilisée directement, soit traduite systématiquement dans le formalisme désiré.

Lors d'une traduction comme celle proposée par Denney, on est amené à exprimer des notions primitives dans le formalisme de départ par des théorèmes dédiés "les imitant" dans le formalisme d'arrivée. C'est le cas pour les opérateurs de réécriture au niveau de l'équivalence propositionnelle dans HOL. Les étapes de réécriture sont remplacées par l'application d'un théorème Coq simulant un pas de réécriture. Cette technique de remplacement de manipulations primitives de termes par l'application de théorèmes sera réutilisée dans notre travail.

5.3.4 Réutilisation des preuves dans Coq

Nous présentons quelques travaux sur la réutilisation des preuves dans Coq. O. Pons a étudié des mécanismes de généralisation de théorèmes afin qu'ils puissent être réutilisés dans d'autres contextes. C. Dubois, J. Grandguillot et M. Jaume ont étudié comment réutiliser un développement formel sur les groupes pour construire une nouvelle théorie sur les groupes, qu'ils ont importée dans le système Foc [GF03].

5.3.4.1 Généralisation de théorèmes

O. Pons [Pon99, Pon00] propose un mécanisme de généralisation des théorèmes et de leurs démonstrations dans le but de les réinstancier et de les réutiliser dans d'autres contextes. A partir d'un théorème comme mult_permute, il dérive un théorème générique f_permute. La généralisation consiste à rajouter comme prémisses à l'énoncé de f_permute toutes les propriétés de base utilisées dans la preuve de f_permute, par exemple f_sym et f_assoc_l. Ce nouveau théorème peut alors être réinstancié par n'importe quelle opération op (à la place de f) du moment qu'elle vérifie les propriétés de base comme op_sym et op_assoc_l. Cette méthode permet la réutilisation de nombreuses preuves lors du développement d'une nouvelle théorie. Cependant elle présente un inconvénient. Lors de la phase d'abstraction, on abstrait une fonction uniquement par son type et ses propriétés logiques sans prendre en considération ses propriétés calculatoires. Un des principaux objectifs du travail que nous présentons dans cette thèse est justement de prendre en compte ces aspects calculatoires lors de la transformation de termes de preuve.

5.3.4.2 Réutilisation d'un développement sur les groupes

C. Dubois, J. Grandguillot et M. Jaume [DGJ03] ont étudié comment reprendre une théorie développée formellement pour l'intégrer dans un nouveau contexte. Ils ont travaillé à l'intégration d'une bibliothèque sur la théorie des groupes dans le système Foc [GF03]. Dans ce système, les structures sont paramétrées par une relation d'équivalence alors que l'étude sur les groupes dans le système Coq utilise l'égalité syntaxique. La première phase de leur travail consiste à remplacer l'égalité utilisée dans le développement Coq par une relation d'équivalence quelconque. Il faut ensuite généraliser les démonstrations effectuées dans le développement initial en les paramétrant par la relation d'équivalence et les propriétés de congruence qu'elles doivent vérifier. La deuxième phase consiste simplement à traduire le nouveau développement Coq obtenu vers le formalisme du projet Foc.

5.3.4.3 Réutilisation des preuves en théorie des types

Parallèlement à notre approche plutôt pragmatique, G. Barthe et O. Pons [BP01] ont étudié l'application des isomorphismes de types à la réutilisation des preuves dans les théories des types avec types dépendants. Leur travail est basé sur une interprétation calculatoire de certains isomorphismes de types en utilisant des cœrcions. Ils étudient entre autres, l'isomorphisme entre entiers de Peano et entiers binaires, ainsi que l'isomorphisme entre les enregistrements (records) associatifs à gauche et les enregistrements associatifs à droite. Ils proposent un formalisme logique dans lequel l'utilisateur peut manipuler les données en les considérant de plusieurs points de vue différents.

Chapitre 6

Du calcul vers le raisonnement logique

Dans Coq, les preuves formelles sont généralement construites par l'application de tactiques. Une fois terminées, elles sont stockées sous forme de termes de preuve. Pour des raisons de robustesse et de complétude, nous avons choisi de travailler au niveau des termes de preuves plutôt qu'avec les scripts de preuves. Dans ce chapitre, nous allons montrer comment transformer une preuve pour qu'elle ne dépende plus des propriétés calculatoires du type de données sur lequel on énonce une propriété. Les travaux présentés dans ce chapitre sont décrits dans les articles [MB00] et [Mag03].

6.1 Termes considérés

Conformément à l'usage en Théorie des Types, on considère un ensemble de sortes S avec trois éléments Prop, Set et Type. Un terme est un élément du langage T défini comme suit :

$$\begin{array}{rl} \mathcal{T} ::= & \lambda x : \mathcal{T}.\mathcal{T} \mid (\mathcal{T}\ \mathcal{T}) \mid \forall x : \mathcal{T}.\ \mathcal{T} \mid \mathcal{S} \mid x \mid \mathcal{C} \mid \mathcal{I} \mid \mathcal{CI} \\ & \mid \mathsf{let}\ x\ = \mathcal{T} : \mathcal{T}\ \mathsf{in}\ \mathcal{T} \\ & \mid \langle \mathcal{T} \rangle \mathsf{case}\ \mathcal{T}\ \mathsf{of}\ \{\mathcal{T}\} \end{array}$$

$\mathcal{C}$ représente l'ensemble des constantes du langage, $\mathcal{I}$ l'ensemble des types inductifs et $\mathcal{CI}$ l'ensemble des constructeurs de ces types inductifs. Ce langage correspond à l'ensemble des termes que l'on rencontre habituellement dans les termes de preuves. Nous avons délibérément ignoré la construction par point-fixe (fix), en effet celle-ci n'apparaît qu'exceptionnellement dans les termes de preuve. La plupart du temps la récursion structurelle est cachée derrière la définition d'une constante ; par exemple l'opérateur de récursion nat_rec. Néanmoins nous proposerons dans le chapitre suivant, une manière systématique de transformer une fonction récursive structurelle (utilisant le construction fix) en une définition par ordre bien-fondé.

Notre travail consiste à rendre explicites toutes les étapes de raisonnement utilisant les propriétés calculatoires des types inductifs et des fonctions structurellement récursives. Dans un premier temps, nous allons *cacher* les constructions de filtrage ($\langle \mathcal{T} \rangle \mathsf{case}\ \mathcal{T}\ \mathsf{of}\ \{\mathcal{T}\}$) sur les données définies inductivement en les remplaçant par l'application d'un opérateur de récursion structurelle bien choisi. Une fois cette phase achevée, nous verrons comment représenter les fonctions récursives structurelles comme des objets logiques sans propriétés calculatoires. Enfin, nous montrerons comment rendre explicites toutes les étapes calculatoires utilisées dans la construction d'un terme de preuve. Au bout du compte, nous aurons développé un mécanisme pour transformer les termes de preuve en de nouveaux qui ne dépendront plus implicitement mais explicitement de l'implantation concrète et donc des propriétés calculatoires des fonctions.

6.2 Élimination des constructions de filtrage explicite

Les constructions de filtrage présentes dans les termes de preuve peuvent avoir deux origines différentes :

- l'utilisateur peut avoir eu recours à la tactique Case pour résoudre un but en faisant de l'analyse par cas, plutôt que d'utiliser le principe d'induction correspondant ;
- il est plus probable que l'apparition de cette construction de filtrage dans le terme de preuve soit due à l'application de la tactique Inversion [CT96] par l'utilisateur.

Dans cette section, nous prenons comme exemple la propriété $\forall n : \mathsf{nat}.\ n \leq O \rightarrow n = O$ qui se démontre par inversion sur le prédicat inductif le. Cette preuve contient des expressions de filtrage structurel dû à cette inversion.

Ce théorème est démontré par le script suivant :

```
Lemma example : (n:nat)(le n O)->n=O.
Intros n H ; Inversion H ; Auto.
Qed.
```

et le terme de preuve construit est donné à la figure 6.1.

$$
\begin{array}{l}
\lambda n : \mathsf{nat}.\lambda H : (\mathsf{le}\ n\ \mathsf{O}). \\
\quad \mathit{let}\ H_0\ =\ \langle \lambda p : \mathsf{nat}. p = \mathsf{O} \rightarrow n = \mathsf{O} \rangle \\
\qquad\qquad \mathit{Cases}\ H\ \mathit{of} \\
\qquad\qquad\quad \mathsf{le\_n} \Rightarrow\ \lambda H_0 : n = \mathsf{O}. \\
\qquad\qquad\qquad (\mathsf{eq\_ind}\ \mathsf{nat}\ \mathsf{O}\ \lambda n : \mathsf{nat}. n = \mathsf{O}\ \ (\mathsf{refl\_equal}\ \mathsf{nat}\ \mathsf{O}) \\
\qquad\qquad\qquad\qquad\quad n\ (\mathsf{sym\_eq}\ \mathsf{nat}\ n\ \mathsf{O}\ H_0)) \\
\qquad\qquad\ |\ (\mathsf{le\_S}\ m\ H_0) \Rightarrow\ \lambda H_1 : (\mathsf{S}\ m) = \mathsf{O}. \\
\qquad\qquad\qquad (\mathit{let}\ H_2\ =\ (\mathsf{eq\_ind}\ \mathsf{nat}\ (\mathsf{S}\ m) \\
\qquad\qquad\qquad\qquad\quad \lambda e : \mathsf{nat}. \langle \mathsf{Prop} \rangle \mathit{Cases}\ e\ \mathit{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad O \Rightarrow \mathsf{False} \\
\qquad\qquad\qquad\qquad\qquad \mathit{end}\ \mathsf{I}\ \mathsf{O}\ H_1) \\
\qquad\qquad\qquad \mathit{in}\ (\mathsf{False\_ind}\ (\mathsf{le}\ n\ m) \rightarrow n = \mathsf{O}\ H_2)\ H_0) \\
\qquad\qquad \mathit{end} \\
\quad \mathit{in}\ (H_0\ (\mathsf{refl\_equal}\ \mathsf{nat}\ \mathsf{O})).
\end{array}
$$

FIG. 6.1 – Un terme prouvant le lemme example

Dans cette section, nous allons montrer comment remplacer une construction de filtrage par l'application d'un principe d'induction bien choisi. Nous considérons le type inductif suivant

$$
\begin{array}{rl}
\mathsf{Inductive}\ T\ :\ \mathsf{S}\ :=\ & |\ c1 : \forall t_{1,1} : T_{1,1} \ldots\ \forall t_{1,i_1} : T_{1,i_1}.\ T \\
& |\ \ldots \\
& |\ c_k : \forall t_{k,1} : T_{k,1} \ldots\ \forall t_{k,i_k} : T_{k,i_k}.\ T \\
& |\ \ldots \\
& |\ c_n : \forall t_{n,1} : T_{n,1} \ldots\ \forall t_{n,i_n} : T_{n,i_n}.\ T.
\end{array}
$$

6.2.1 Algorithme d'élimination des constructions de filtrage

Considérons une preuve p d'une propriété quelconque des éléments de T. L'algorithme effectue une analyse récursive structurelle du terme p. Pour la plupart des termes (les constantes, les

$$
\begin{array}{llll}
\langle P\rangle & \mathsf{Cases} & t\ \mathsf{of} & \\
 & & \mid (c_1\ t_{1,1} \ldots\ t_{1,i_1}) \Rightarrow r_1 & \\
 & & \mid \ldots & \\
 & & \mid (c_k\ t_{k,1} \ldots\ t_{k,i_k}) \Rightarrow r_k & \equiv \\
 & & \mid \ldots & \\
 & & \mid (c_n\ t_{n,1} \ldots\ t_{n,i_n}) \Rightarrow r_n & \\
 & \mathsf{end} & &
\end{array}
\qquad
\begin{array}{lll}
\langle P\rangle & \mathsf{Case} & t\ \mathsf{of} \\
 & & \lambda t_{1,1} : T_{1,1} \ldots\ \lambda t_{1,i_1} : T_{1,i_1}.r_1 \\
 & & \ldots \\
 & & \lambda t_{k,1} : T_{k,1} \ldots\ \lambda t_{k,i_k} : T_{k,i_k}.r_k \\
 & & \ldots \\
 & & \lambda t_{n,1} : T_{n,1} \ldots\ \lambda t_{n,i_n} : T_{n,i_n}.r_n \\
 & \mathsf{end} &
\end{array}
$$

FIG. 6.2 – Un exemple de construction de filtrage que notre algorithme doit savoir traiter

applications, les constructions let-in, les abstractions...), l'algorithme s'appelle récursivement sur les sous-termes de l'expression considérée, s'il y en a ; sinon il retourne simplement le terme. Le cas intéressant est celui du traitement de l'analyse par cas. La figure 6.2 donne un exemple d'expression de filtrage que l'on peut être amené à traiter. La partie gauche correspond à la notation habituelle dans le système Coq alors que la partie droite utilise une notation beaucoup plus proche de l'implantation réelle du traitement par cas dans Coq. En présence d'une construction de filtrage, l'algorithme procède de la façon suivante :

1. On commence par calculer le type de t. Supposons que t soit de type T, i.e. $t : T$. Comme nous l'avons vu dans le chapitre 5, il existe six opérateurs de récursion structurelle pour T selon qu'il soit dépendant ou non et que la sorte des objets construits au moyen de ce principe soit Prop, Set ou Type. Ce sont T_rec, T_ind, et T_rect ainsi que T_min_rec, T_min_ind et T_min_rect.
2. Cette étape va permettre de déterminer lequel des six principes donnés ci-dessus va être utilisé pour simuler le traitement par cas. Comme le montre la figure 6.2, P est le prédicat d'élimination associé à la construction de filtrage [Coq02, chap. 14]. Les types des branches de la construction de filtrage sont des instances du prédicat d'élimination. La forme de P permet de déterminer si nous sommes en présence d'un filtrage dépendant ou non. Si le type du terme sur lequel on fait le traitement par cas (dans notre exemple H : (le n O)) apparaît en dernière λ-abstraction, alors le filtrage est dépendant. Sinon, le filtrage est non-dépendant. Dans notre cas, le filtrage est non dépendant puisqu'il n'y a pas de λ-abstraction sur (le n O) dans P. Une fois que l'on a déterminé si le filtrage est dépendant ou non, il reste à déterminer le type des objets construits par cette expression. Pour cela, on calcule la sorte de la valeur retournée par P. Par exemple, si on a $P \equiv\ \lambda p : \mathsf{nat}.\ p = \mathsf{O} \rightarrow n = \mathsf{O}$, alors la sorte retournée est Prop. Par conséquent, on utilisera le principe le_ind.
3. Une fois que l'on a identifié l'opérateur de récursion adéquat (appelons-le induction_for_T), on construit une instance du principe d'analyse par cas correspondant (appelons-le case_for_T). Par exemple, on dérive de nat_rec un opérateur (non-récursif) d'analyse par cas nat_case_rec qui a le type suivant :

$$\forall P : \mathsf{nat} \rightarrow \mathsf{Set}.\ (P\ \mathsf{O}) \rightarrow (\forall n : \mathsf{nat}.\ (P\ (\mathsf{S}\ n))) \rightarrow \forall n : \mathsf{nat}.\ (P\ n).$$

   Le calcul ci-dessus est indépendant du reste des calculs effectués par l'algorithme.
4. L'algorithme calcule récursivement de nouveaux termes $r'_1, \ldots, r'_n$ (ne contenant plus aucune construction de filtrage structurel) pour tous les sous-termes $r_1, \ldots, r_n$ de cette expression de filtrage.
5. Il reste maintenant à construire une application dont la tête est case_for_T et qui imite le traitement par cas structurel en utilisant cet opérateur de traitement par cas défini comme

une constante. Finalement, l'algorithme retourne le terme suivant en remplacement de la construction de filtrage :

$$\begin{array}{ll}(\mathsf{case\_for\_T}\ P & \lambda t_{1,1} : T_{1,1} \ldots\ \lambda t_{1,i_1} : T_{1,i_1}.r'_1 \\ & \ldots \\ & \lambda t_{n,1} : T_{n,1} \ldots\ \lambda t_{n,i_n} : T_{n,i_n}.r'_n \\ & t)\end{array}$$

En résumé, cette transformation supprime les constructions de filtrage explicites dans les termes de preuve et les dissimule par l'utilisation d'opérateurs de traitement par cas définis comme des constantes. Cela permet d'avoir un moyen de décrire les constructions d'analyse par cas sans connaître la représentation inductive effective du type de données.

6.2.2 Exemple

A partir du terme de preuve donné à la figure 6.1, l'algorithme produit un nouveau terme de preuve *équivalent*, mais sans aucune construction de filtrage. Ce terme est donné à la figure 6.3.

L'expression *Cases H of ...* est remplacée par l'application du principe de traitement par cas le_case_ind. De plus, *Cases e of ...* est remplacée par l'application de l'opérateur minimal (i.e. non-dépendant) nat_case_rect_min qui a le type suivant : $\forall P : \mathsf{Type}.\ P \to (\mathsf{nat} \to P) \to \mathsf{nat} \to P$.

$$\begin{array}{l}\lambda n : \mathsf{nat}.\lambda H : (\mathsf{le}\ n\ \mathsf{O}). \\ \quad let\ H_0\ = (\mathsf{le\_case\_ind}\ n\ \lambda p : \mathsf{nat}.p = \mathsf{O} \to n = \mathsf{O} \\ \qquad\qquad \lambda H_0 : n = \mathsf{O}. \\ \qquad\qquad\quad (\mathsf{eq\_ind}\ \mathsf{nat}\ \mathsf{O}\ \lambda n : \mathsf{nat}.n = \mathsf{O}\ (\mathsf{refl\_equal}\ \mathsf{nat}\ \mathsf{O}) \\ \qquad\qquad\qquad\quad n\ (\mathsf{sym\_eq}\ \mathsf{nat}\ n\ \mathsf{O}\ H_0)) \\ \qquad\qquad \lambda m : \mathsf{nat}.\lambda H_0 : (\mathsf{le}\ n\ m).\lambda H_1 : (\mathsf{S}\ m) = \mathsf{O}. \\ \qquad\qquad\quad (let\ H_2\ =\ (\mathsf{eq\_ind}\ \mathsf{nat}\ (\mathsf{S}\ m) \\ \qquad\qquad\qquad\qquad\quad \lambda e : \mathsf{nat}. \\ \qquad\qquad\qquad\qquad\quad (\mathsf{nat\_case\_rect\_min}\ \mathsf{Prop}\ \mathsf{False}\ \lambda\_ : \mathsf{nat}.\mathsf{True}\ e) \\ \qquad\qquad\qquad\qquad\quad \mathsf{I}\ \mathsf{O}\ H_1) \\ \qquad\qquad\qquad\ in\ (\mathsf{False\_ind}\ (\mathsf{le}\ n\ m) \to n = \mathsf{O}\ H_2)\ H_0) \\ \qquad\qquad \mathsf{O}\ \mathsf{H}) \\ \quad in\ (H_0\ (\mathsf{refl\_equal}\ \mathsf{nat}\ \mathsf{O})).\end{array}$$

FIG. 6.3 – Le terme de preuve sans constructions de filtrage après l'application de l'algorithme

6.3 Propriétés caractéristiques des fonctions

Les fonctions définies par récursion structurelle ont des règles de réduction qui leur sont associées. Considérons l'exemple de la fonction plus d'addition de deux entiers de Peano :

```
Fixpoint plus [n:nat] : nat -> nat :=
[m:nat] Cases n of O     => m
                | (S p) => (S (plus p m))
        end.
```

Ls termes (plus O n) et n sont intentionnellement égaux ($\beta\delta\iota$-équivalents), de même pour les termes (plus (S p) n) et (S (plus p n)). En effet, les règles de réduction associées à la définition de plus sont les suivantes :

$$(\mathsf{plus}\ \mathsf{O}\ n) \xrightarrow{\iota} n \qquad (\mathsf{plus}\ (\mathsf{S}\ n)\ m) \xrightarrow{\iota} (\mathsf{S}\ (\mathsf{plus}\ n\ m))$$

6.3.1 Spécification des propriétés

Les propriétés de l'addition sur les entiers de Peano peuvent être axiomatisées de façon complètement logique. On peut caractériser cette fonction par les équation suivantes :

```
Axiom plus_O_n : (n:nat)(plus O n)=n.
Axiom plus_Sn_m : (n,m:nat)(plus (S n) m)=(S (plus n m)).
```

Ces équations sont démontrables par simple réflexivité de l'égalité puisque leurs membres gauche et droit sont convertibles. Remplacer ces équations par des propriétés équivalentes pour une nouvelle représentation sera possible alors que l'on ne peut pas transporter la convertibilité d'une représentation à l'autre. D'un point de vue plus technique, ces équations sont ajoutées dans une base de réécriture (appelée simplification) qui servira plus tard pour prouver les conjectures engendrées par l'algorithme d'extraction des étapes de raisonnement par le calcul dans les termes de preuves.

6.3.2 Opérateurs de récursion et réduction

Les opérateurs de récursion peuvent être vus comme des fonctions comme les autres. Par exemple l'opérateur de récursion dépendant nat_rec qui a le type suivant

$$\forall P : \mathsf{nat} \to \mathsf{Set}.\ (P\ \mathsf{O}) \to (\forall n : \mathsf{nat}.\ (P\ n) \to (P\ (\mathsf{S}\ n)) \to \forall n : \mathsf{nat}.\ (P\ n)$$

est simplement la fonction prenant un certain nombre d'arguments, parmi lesquels n et qui retourne un élément appartenant au type dépendant $(P\ n)$. Il peut être décrit par les équations suivantes :

$$(\mathsf{nat\_rec}\ P\ v0\ vr\ \mathsf{O}) = v0$$
$$\forall n : \mathsf{nat}.\ (\mathsf{nat\_rec}\ P\ v0\ vr\ (\mathsf{S}\ n)) = (vr\ n\ (\mathsf{nat\_rec}\ P\ v0\ vr\ n))$$

qui simulent la ι-réduction.

Une fois que l'on a formulé ces définitions et leurs règles de réduction sous une forme purement logique, on peut étudier comment rendre explicite les étapes de calcul utilisées dans un terme de preuve. C'est ce que nous allons voir dans la section suivante.

6.4 Extraction des étapes de calcul implicite des termes de preuves

Dans cette partie, nous étudions comme rendre un statut logique à toutes les étapes de raisonnement cachées dans du calcul. Les règles de réduction du calcul des constructions inductives permettent d'identifier deux termes syntaxiquement différents. Le but de la transformation proposée est de différencier deux termes convertibles modulo $\beta\delta\iota$-réduction, mais qui ne le sont pas modulo $\beta\delta$-réduction.

6.4.1 Un exemple

Tout au long de cette section, nous considérons le théorème plus_n_O à titre d'exemple :

$$\forall n \in \mathsf{nat} \quad n = (\mathsf{plus}\ n\ \mathsf{O}) \tag{6.1}$$

Le λ-terme suivant est une preuve de cette propriété :

$$\begin{aligned} \lambda n : \mathsf{nat}.(\mathsf{nat\_ind} \quad & (\lambda n0 : \mathsf{nat}.n0 = (\mathsf{plus}\ n0\ \mathsf{O})) \\ & (\mathsf{refl\_equal\ nat\ O}) \\ & \lambda n0 : \mathsf{nat}; H : (n0 = (\mathsf{plus}\ n0\ \mathsf{O})). \\ & \quad (\mathsf{f\_equal\ nat\ nat\ S}\ n0\ (\mathsf{plus}\ n0\ \mathsf{O})\ H)\ n) \end{aligned}$$

Ce théorème est démontré par récurrence sur n, en utilisant le principe d'induction nat_ind. Le terme (refl_equal nat O) est une preuve que O = O. Cependant, grâce au mécanisme de conversion, c'est aussi une preuve de O = (plus O O). De même,

$$\lambda n0 : \mathsf{nat}; H : (n0 = (\mathsf{plus}\ n0\ \mathsf{O})).\ (\mathsf{f\_equal\ nat\ nat\ S}\ n0\ (\mathsf{plus}\ n0\ \mathsf{O})\ H)$$

est une preuve de

$$\forall n0 : \mathsf{nat}.\ n0 = (\mathsf{plus}\ n0\ \mathsf{O}) \Rightarrow (\mathsf{S}\ n0) = (\mathsf{S}\ (\mathsf{plus}\ n0\ \mathsf{O}))$$

C'est aussi une preuve de $\forall n0 : \mathsf{nat}.\ n0 = (\mathsf{plus}\ n0\ \mathsf{O}) \Rightarrow (\mathsf{S}\ n0) = (\mathsf{plus}\ (\mathsf{S}\ n0)\ \mathsf{O})$ qui correspond au type attendu pour le troisième argument du principe d'induction nat_ind dans la preuve ci-dessus.

6.4.2 Notions de type attendu et type proposé

Afin de pouvoir rendre explicite les étapes de calcul implicites dans un terme de preuve, il faut tout d'abord les repérer. Pour cela, nous utilisons une technique proche de celle proposée par Y. Coscoy [Cos00] dans son travail sur la traduction en langage naturel des termes de preuve. Cette technique est également très proche de celles proposées par L. Magnusson [Mag95] pour typer des expressions avec des types dépendants dans le système d'aide à la preuve ALF. Nous avons besoin de déterminer les positions dans le terme de preuve où les types proposé et attendu sont différents. Nous nous intéressons plus particulièrement aux types de données concrets et à l'égalité de Leibniz telle qu'elle est fournie dans le système Coq.

L'algorithme que nous allons présenter dans le paragraphe suivant utilise les notions de types attendu et proposé pour un sous-terme d'un terme plus grand. Considérons un terme t, son type attendu T_A est le type que doit avoir t pour que le terme total (auquel appartient t) soit bien typé. Le type proposé T_P est, quant à lui, le type inféré par le système pour le terme t. A titre d'illustration, considérons l'application suivante $(t_1\ t_2)$. Si les types inférés par le système sont :

$$t_1\ :\ T\ \rightarrow\ U \qquad \& \qquad t_2\ :\ T'$$

Le type attendu pour t_2 est alors l'argument T du type fonctionnel $T \rightarrow U$, alors que son type proposé est T'. L'application $(t_1\ t_2)$ est bien typée si les types T et T' sont convertibles. Quand on utilise le paradigme *proofs-as-terms* (aussi connu sous le nom *formulas-as-types*), on voit que le type attendu correspond à une obligation de preuve, il exprime la propriété que l'on cherche à établir. Le type proposé correspond, quant à lui, à un constat de preuve.

6.4.3 Une première solution

Nous rappelons ici que la motivation principale de ce travail est de proposer des moyens de transférer une démonstration faite en utilisant une certaine représentation concrète d'une des données (sous la forme d'un type inductif) en une démonstration d'une propriété équivalente, exprimée en utilisant une autre représentation. Dans ce cadre, notre première approche consiste à transformer syntaxiquement un énoncé et sa preuve (en suivant la table d'association relative au changement de représentation considéré). Cette traduction conduit à un terme qui peut être mal typé. Il faut alors l'analyser et le transformer en un terme bien typé. Dans [MB01], nous décrivons un tel algorithme. Il s'agit d'une simple variante de l'algorithme que nous allons présenter dans la suite de ce document.

Par exemple si t est une preuve de T, mais pas de T' qui est le type attendu pour t, on construit une implication entre types proposé T et attendu T' dès lors que ces types ne coïncident plus modulo $\beta\delta\iota$-réduction dans la nouvelle représentation.

L'avantage de cette approche est qu'elle permet de n'ajouter des conjectures reliant les types proposé et attendu que quand cela est nécessaire dans la nouvelle représentation. Ainsi on n'augmentera pas inutilement la taille des termes de preuve. Cependant, si l'on veut de nouveau changer de représentation concrète, on devra recommencer toute la transformation. C'est pourquoi nous proposons dans la suite une méthode qui permet de conserver un terme de preuve correct (c'est-à-dire dont le type peut être vérifié par le vérificateur de types de Coq) au cours de la transformation.

6.4.4 Une solution plus élégante

L'algorithme que nous proposons [MB00] prend en entrée un terme de preuve et retourne un nouveau terme de preuve accompagné d'une série de conjectures reliant les types attendu et proposé quand c'est nécessaire dans la représentation initiale. L'algorithme parcourt le terme et détermine de manière préventive les endroits où type attendu et proposé ne coïncident pas. Ainsi, on aura un terme bien typé d'un bout à l'autre de la chaîne de transformation des termes de preuve.

6.4.4.1 Algorithme proposé

L'algorithme que nous proposons fonctionne sur les termes bien typés. Il prend en entrée un terme bien typé et retourne un terme bien typé dans lequel toutes les étapes de raisonnement implicite ont été mises en évidence. En plus de ce terme bien typé, il retourne un certain nombre de conjectures (une série de formules logiques reliant les types attendu T_A et proposé T_P partout où ceux-ci ne coïncident pas modulo $\beta\delta$-réduction) ; ces conjectures sont les étapes de raisonnement par le calcul rendu explicite par l'algorithme. Ces conjectures seront prouvées plus tard.

Le langage de termes que nous considérons est le suivant :

$$\begin{aligned}\mathcal{T} ::= \quad & \lambda x : \mathcal{T}.\mathcal{T} \mid (\mathcal{T}\ \mathcal{T}) \mid \forall x : \mathcal{T}.\ \mathcal{T} \mid \mathcal{S} \mid x \mid \mathcal{C} \mid \mathcal{I} \mid \mathcal{CI} \\ & \mid \mathsf{let}\ x\ = \mathcal{T} : \mathcal{T}\ \mathsf{in}\ \mathcal{T}\end{aligned}$$

C'est le langage donné au début du chapitre auquel on a retiré les constructions de traitement par cas. En effet, cette construction a été éliminée des termes de preuves par la méthode présentée plus haut dans ce chapitre.

Concrètement, l'algorithme prend en arguments un énoncé T, une preuve t de T, et un contexte C. Au préalable, on met le terme t en forme η-longue. Par exemple, si t a le type $\forall x : A.\ (B\ x)$ et si t n'est pas une abstraction, alors on remplace t par $\lambda x : A.\ (t\ x)$. L'algorithme fonctionne par récursion structurelle sur t et retourne un nouveau terme ainsi qu'une liste de conjectures.

Nous donnons maintenant le comportement de l'algorithme sur chacune des constructions du langage :

- **Variables, constantes, définitions inductives, constructeurs.**
  Toutes ces constructions sont inchangées et sont simplement retournées par l'algorithme.
- **Cas de l'abstraction.** t est de la forme $\lambda x : A.\ b$
  Par hypothèse, le terme t est bien typé. Cela garantit que T peut être réduit (par le calcul de sa forme normale de tête faible) sous la forme d'un produit $\forall x : A'.B$. De plus, on sait que les types A et A' sont convertibles modulo $\beta\delta\iota$-réduction. On choisit d'ajouter la liaison $x : A'$ dans le contexte C. On appelle ensuite récursivement l'algorithme sur de nouvelles entrées : b à la place de t, B à la place de T, et $x : A'\ ;\ C$ à la place de C.
- **Cas de l'application.** $t \equiv (h\ u_1 \dots\ u_n)$
  Ici, nous ne connaissons que le type attendu pour l'application de h à tous ses arguments $u_1\ \dots\ u_n$. Il n'y a pas de type attendu pour la tête h. Il est par contre possible de calculer son type proposé T_P. C'est obligatoirement un produit puisque le terme donné en entrée est bien typé. A partir de ce type, on peut inférer le type attendu TE_{u_1} pour le premier argument u_1. On le compare ensuite avec le type proposé TP_{u1} pour le terme u'_1 retourné par l'appel récursif à l'algorithme avec u_1 et TE_{u_1}. Si les types TE_{u_1} et TP_{u_1} sont convertibles par $\beta\delta$ conversion, l'algorithme conserve le terme intermédiaire $v_1 = u'_1$. Dans le cas contraire, l'algorithme construit une conjecture c_1 qui a le type $\forall x_1 : t_1, \dots x_k : t_k.TP_{u_1} \rightarrow TE_{u_1}$. Les x_i sont les variables qui apparaissent libres dans TP_{u_1} et TE_{u_1}. On trouve leurs types dans le contexte C. L'algorithme conserve comme résultat intermédiaire le terme $v_1 = (c_1\ x_1 \cdots\ x_k\ u'_1)$. Le même processus est répété pour chacun des arguments u_i de l'application. A chaque fois, l'algorithme retourne un nouveau terme v_i et parfois une nouvelle conjecture c_i. Finalement, l'algorithme construit un terme $v = (h\ v_1 \dots v_n)$. Si ce terme a pour type proposé T, alors l'algorithme retourne simplement le terme v ; sinon l'algorithme construit une conjecture supplémentaire c exprimant que $\forall x_1 : t_1, \dots x_k : t_k.T' \rightarrow T$ et retourne le terme $(c\ x_1 \cdots\ x_k\ (h\ v_1 \cdots v_n))$ et une liste de conjectures contenant c ainsi que la liste de toutes les conjectures construites au cours du parcours du terme t.
- **Cas du let-in.** $t \equiv (\mathsf{let}\ x : t\ =\ e\ \mathsf{in}\ b)$
  On analyse récursivement le terme e qui a pour type attendu t, cela produit un nouveau terme e'. On calcule ensuite, à partir de b, de son type attendu (qui est le type attendu de l'expression totale) et d'un nouveau contexte où l'on a ajouté la liasion $x : t$ un nouveau terme b' dans lequel les étapes de conversions sont rendues explicites. Il ne reste plus qu'à tester si le type de $(\mathsf{let}\ x : t\ =\ e'\ \mathsf{in}\ b')$ est $\beta\delta$-convertible avec le type attendu pour cette expression. Si ce n'est pas le cas, le processus de génération d'une conjecture décrit dans le cas de l'application est amorcé.

6.4.4.2 Preuve des conjectures générées

Dans le paragraphe précédent, nous venons de décrire comment rendre explicites les étapes de calcul utilisant les règles de ι-réduction. A partir d'un terme de preuve, on construit un nouveau terme de preuves ainsi qu'un certain nombre de conjectures qui correspondent aux étapes de raisonnement effectuées par ι-réduction. Dans la représentation inductive initiale des données, ces conjectures sont de simples conséquences des règles de convertibilité. Dans la nouvelle représentation, ces conjectures doivent être démontrées. Elles peuvent l'être par réécriture en utilisant les équations permettant de représenter les fonctions. Nous avons vu plus tôt dans ce chapitre que les équations caractéristiques des fonctions ont toutes été regroupées dans une base de réécriture (simplification) ;

ceci dans le but de simplifier les démonstrations de ces conjectures. Par exemple, la conjecture H_b générée lors du traitement de la preuve de plus_n_O peut être prouvée automatiquement par l'application de cette suite de tactiques :

```
Repeat (Intro; AutoRewrite [simplification]);Auto.
```

L'énoncé (i.e. le type) de H_b est $\forall n : \mathsf{nat}.\ n = (\mathsf{S}\ (\mathsf{Aplus}\ n\ \mathsf{O}) \rightarrow (\mathsf{S}\ n) = (\mathsf{Aplus}\ (\mathsf{S}\ n)\ \mathsf{O})$. On voit bien qu'il suffit de réécrire la conclusion en utilisant la deuxième équation (à savoir plus_Sn_m) permettant de représenter de façon abstraite la fonction plus (voir 6.3).

6.4.4.3 Comportement sur des exemples

Un premier exemple L'exécution de l'algorithme sur l'exemple plus_n_O présenté au début de cette section donne le résultat suivant :

$$\begin{array}{l}\lambda n : \mathsf{nat}.\ (\ \mathsf{nat\_ind}\\ \qquad (\lambda n0 : \mathsf{nat}.n0 = (\mathsf{plus}\ n0\ \mathsf{O}))\\ \qquad (Ha\ (\mathsf{refl\_equal\ nat\ O}))\\ \qquad \lambda n0 : \mathsf{nat}; H : (n0 = (\mathsf{plus}\ n0\ \mathsf{O})).\\ \qquad\quad (Hb\ n0\ (\mathsf{f\_equal\ nat\ nat\ S}\ n0\ (\mathsf{plus}\ n0\ \mathsf{O})\ H))\ n)\end{array}$$

Deux conjectures Ha et Hb ont été générées au cours de l'exécution de l'algorithme afin de relier les types proposé et attendu de chacune des branches de l'application du principe de récursion nat_ind.

$$\begin{array}{c} Ha : \mathsf{O}\ =\ \mathsf{O}\ \Rightarrow\ \mathsf{O}\ =\ (\mathsf{plus\ O\ O})\\ Hb : \forall n : \mathsf{nat}.\ (\mathsf{S}\ n)\ =\ (\mathsf{S}\ (\mathsf{plus}\ n\ \mathsf{O}))\ \Rightarrow\ (\mathsf{S}\ n)\ =\ (\mathsf{plus}\ (\mathsf{S}\ n)\ \mathsf{O})\end{array}$$

Bien évidemment, ces deux conjectures peuvent être prouvées immédiatement en introduisant les prémisses et en utilisant la réflexivité de l'égalité de Leibniz. Un tel raisonnement est valable parce que les deux termes de part et d'autre de l'égalité sont convertibles modulo $\beta\delta\iota$-réduction.

Un autre exemple Reprenons l'exemple utilisé dans la partie traitant l'élimination des constructions de filtrage. Le terme donné à la figure 6.3 a été obtenu après l'étape d'élimination des constructions d'analyse par cas. Il faut encore le transformer afin d'en extraire les étapes de calcul implicites. L'analyse du terme conduit à la génération de deux conjectures example_rr1 et example_rr2. Le terme example_rr1 doit être une preuve que

$$\forall m : \mathsf{nat}.\ \mathsf{True} \rightarrow (\mathsf{nat\_case\_rect\_min\ Prop\ False}\ \lambda\_ : \mathsf{nat.True}\ (\mathsf{S}\ m)) \tag{6.2}$$

et example_rr2 une preuve que

$$(\mathsf{nat\_case\_rect\_min\ Prop\ False}\ \lambda\_ : \mathsf{nat.True\ O}) \rightarrow \mathsf{False} \tag{6.3}$$

Les deux énoncés (6.2) et (6.3) sont démontrés automatiquement par réécriture en utilisant les règles de réduction associées à la constante nat_case_rect_min.

$\lambda n : \mathsf{nat}.\lambda H : (\mathsf{le}\ n\ \mathsf{O}).$
$\quad let\ H_0\ = (\mathsf{le\_case\_ind}\ n\ \lambda p : \mathsf{nat}.p = \mathsf{O} \rightarrow n = \mathsf{O}$
$\qquad\qquad \lambda H_0 : n = \mathsf{O}.$
$\qquad\quad (\mathsf{eq\_ind}\ \mathsf{nat}\ \mathsf{O}\ \lambda n : \mathsf{nat}.n = \mathsf{O}\ (\mathsf{refl\_equal}\ \mathsf{nat}\ \mathsf{O})$
$\qquad\qquad\quad n\ (\mathsf{sym\_eq}\ \mathsf{nat}\ n\ \mathsf{O}\ H_0))$
$\qquad\quad \lambda m : \mathsf{nat}.\lambda H_0 : (\mathsf{le}\ n\ m).\lambda H_1 : (\mathsf{S}\ m) = \mathsf{O}.$
$\qquad\quad (let\ H_2\ =\ (\mathbf{example\_rr2}\ (\mathsf{eq\_ind}\ \mathsf{nat}\ (\mathsf{S}\ m)$
$\qquad\qquad\qquad\qquad \lambda e : \mathsf{nat}.$
$\qquad\qquad\qquad\qquad (\mathsf{nat\_case\_rect\_min}\ \mathsf{Prop}\ \mathsf{False}\ \lambda\_ : \mathsf{nat}.\mathsf{True}\ e)$
$\qquad\qquad\qquad\qquad (\mathbf{example\_rr1}\ m\ \mathsf{I})\ \mathsf{O}\ H_1))$
$\qquad\quad\ in\ (\mathsf{False\_ind}\ (\mathsf{le}\ n\ m) \rightarrow n = \mathsf{O}\ H_2)\ H_0)$
$\qquad\qquad \mathsf{O}\ \mathsf{H})$
$in\ (H_0\ (\mathsf{refl\_equal}\ \mathsf{nat}\ \mathsf{O})).$

FIG. 6.4 – Le terme de preuve du théorème example après traitement par les deux algorithmes

6.5 Traitement des définitions

6.5.1 Cas des fonctions

Lors du passage vers une nouvelle structure de données, on veut définir les nouvelles opérations sur le type de données de manière efficace en tirant profit de la structure de la nouvelle représentation. Dans ce cas, les définitions des opérations sur la nouvelle représentation ne sont pas des traductions syntaxiques directes de celles données dans la première représentation. Ces nouvelles définitions peuvent être sans rapport avec les définitions dans la représentation initiale.

Jusqu'ici nous n'avons considéré que la question de la ι-réduction associée aux définitions de fonctions. Dans certains cas, on doit de plus mettre en évidence les propriétés directement liées à la définition des fonctions (δ-réduction) qui peuvent être utilisées implicitement dans les termes de preuve.

Par exemple, la fonction de multiplication par 2 d'un entier double est définie en fonction de l'opération plus dans la représentation unaire des entiers. Dans la représentation binaire, cette définition peut se faire indépendamment de plus, en ajoutant simplement un zéro à la fin de la représentation binaire du nombre.

Supposons que nous avons déjà prouvé et que nous voulons abstraire le théorème suivant par rapport à la représentation concrète des entiers naturels sous la forme d'entiers de Peano.

$$\forall n, m : \mathsf{nat}.\ (\mathsf{double}\ (\mathsf{plus}\ n\ m)) = (\mathsf{plus}\ (\mathsf{double}\ n)\ (\mathsf{double}\ m)) \tag{6.4}$$

Une preuve de (6.4) peut utiliser la propriété que (double n) et (plus n n) sont convertibles modulo δ-réduction. Malheureusement, cette propriété ne sera pas nécessairement conservée lors d'un changement de représentation. C'est pourquoi il faut prendre soin de la rendre explicite sous la forme d'une équation logique reliant double et plus.

$$\forall n : \mathsf{nat}.\ (\mathsf{double}\ n) = (\mathsf{plus}\ n\ n)$$

Remarque : on aurait pu se contenter de modifier l'algorithme de mise en évidence des étapes implicites de raisonnement en remplaçant toutes les étapes de dépliage de définitions (δ-réduction)

pour des réécritures sous des équations équivalentes. Nous n'avons pas choisi cette méthode à cause du risque d'explosion de la taille des termes de preuve qu'elle présente.

6.5.2 Cas des propositions logiques

Considérons maintenant la relation d'ordre lt :

$$\mathsf{lt} = \lambda n, m : \mathsf{nat}.(\mathsf{le}\ (\mathsf{S}\ n)\ m)$$

Une conséquence des définitions de le et lt est que (le (S n) m) et (lt n m) sont convertibles alors qu'il n'y a aucune raison (*a priori*) pour que lt soit défini à partir de le.

Cela peut devenir un problème dans la nouvelle représentation ; les homologues de lt et le peuvent ne pas être reliés de la même manière. Par conséquent l'égalité implicite entre (lt n m) et (le (S n) m) peut être perdue. Une première idée pour résoudre cette difficulté est de procéder exactement comme dans le cas de la fonction double. On suffit alors de montrer que

$$\forall n, m : \mathsf{nat}.\ (\mathsf{le}\ (\mathsf{S}\ n)\ m) == (\mathsf{lt}\ n\ m) \tag{6.5}$$

où == représente l'égalité de Leibniz pour la sorte Type [1]. Cependant, au moment de l'instantiation du développement formel par une nouvelle représentation concrète, on risque de se trouver bloqué. L'instantiation sur les binaires (voir le chapitre suivant) conduit à des problèmes avec l'égalité de Leibniz. On peut se trouver confronter à montrer que (true=true)==(¬false=true). Ce qui n'est pas démontrable dans le système Coq. Afin d'éviter de rencontrer ce genre de difficultés lors de la réinstantiation, nous nous contentons de la propriété d'équivalence suivante :

$$\forall n, m : \mathsf{nat}.\ (\mathsf{le}\ (\mathsf{S}\ n)\ m) \leftrightarrow (\mathsf{lt}\ n\ m) \tag{6.6}$$

où $\leftrightarrow$ est la conjonction de deux implications logiques $A \leftrightarrow B \ \equiv\ (A \rightarrow B) \wedge (B \rightarrow A)$. Malheureusement, cette propriété ne peut pas être utilisée directement. En effet on voudrait pouvoir substituer (lt n m) par (le (S n) m) et on ne peut pas réécrire directement avec une relation d'équivalence.

Dans le reste de cette section, nous allons étudier les deux possibilités de résoudre ce problème. Une première solution est de supposer que l'axiome $\forall P, Q : \mathsf{Prop}.\ (P \leftrightarrow Q) \rightarrow P == Q$ est valide dans le calcul des constructions. A partir de cette hypothèse et de la propriété (6.6), on peut déduire l'équation (6.5) et réécrire avec cette équation. La seconde possibilité est de considérer le sétoïde (Prop,$\leftrightarrow$) et de faire de la réécriture dans ce sétoïde.

6.5.3 Traitement de l'équivalence propositionnelle comme l'égalité de Leibniz

Il est cohérent [Miq02] d'ajouter l'axiome suivant dans le système Coq.

$$\mathsf{Axiom\ iff\_implies\_equiv}\ : \forall P, Q : \mathsf{Prop}.\ (P \leftrightarrow Q) \rightarrow P == Q$$

Cependant, tout au long de ce travail, nous avons toujours privilégié une approche définitionnelle sans utiliser aucun paramètre ou axiome. De plus, toutes les preuves transformées sont vérifiées par le noyau de Coq avant d'être acceptées. C'est pourquoi nous avons préféré utiliser la réécriture avec le sétoïde (Prop,$\leftrightarrow$) plutôt que d'introduire un axiome dans notre développement.

[1] `Inductive eqT [A : Type ; x : A] : A->Prop := refl_eqT : x==x.`

6.5.4 Réécriture pour les sétoïdes

Le système Coq fournit une tactique de réécriture Setoid_rewrite [Coq02, Chap. 20] qui permet de réécrire suivant une relation d'équivalence. Nous avons considéré le sétoïde (Prop,$\leftrightarrow$) et testé comment utiliser la tactique Setoid_rewrite pour remplacer (lt n m) par (le (S n) m). Réécrire en utilisant des sétoïdes nécessite de montrer que toutes les opérations traversées pour atteindre le terme à réécrire sont bien des morphismes. Par exemple réécrire (lt n m) $\wedge$ A en (le (S n) m) $\wedge$ A pour un A quelconque nécessite de montrer la propriété suivante :

$$\forall A, B, C, D : \mathsf{Prop}.\ (A \leftrightarrow C) \rightarrow (B \leftrightarrow D) \rightarrow A \wedge B \rightarrow C \wedge D.$$

De telles propriétés peuvent être facilement établies dans le système Coq tant que l'on ne considère que des connecteurs logiques. C'est un peu plus compliqué quand on considère des propriétés de décidabilité comme c'est le cas avec sumbool[2]. On construit alors une nouvelle fonction qui, à partir d'un élément de type $\{A\} + \{B\}$, construit un élément de type $\{C\} + \{D\}$.

$$\forall A, B, C, D : \mathsf{Prop}.\ (A \leftrightarrow C) \rightarrow (B \leftrightarrow D) \rightarrow \{A\} + \{B\} \rightarrow \{C\} + \{D\}.$$

On utilise cette propriété pour établir les conjectures produites par l'algorithme de mise en évidence des étapes de calcul présenté à la section 6.4.4. Cela conduit à des preuves un peu plus compliquées, mais permet d'éviter d'utiliser un axiome dans notre développement formel.

6.5.4.1 Preuves de propriétés de décidabilité

Les propriétés de décidabilité des relations s'expriment en Coq en utilisant les types inductifs sumor et sumbool.

```
Inductive sumor [A : Set; B : Prop]  : Set :=
      inleft : A->A+{B} | inright : B->A+{B}.

Inductive sumbool [A : Prop; B : Prop]  : Set :=
      left : A->{A}+{B} | right : B->{A}+{B}.
```

Les théorèmes de décidabilité peuvent contenir un nombre arbitraire de clauses ; la plupart du temps, il n'y a que deux clauses : par exemple $\forall n, m : \mathsf{nat}.\{(\mathsf{lt}\ n\ m)\} + \{(\mathsf{le}\ m\ n)\}$. Dans ce cas, on utilise le type inductif sumbool qui prend en arguments des propriétés de sorte Prop. Lorsque les clauses sont en nombre supérieur à 2, on doit utiliser le type inductif sumor. Pour être générale, notre approche doit produire une collection de théorèmes permettant de traverser autant de définitions inductives (comme sumor) que nécessaire pour atteindre la formule logique.

6.6 Implantation dans Coq et expériences pratiques

6.6.1 Implantation

Dans ce chapitre, nous avons présenté un algorithme d'élimination des constructions de filtrage explicite dans les termes de preuves. Ensuite nous avons proposé un mécanisme de mise en évidence des étapes de raisonnement par calcul dans les termes de preuve. Ces étapes ont été remplacés par des étapes de réécriture. Ces deux algorithmes ont été implantés en Caml au dessus du système Coq.

[2] `Inductive sumbool [A : Prop ; B : Prop] : Set :=`
`left : A->A+B | right : B->A+B.`

6.6.2 Expériences

Avec l'outil développé dans le système Coq, nous avons pu mettre en évidence des étapes de raisonnement implicite dans les termes de preuve des bibliothèques Arith et PolyList de Coq. Cette transformation a permis de construire des preuves indépendantes de la représentation concrète des entiers et des listes polymorphes.

6.6.2.1 Traduction de la bibliothèque Arith de Coq

Nous avons donné une description abstraite (sous forme de propriétés logiques) de toutes les fonctions et de toutes les relations définies dans la bibliothèque Arith de Coq. Une fois ces déclarations faites, nous avons pu exécuter les algorithmes proposés précédemment pour produire des termes indépendants de la représentation inductive des entiers de Peano. Nous avons pu transformer toute la bibliothèque y compris toutes les propriétés de décidabilité liées à la relation d'ordre lt et les propriétés de la division euclidienne sur les entiers. Au total, nous avons pu rendre explicites les étapes de ι-réduction implicites dans plus de 150 termes de preuve ; ceci de manière complètement automatique.
Nous verrons dans le chapitre suivant que cela nous a permis de développer une bibliothèque complète (contenant tous les théorèmes de Arith sur la représentation binaire des entiers).

6.6.2.2 Abstraction des listes

La même technique a pu être utilisée pour abstraire la représentation concrète des listes polymorphes. Ainsi on a rendu indépendante de la représentation inductive des listes les preuves des propriétés des opérations de base sur les listes : la concaténation, les fonctions de calcul de la tête, de la queue, et de la longueur d'une liste. Nous avons aussi considéré les propriétés d'inclusion entre listes.

6.6.2.3 Variante :modification d'une définition

De la même façon que l'on fait abstraction de la représentation concrète d'un type de données, on peut simplement vouloir abstraire la représentation concrète d'une fonction. On peut, par exemple, vouloir remplacer la définition habituelle de l'addition plus par sa définition récursive terminale. Dans ce cas, il suffit de construire un terme n'utilisant pas les propriétés calculatoires liées à la définition de cette fonction. Cette variante consiste simplement à converser la même représentation concrète des types de données, à modifier la définition de la fonction puis à appliquer exactement les mêmes procédés que pour les transformations de preuve étudiées précédemment dans ce chapitre.

6.7 Conclusion

Maintenant que l'on dispose des moyens de faire abstraction de la représentation concrète des données dans les termes de preuve, on peut porter facilement les preuves d'une représentation à l'autre. En effet, les termes de preuves sont devenus indépendants de la représentation concrète des notions mathématiques sur lesquels ils expriment des propriétés.

Chapitre 7

De nouvelles implantations concrètes

Maintenant que l'on dispose des moyens de rendre explicite toutes les étapes des démonstrations (en particulier les étapes de calcul), on peut étudier comment mettre une nouvelle représentation des données à la place de la représentation existante. Cette transformation est immédiate ; il s'agit d'un simple transformation syntaxique. En effet, le terme de preuve ne comporte plus aucune étape reposant sur des propriétés propres aux types inductifs mis en jeu. Il suffit alors d'associer au type initial un type cible, aux constructeurs du type initial leurs homologues sous forme de constructeurs d'un autre type de données ou même de fonctions.

Dans tous les cas, il nous faut par contre définir sur le nouveau type de données concret les opérations qui nous avons *axiomatisées* au moment de notre transformation des termes de preuve dans le chapitre précédent.

7.1 Nouveaux types de données concrets

Revenons à notre exemple des entiers naturels représentés de façon unaire. Nous pouvons les représenter en binaire, ce qui nous donnera des opérations plus efficaces. Nous reprenons la représentation des entiers binaires introduite dans Coq par Pierre Crégut au moment de l'implantation de la tactique Omega [Coq02, chap. 16]. Le type pos décrit les éléments strictement positifs : xH correspond à 1, xO à l'opération $2 \cdot x$ et xI à l'opération $2 \cdot x + 1$. Le type bin permet d'ajouter l'élément 0 à ce type.

```
Inductive bin : Set := b0 : bin | bp : pos -> bin.
Inductive pos : Set := xH : pos | x0 : pos -> pos | xI : pos -> pos.
```

On peut vérifier facilement que le type bin est isomorphe à nat ; et que deux éléments isomorphes ont bien la même interprétation. Par ailleurs, les entiers binaires représentés dans un système comme Coq sont directement les entiers relatifs. A partir de ce type, on peut aussi définir le type des entiers relatifs positifs par un Σ-type. Ainsi on définit Zpos comme l'ensemble des entiers n de type $\mathbb{Z}$ vérifiant la propriété $0 \leq n$.

$$\mathsf{Definition}\ \ \mathsf{Zpos}\ :=\ \{n : \mathsf{Zpos} \mid 0 \leq n\}$$

7.1.1 Mise en correspondance des types de données

La première étape de la réinstantiation d'une théorie sur un nouveau type de données concret consiste à décrire les constructeurs de nat dans les nouvelles représentations. Le constructeur O

nat	bin	Zpos
O	b0	z0
S	bS	zS
plus	bplus	Zpos_plus

FIG. 7.1 – Mise en correspondance de nat, bin et Zpos

du type nat est transformé en le constructeur b0 du type bin. Nous donnons ici la définition de la fonction successeur bS sur les entiers binaires.

```
Fixpoint aux_bS [p : pos] : pos :=
 Cases p of xH => (xO xH) | (xI t) => (xO (aux_bS t)) | (xO t) => (xI t) end.

Definition bS : bin ->  bin :=
   [n : bin]  Cases n of b0 => (bp xH) | (bp p) => (bp (aux_bS p)) end.
```

De même que pour les entiers binaires, on définit les opérations z0 et zS qui joueront le rôle de O et S pour le type Zpos.

```
Definition b0 :Zpos := (exist ? ? `0` (Zle_n `0`)).
```

La fonction zS, de type Zpos → Zpos, prend en argument un nombre n et une preuve qu'il est positif et construit le nombre $n + 1$ et une preuve qu'il est positif. Dans le système COQ, sa définition se fait interactivement sous forme d'une preuve.

7.2 Représentation concrète des fonctions

Nous avons vu dans le chapitre précédent comment exprimer les propriétés des fonctions sous forme d'équations. Dans le cadre du changement de représentation de nat vers bin ou Zpos, on va définir de nouvelles fonctions, par exemple bplus et zplus à la place de plus.

7.2.1 Spécification de leurs propriétés

De nouvelles opérations d'addition, de multiplication, etc. sont définies sur les deux nouveaux types de données concrets. Afin de s'assurer que ces fonctions implantent bien l'addition sur les entiers, nous devons vérifier qu'elles ont bien les propriétés de l'addition des entiers de Peano. Dans le cas du type bin, il faut donc vérifier que :

$$\forall n : \mathsf{bin}.\ (\mathsf{bplus}\ \mathsf{b0}\ n) = n$$
$$\forall n, m : \mathsf{bin}.\ (\mathsf{bplus}\ (\mathsf{bS}\ n)\ m) = (\mathsf{bS}\ (\mathsf{bplus}\ n\ m))$$

La plupart du temps, les fonctions sont redéfinies afin d'être efficaces dans la nouvelle représentation. Par exemple, l'addition sur les entiers binaires est directement définie par récursion structurelle sur la représentation binaire. En effet, la complexité de l'algorithme d'addition sur les entiers binaires est logarithmique alors que l'algorithme d'addition sur les entiers de Peano est linéaire. Cependant dans certains cas, il n'existe pas d'autre manière de définir une fonction que celle donnée dans la représentation initiale. Les fonctions *factorielle* et *Fibonacci* en sont de bons exemples. Le calcul

de (fact n) ne peut se faire de façon exacte qu'en calculant d'abord (fact $n-1$). De même pour la fonction *Fibonacci*, le calcul de (fib n) nécessitera toujours de calculer au moins (fib $n-1$).

Dans ce cas, nous allons voir qu'il est possible de transformer une fonction définie par point-fixe et filtrage structurel en une fonction dont la structure de calcul est indépendante du type inductif sur lequel elle est définie.

7.2.2 Transformation d'une construction par point-fixe

Dans ce paragraphe, nous proposons une méthode pour transformer les fonctions définies par récursion structurelle en utilisant la construction Fixpoint en des fonctions définies par récursion suivant un ordre bien-fondé.

Regardons de plus près la définition de la factorielle sur les entiers de Peano nat.

```
Fixpoint fact [n:nat] : nat :=
  Cases n of O     => (S O)
           | (S p) => (mult (S p) (fact p))
  end.
```

L'utilisation d'une représentation binaire des entiers ne donne pas de moyens plus efficaces de calculer la factorielle. La structure du calcul sera basée sur la représentation unaire des entiers et nécessitera l'utilisation d'une définition par ordre bien-fondé. Dans un premier temps, on va chercher à définir la fonction de calcul de la factorielle par ordre bien-fondé sur les entiers de Peano. Pour l'instant, on ne change pas de représentation inductive, on transforme simplement la définition de la fonction fact sur les entiers de Peano.

1. On considère la relation d'ordre strict sur les entiers de Peano lt. Cette relation d'ordre correspond à la clôture transitive de l'ordre sous-terme sur nat. On considère également une preuve lt_wf que l'ordre lt est bien fondé ; c'est-à-dire qu'il n'existe pas de chaînes infinies décroissantes sur nat pour l'ordre lt.
2. On va maintenant définir une fonction de type

   $$\forall n : \mathsf{nat}.\ \forall f : (\forall m : \mathsf{nat}.\ (\mathsf{lt}\ m\ n) \rightarrow \mathsf{nat}).\ \mathsf{nat}.$$

   Cette fonctionnelle F prend en argument un entier n et une fonction f permettant de faire des appels récursifs sur des arguments inférieurs à n. Elle est définie en Coq par le code suivant :

   ```
   Definition F : (n:nat)(f:(m:nat)(lt m n)->nat)nat
     := [n:nat] <[n:nat](f:(m:nat)(lt m n)->nat)nat>
        Cases n of O     => [f:?](1)
                 | (S p) => [f:?](mult (S p) (f p (le_n (S p))))
        end.
   ```

   Dans cette définition, (le_n (S p)) est une preuve que (lt p (S p)).
3. Ensuite on va utiliser l'opérateur well_founded_induction pour construire la fonction de calcul de la factorielle wffact. L'opérateur well_founded_induction prend en argument un type A de l'élément sur lequel se fait la récursion, un ordre sur A, une preuve que cet ordre est bien fondé, le type de retour de la fonction ainsi que la fonctionnelle permet de faire un pas de récursion.

```
Definition wffact : nat->nat
  := (well_founded_induction nat lt lt_wf [_:nat]nat F).
```

A cette étape, on dispose d'une nouvelle fonction de calcul de la factorielle. Malheureusement les termes (wffact O) et (wffact (S n)) ne se réduisent pas. On va donc simuler ces réductions par une équation de point fixe.

4. Pour cela, nous suivons la technique proposée par A. Balaa et Y. Bertot dans [BB00], on en déduit alors l'équation de point fixe simplifiée suivante :

```
(wffact n) = Cases n of O     => (1)
                      | (S p) => (mult (S p) (wffact p))
             end
```

Dans ce cas, on n'utilise plus directement la structure des entiers de Peano pour calculer récursivement, mais plutôt les propriétés de l'ordre lt sur nat. Il faut maintenant abstraire un peu plus et remplacer le traitement par cas fait avec l'expression Cases n of ... par une analyse par cas indépendante de la structure des entiers de Peano.

7.2.3 Gestion du filtrage

Pour cela, on utilise une technique inspirée de celle proposée par McBride et McKinna dans [MM04]. Il s'agit de définir un prédicat de filtrage sur les entiers en utilisant un type inductif dépendant :

```
Inductive filtrage : nat ->  Set :=
  is_O: (filtrage O)
 | is_S: (y : nat)  (filtrage (S y)).
```

Ce prédicat de filtrage doit être suffisamment général pour être utilisé avec les autres représentations possibles des entiers naturels. Par exemple, on peut définir filtrage_bin comme :

```
Inductive filtrage_bin : bin -> Set :=
   is_b0 : (filtrage b0)
 | is_bS : (y:bin)(filtrage (bS y)).
```

Dans ce cas, b0 et bS ne sont pas des constructeurs, mais le type inductif filtrage permet de raisonner sur ces entiers binaires en les *voyant* comme des entiers de Peano.

On peut définir une fonction de filtrage filtre qui aura le type $\forall y$: nat. (filtrage y). Une telle fonction sera généralement définie par preuve. La principale difficulté étant de donner un terme appartenant à la bonne instance de la famille inductive filtrage dans chaque cas. Supposons que l'on dispose d'une fonction qui construit un élément de type (filtrage y) à partir d'un élément y de type nat. A partir de cette fonction de filtrage, on peut définir la fonctionnelle F2 puis la fonction wffact de calcul de la factorielle sur les entiers.

```
Definition F2 : (n:nat)(f:(m:nat)(lt m n)->nat)nat
  := [n:nat] <[n:nat] (f:(m:nat)(lt m n)->nat)nat>
     Cases (filtre n) of is_zero     => [f:?](1)
                       | (is_S p)    => [f:?](mult (S p) (f p (le_n (S p)))
     end.

Definition wffact2 : nat->nat
  := (well_founded_induction nat lt lt_wf [_:nat]nat F2).
```

On remarquera que le prédicat d'élimination donné dans les diamants $< \ldots >$ est inchangé quand le filtrage se fait sur (filtre n) plutôt que sur n. En particulier le bon type est inféré pour f dans chaque branche grâce à la dépendance par rapport à n dans le type (filtrage n) de (filtre n).

Prouver l'équation de point-fixe générale se fait simplement en faisant un traitement par cas sur (filtre n) qui est un élément de type (filtrage n). La technique proposée par A. Balaa et Y. Bertot [BB00] fonctionne très bien et permet d'obtenir l'égalité suivante :

$$\forall n : \mathsf{nat}.\ (\mathsf{wffact2}\ n) = (\mathsf{F2}\ n\ \lambda p : \mathsf{nat}.\ \lambda h : (\mathsf{lt}\ p\ n).\ (\mathsf{wffact2}\ p))$$

A partir de cette équation de point fixe, on va chercher à exprimer sous forme d'équations les règles de réduction de fact. Pour cela, nous commençons par montrer les deux propriétés suivantes :

$$(\mathsf{filtre}\ O) = \mathsf{is\_O} \tag{7.1}$$

$$\forall n : \mathsf{nat}.\ (\mathsf{filtre}\ (\mathsf{S}\ n)) = (\mathsf{is\_S}\ n) \tag{7.2}$$

7.2.3.1 Remarque importante

La façon dont la fonction filtre est implantée n'intervient pas dans la preuve des deux propriétés ci-dessus. En effet, les données nécessaires pour établir ces propriétés se trouvent dans le type (dépendant) filtrage. Ces démonstrations se font en utilisant l'égalité dépendante eq_dep de Coq. On définit eq_filtrage par (eq_dep nat filtrage) et on démontre d'abord les théorèmes

$$\forall y : \mathsf{nat}.\ \forall x : (\mathsf{filtrage}\ y).\ y = b0 \rightarrow (\mathsf{eq\_filtrage}\ y\ x\ \mathsf{b0}\ \mathsf{is\_O})$$

et

$$\forall n, m : \mathsf{nat}.\ m = (\mathsf{S}\ n) \rightarrow \forall v : (\mathsf{filtrage}\ (\mathsf{S}\ n)).\ (\mathsf{eq\_filtrage}\ (\mathsf{S}\ n)\ v\ (\mathsf{S}\ n)\ (\mathsf{is\_S}\ n)).$$

La démonstration de ces théorèmes nécessite l'utilisation des propriétés d'injectivité et de non-confusion des constructeurs O et S du type nat. Ces preuves utilisent les tactiques Discriminate et Injection de Coq. Ensuite on déduit les équations (7.1) et (7.2) des deux propriétés que l'on vient de montrer.

Quand on considère la représentation binaire des entiers, il suffit de disposer des théorèmes

$$\forall n : \mathsf{bin}.\ \neg\mathsf{b0} = (\mathsf{bS}\ n) \qquad \forall n, m : \mathsf{nat}.\ (\mathsf{bS}\ n) = (\mathsf{bS}\ m) \rightarrow n = m$$

Ces propriétés peuvent être démontrées par induction structurelle sur cette représentation.

Pour conclure, nous obtenons donc sous forme d'équations les deux règles de calcul attendus pour la factorielle, à savoir :

$$(\mathsf{wffact2}\ \mathsf{O}) = (1) \qquad (\mathsf{wffact2}\ (\mathsf{S}\ p)) = (\mathsf{mult}(\mathsf{S}\ p)\ (\mathsf{wffact2}\ p))$$

7.2.3.2 Le cas de l'addition

On aurait pu procéder de la même manière pour définir l'opération d'addition sur les entiers binaires. Cependant, on va construire une nouvelle fonction bplus directement par récursion structurelle sur les entiers binaires. On montrera simplement que cette fonction bplus vérifie la spécification de l'addition telle qu'elle est donnée dans l'arithmétique de Peano.

$$\forall n : \mathsf{bin}.\ (\mathsf{bplus}\ \mathsf{b0}\ n) = n \qquad \forall n, m : \mathsf{bin}.\ (\mathsf{bplus}\ (\mathsf{bS}\ n)\ m) = (\mathsf{bS}\ (\mathsf{bplus}\ n\ m))$$

Dès lors que le passage à la représentation binaire des entiers permet de produire des fonctions plus efficaces, on choisit de définir ces fonctions directement sans passer par la transformation présentée dans la section précédente. Ainsi les opérations arithmétiques de base comme l'addition, la multiplication et la soustraction sont directement redéfinies dans le nouveau contexte.

7.2.4 Transfert de la relation d'ordre d'une représentation à l'autre

Afin de pouvoir transformer une définition par ordre bien-fondé sur les entiers de Peano en une définition par ordre bien-fondé équivalente sur les entiers binaires. Il faut au préalable définir l'homologue de la relation d'ordre lt sur les entiers de Peano pour la représentation binaire des entiers. Pour cela, on va définir un nouvel ordre wfblt sur les entiers binaires à partir de lt et d'une traduction des entiers binaires vers les entiers de Peano

$$(\mathsf{wfblt}\ n\ m) \equiv (\mathsf{lt}\ (\mathsf{b2n}\ n)\ (\mathsf{b2n}\ m))$$

La fonction de traduction b2n est définie de la façon suivante. p2n est la fonction de conversion des entiers positifs vers les entiers de Peano. b2n est l'extension de cette fonction au cas 0.

```
Fixpoint p2n [p : pos] : nat :=
 Cases p of
   one => (S O)
  | (pI t) => (S (mult (p2n t) (S (S O))))
  | (pO t) => (mult (p2n t) (S (S O)))
 end.

Definition b2n : bin ->  nat :=
   [n : bin]  Cases n of b0 => O | (bp p) => (p2n p) end.
```

Dans le cas de la représentation Zpos,

$$(\mathsf{zlt}\ n\ m) \equiv (\mathsf{lt}\ (\mathsf{z2n}\ n)\ (\mathsf{z2n}\ m))$$

où z2n consiste simplement à retourner la première composante n d'un objet de type $\{n : \mathbb{Z} \mid 0 \leq n\}$.

Les définitions de wfblt et zlt étant faites par traduction d'une représentation dans l'autre, on peut très facilement montrer que ces relations sont bien fondées à partir de la propriété que l'ordre lt est bien fondé[1].

Ainsi on obtient les éléments nécessaires à la construction d'une fonction par ordre bien fondé sur les représentations des entiers binaires. Nous pouvons donc transformer les descriptions par ordre bien-fondé des fonctions sur les entiers de Peano en des descriptions équivalentes sur les entiers binaires, qu'ils soient représentés par le type bin ou le type Zpos.

7.2.4.1 Traduction vers la représentation binaire

Une fois que l'on a obtenu une définition par ordre bien-fondé, il ne reste plus qu'à traduire syntaxiquement la définition wffact en une définition équivalente bwffact sur les entiers binaires, en utilisant les opérations binaires au lieu des opérations unaires : à savoir filtrage_bin (resp. filtre_bin) au lieu de filtrage (resp. filtre). Cela donne les définitions suivantes :

```
Definition bF : (n:bin)(f:(m:bin)(blt m n)->bin)bin
  := [n:bin] <[n:bin] (f:(m:bin)(blt m n)->bin)bin>
     Cases (filtre_bin n) of is_b0     => [f:?](bS b0)
                          | (is_bS p)    => [f:?](bmult (bS p) (f p (ble_n (bS p)))
     end.

Definition bwffact : bin->bin
  := (well_founded_induction bin wfblt wfblt_wf [_:bin]bin bF).
```

[1] Pour cela, on utilise le théorème wf_inverse_image de la bibliothèque Wellfounded.

De même que dans le cas des entiers unaires, on applique la technique de preuve de l'équation de point fixe d'une fonction récursive générale proposée par A. Balaa et Y. Bertot dans [BB00]. On dispose alors d'une nouvelle description de la fonction *factorielle* dans la représentation binaire des entiers ainsi que de ses propriétés calculatoires sous forme d'équations.

7.2.5 Les principes d'induction : des fonctions comme les autres

Comme nous l'avons déjà indiqué, les principes d'induction structurelle sur les entiers de Peano peuvent être vus comme des fonctions comme les autres. A ce titre, on peut leur imposer le traitement présenté dans les paragraphes précédents. La seule nouveauté est que le type retourné par la fonction est dépendant de l'entrée. Le principe d'induction dépendant nat_rect est défini dans Coq de la façon suivante :

```
Variable P:nat->Type.
Variable h0:(P O).
Variable hr:(n:nat)(P n)->(P (S n)).

Fixpoint nat_rect [n:nat] : (P n) :=
  <P>Cases n of O     => h0
              | (S p) => (hr p (nat_rect p))
  end.
```

En appliquant le même procédé que pour la factorielle, on en déduit les définitions suivantes :

```
Definition F : (n:nat)(f:(m:nat)(lt m n)->(P m))(P n) :=
[n:nat] <[n:nat](f:(m:nat)(lt m n)->(P m))(P n)>
   Cases (filtre n) of is_O     => [f:?]h0
                     | (is_S p) => [f:?](hr p (f p (le_n (S p))))
   end.

Definition wfnat_rect : (n:nat)(P n) :=
 (well_founded_induction_type nat lt lt_wf [n:nat](P n) F).
```

Les outils issus des travaux de A. Balaa et Y. Bertot [BB00] permettent d'obtenir les équations suivantes :

$$(\mathsf{nat\_rect}\ P\ h0\ hr\ \mathsf{O}) = h0 \qquad \forall n : \mathsf{nat}.\ (\mathsf{nat\_rect}\ P\ h0\ hr\ (\mathsf{S}\ p)) = (hr\ p\ (\mathsf{nat\_rect}\ P\ h0\ hr\ p))$$

Les règles de réduction des principes de récursion sont utiles dans les termes de preuves engendrées par l'application des tactiques Injection et/ou Discriminate. En effet, l'étape d'élimination des constructions d'analyse par cas présentée dans le chapitre 6 consiste à remplacer les occurrences de l'opérateur de filtrage Cases de Coq par l'application d'un principe de récursion équivalent. La plupart du temps, ces filtrages sont non dépendants et il suffit donc de disposer d'un principe de récursion minimal et des règles de réduction qui lui sont associées ; par exemple, le principe non dépendant nat_min_rec

$$\forall P : \mathsf{Set}.\ P \to (\mathsf{nat} \to P \to P) \to \mathsf{nat} \to P$$

et ses règles de réduction (sous forme d'équations) :

$$(\mathsf{nat\_min\_rec}\ P\ v0\ vr\ \mathsf{O}) = v0 \qquad \forall n : \mathsf{nat}.\ (\mathsf{nat\_min\_rec}\ P\ v0\ vr\ (\mathsf{S}n)) = (vr\ n\ (\mathsf{nat\_min\_rec}\ P\ v0\ vr\ n)).$$

Ces principes peuvent être construits en utilisant les définitions par ordre bien-fondé. Par conséquent, leur comportement peut être décrit par des équations comme c'est le cas pour nat_rect.

7.3 Représentation des prédicats inductifs

7.3.1 Définitions et principes d'induction associées

On s'intéresse maintenant à la représentation concrète dans la nouvelle représentation binaire des entiers des prédicats inductifs comme le. A ces prédicats sont associées des principes de raisonnement que l'on doit savoir transférer vers la nouvelle représentation en même temps que la nouvelle définition de le dans la représentation binaire.

Le prédicat le est défini directement sur les entiers binaires. Les définitions suivantes s'inspirent de la définition des relations Zlt et Zle de la bibliothèque ZArith de Coq.

```
Inductive relation : Set :=
  EGAL :relation | INFERIEUR : relation | SUPERIEUR : relation.

Fixpoint compare [x,y:pos]: relation -> relation :=
  [r:relation] <relation> Cases x of
            (pI x') => <relation>Cases y of
                         (pI y') => (compare x' y' r)
                       | (pO y') => (compare x' y' SUPERIEUR)
                       | xH => SUPERIEUR
                       end
          | (pO x') => <relation>Cases y of
                         (pI y') => (compare x' y' INFERIEUR)
                       | (pO y') => (compare x' y' r)
                       | one => SUPERIEUR
                       end
          | one => <relation>Cases y of
                     (pI y') => INFERIEUR
                   | (pO y') => INFERIEUR
                   | one => r
                   end
  end.

Definition Bcompare := [x,y:bin]
  <relation>Cases x of
      b0 => <relation>Cases y of
                b0 => EGAL
              | (bp y') => INFERIEUR
              end
   | (bp x') => <relation>Cases y of
                b0 => SUPERIEUR
              | (bp y') => (compare x' y' EGAL)
              end
  end.

Definition ble := [x,y:bin]~(Bcompare x y)=SUPERIEUR.

Definition blt := [x,y:bin](Bcompare x y)=INFERIEUR.
```

Une fois que l'on a défini ble, il faut démontrer l'homologue du principe de raisonnement par récurrence de le. Cette preuve se fait en Coq par induction structurelle sur les entiers binaires et nécessite la démonstration de deux lemmes principaux qui sont les homologues des constructeurs

du type inductif le, à savoir

$$\forall n : \mathsf{bin}.\ (\mathsf{ble}\ n\ n)$$

et

$$\forall n, m : \mathsf{bin}.\ (\mathsf{ble}\ n\ m) \rightarrow (\mathsf{ble}\ n\ (\mathsf{bS}\ m))$$

Le principe d'induction ble_ind a l'énoncé suivant :

ble_ind : $\forall n : \mathsf{bin}; P : \mathsf{bin} \rightarrow \mathsf{Prop}.\ (P\ n) \rightarrow (\forall m : \mathsf{bin}.\ (\mathsf{ble}\ n\ m) \rightarrow (P\ m) \rightarrow (P\ (\mathsf{S}\ m))) \rightarrow \forall n0 : \mathsf{bin}.\ (\mathsf{ble}\ n\ n0) \rightarrow (P\ n0)$

Expérimentalement, en Coq, ce principe a été démontré directement sur machine. On a énoncé le théorème dans Coq, puis on a commencé à le démontrer en dépliant les définitions et en raisonnant par récurrence en suivant la structure de la fonction compare. Cette technique nous a conduit à prouver de nombreux lemmes auxiliaires de la forme

$\forall p, q : \mathsf{pos}.\ \forall r : \mathsf{relation}.\ (\mathsf{compare}\ p\ q\ \mathsf{INFERIEUR}) = \mathsf{SUPERIEUR} \rightarrow (\mathsf{compare}\ p\ q\ r) = \mathsf{SUPERIEUR}.$

La plupart de ces lemmes se démontrent de la même manière par cas sur r , puis par récurrence sur p et q. Chacune des branches du raisonnement se conclut soit trivialement les deux membres de l'égalité sont égaux, soit par l'application de Discriminate car les constructeurs de tête sont différents.

7.3.2 Liens entre les relations

Il faut relier cette définition à la définition de blt (On rappelle que wfblt est une autre manière de définir un ordre sur les entiers binaires : c'est l'ordre induit sur bin par la traduction b2n). Comme nous l'avons vu dans le chapitre précédent, il suffit d'établir l'équivalence suivante :

$$\forall n, m : \mathsf{bin}.\ (\mathsf{blt}\ n\ m) \leftrightarrow (\mathsf{ble}\ (\mathsf{bS}\ n)\ m)$$

Cette preuve se fait également par double induction structurelle sur n et m et nécessite la démonstration de bon nombre de lemmes intermédiaires semblables à ceux développés pour la preuve du principe d'induction non structurelle sur ble.

Une fois que l'on dispose de cette équivalence, on pourra l'utiliser pour remplacer l'un des membres par l'autre avec la tactique Setoid_Rewrite. Pour que cela soit possible, il faudra au préalable avoir montré que tous les opérateurs traversés pour atteindre cette expression depuis le sommet du terme sont bien des morphismes. C'est ce que nous avons fait dans la section 6.5 du chapitre précédent.

7.4 Mise en correspondance des représentations initiale et finale

Afin de pouvoir procéder à la traduction systématique des termes de preuve, nous devons établir une correspondance formelle entre les éléments décrits dans chacune des représentations considérées.

7.4.1 Les types et leurs éléments

On commence par associer à nat le type bin ou Zpos. Ensuite on va relier ses constructeurs O et S avec leurs nouvelles représentations dans l'arithmétique binaire.

nat	bin	Zpos
O	b0	z0
S	bS	zS

7.4.2 Les fonctions

Il faut ensuite relier entre elles les représentations des fonctions dans les différentes représentations.

plus	bplus	zplus
mult	bmult	zmult
fact	bwffact	zfact
nat_rect	new_bin_rect	new_Zpos_rect
nat_case_rect_min	new_bin_case_rect_min	new_Zpos_case_rect_min

En plus de la mise en correspondance purement syntaxique entre les noms des fonctions dans les différentes représentations, il faut ajouter leurs propriétés caractéristiques dans la base de réécriture (base que l'on a appelée simplification dans le chapitre précédent). Par exemple pour l'addition,

$$\mathsf{bplus\_1} : \forall n : \mathsf{bin}.\ (\mathsf{bplus}\ \mathsf{b0}\ n) = n \qquad \mathsf{bplus\_2} : \forall n, m : \mathsf{bin}.\ (\mathsf{bplus}\ (\mathsf{bS}\ n)\ m) = (\mathsf{S}\ (\mathsf{bplus}\ n\ m)$$

Ces équations sont prouvées à la main une fois pour toutes par l'utilisateur au moment de la définition de la fonction bplus. Elles serviront à prouver les conjectures engendrées lors de l'extraction des étapes implicites de preuves.

7.4.3 Les prédicats inductifs et leurs principes de raisonnement associés

Les prédicats inductifs, leurs constructeurs ainsi que leurs principes d'induction associés doivent aussi trouver des équivalents dans les nouvelles représentations des entiers.

le	ble	zle
le_n	ble_n	zle_n
le_S	ble_bS	zle_zS
le_ind	ble_ind	zle_ind
le_case_ind	ble_case_ind	zle_case_ind

A la main, l'utilisateur doit définir une fois pour toutes le prédicat ble et les propriétés suivantes :

$$\mathsf{ble\_n} : \forall n : \mathsf{bin}.\ (\mathsf{ble}\ n\ n) \quad \mathsf{ble\_bS} : \forall n, m : \mathsf{bin}.\ (\mathsf{ble}\ n\ m) \rightarrow (\mathsf{ble}\ n\ (\mathsf{bS}\ m))$$

Il doit aussi démontrer lui-même le principe d'induction ble_ind calqué sur le principe le_ind mais adapté pour le prédicat ble.

On peut noter que la construction des opérations simulant le, le_ind et lt sur la représentation Zpos des entiers nécessite l'utilisation de l'axiome de non-pertinence des preuves[2] pour pouvoir considérer comme *égales* deux preuves différentes d'une propriété de la forme $0 \leq n$.

7.4.4 Les propriétés et leurs preuves

Jusqu'ici, l'utilisateur a dû définir lui-même les nouvelles opérations comme bplus et montrer les règles de réduction attendues (celles de plus exprimées dans le contexte binaire) sous forme d'équations. Il a également dû définir la nouvelle relation d'ordre ble ainsi que les propriétés de base qui lui sont associées ble_n, ble_bS et ble_ind. Maintenant la traduction de toutes les autres propriétés (distributivité, régularité des opérations, compatibilité de l'ordre avec les opérations comme bplus et bmult, etc.) vont être démontrées automatiquement.

La procédure de traduction automatique des propriétés et de leurs preuves fonctionne de la manière suivante.

[2] $\forall A : \mathsf{Prop}.\forall p, q : A.\ p == q$

1. Elle transforme automatiquement l'énoncé du théorème à traduire en suivant les tables d'association donné dans les sections précédentes.
2. Elle extrait les étapes implicites de raisonnement du terme de preuve initial (grâce à l'algorithme du chapitre précédent).
3. Ce calcul engendre des conjectures reliant types attendu et proposé là où ils ne coïncident pas. Ces conjectures sont traduites vers la représentation binaire et démontrées au moyen des équations caractéristiques (par exemple bplus_1 et bplus_2) des fonctions (par exemple plus).
4. Finalement, le terme complet généré à l'étape 2 est traduit syntaxiquement suivant les tables d'association.

7.5 Expériences et résultats

7.5.1 Traduction de la bibliothèque Arith de Coq

Ces travaux ont permis de transformer toute la bibliothèque Arith distribuée avec Coq. Ainsi nous avons construit une nouvelle bibliothèque représentations en binaire. Dans ce qui suit, nous reprenons les deux exemples conducteur du chapitre 6 et nous montrons les termes résultants après la traduction vers l'arithmétique binaire.
Le λ-terme suivant est une preuve du théorème $\forall n, m : \mathsf{bin}.\ (\mathsf{bplus}\ n\ m) = (\mathsf{bplus}\ m\ n)$:

$$\begin{array}{l}\lambda n : \mathsf{bin}.\ (\ \mathsf{new\_bin\_ind}\\ \qquad (\lambda n0 : \mathsf{bin}.n0 = (\mathsf{bplus}\ n0\ \mathsf{bO}))\\ \qquad (Ha'\ (\mathsf{refl\_equal\ bin\ bO}))\\ \qquad \lambda n0 : \mathsf{bin}; H : (n0 = (\mathsf{bplus}\ n0\ \mathsf{bO})).\\ \qquad\quad (Hb'\ n0\ (\mathsf{f\_equal\ bin\ bin\ bS}\ n0\ (\mathsf{bplus}\ n0\ \mathsf{bO})\ H))\ n)\end{array}$$

avec

$$\begin{array}{c}Ha' : \mathsf{bO} = \mathsf{bO} \Rightarrow \mathsf{bO} = (\mathsf{bplus\ bO\ bO})\\ Hb' : \forall n : \mathsf{bin}.\ (\mathsf{bS}\ n) = (\mathsf{bS}\ (\mathsf{bplus}\ n\ \mathsf{bO})) \Rightarrow (\mathsf{bS}\ n) = (\mathsf{bplus}\ (\mathsf{bS}\ n)\ \mathsf{bO})\end{array}$$

Ces deux conjectures ont été prouvées automatiquement grâce aux équations simulant les règles de réduction dans la nouvelle représentation.

Regardons maintenant l'autre exemple étudié lors du chapitre précédent. Nous allons traduire le terme de preuve vers Zpos au lieu de bin. Cela ne pose pas de difficultés particulières, il suffit de traduire les éléments du contexte unaire nat vers des définitions dans le contexte des entiers binaires positifs Zpos. Le terme obtenu est donné à la figure 7.2. Le terme example_rr1' est une preuve que

$$\forall m : \mathsf{Zpos}.\ \mathsf{True} \rightarrow (\mathsf{new\_Zpos\_case\_rect\_min\ Prop\ False}\ \lambda\_ : \mathsf{Zpos.True}\ (\mathsf{zS}\ m)) \tag{7.3}$$

et example_rr2' une preuve que

$$(\mathsf{new\_Zpos\_case\_rect\_min\ Prop\ False}\ \lambda\_ : \mathsf{Zpos.True\ zO}) \rightarrow \mathsf{False} \tag{7.4}$$

Les deux énoncés (7.3) et (7.4) sont démontrés automatiquement par réécriture en utilisant les règles de réduction associées à la constante new_Zpos_case_rect_min.

```
λn : Zpos.λH : (zle n zO).
  let H0 = (zle_case_ind n λp : Zpos.p = zO → n = zO
                         λH0 : n = zO.
                           (eq_ind Zpos zO λn : Zpos.n = zO (refl_equal Zpos zO)
                                   n (sym_eq Zpos n zO H0))
                         λm : Zpos.λH0 : (zle n m).λH1 : (zS m) = zO.
                           (let H2 = (example_rr2' (eq_ind Zpos (zS m)
                                                   λe : Zpos.
                                                   (new_Zpos_case_rect_min Prop False λ_ : Zpos.True e)
                                                   (example_rr1' m I) zO H1))
                            in (False_ind (zle n m) → n = zO H2) H0)
                         zO H)
  in (H0 (refl_equal Zpos zO)).
```

FIG. 7.2 – Le terme de preuve du théorème example après la traduction vers Zpos

7.5.2 Traduction de la bibliothèque de Coq sur les listes polymorphes

En plus de la bibliothèque Arith de COQ, nous nous sommes intéressés à la bibliothèque PolyList sur les listes polymorphes. Nous avons traduit les listes (où le premier élément est le plus facile à extraire) vers les listes renversées (où le dernier élément est le plus facile à extraire). Nous avons défini les homologues des fonctions append, head, tail et length dans cette nouvelle représentation et nous avons prouvé leurs propriétés. De plus, nous avons traduit le principe d'induction list_ind vers un principe équivalent sur les listes renversées. Nous avons aussi traduit les opérateurs de récursion minimaux et avons prouvé leurs règles de réduction sous forme d'équations.

7.6 Conclusion

Dans ce chapitre, nous avons montré comment mettre à jour une théorie développée formellement lors d'un changement de représentation d'une des données mises en jeu. Nous nous sommes focalisés sur l'exemple du passage des entiers de Peano aux entiers binaires. Cependant, le travail présenté ici peut se généraliser facilement au changement de représentations de n'importe quel type dès lors qu'il est représenté par un type inductif simple. La question du changement de représentation de notions mathématiques décrites par des types dépendants sera étudiée partiellement dans le chapitre suivant.

Chapitre 8

De nouvelles approches

Dans ce chapitre, nous présentons les travaux les plus récents effectués dans le cadre de cette thèse. Nous étudions la question des changements de représentation pour les notions décrites par des types dépendants, par exemple l'ensemble des matrices. Nous étudions plus particulièrement la question de la représentation des fonctions en présence des types dépendants. Finalement, nous décrivons une nouvelle méthode de transformation d'une théorie développée formellement lors d'un changement de représentation concrète d'une des données.

8.1 Représentations matricielles

Nous présentons la description fonctionnelle des opérations de base sur les matrices carrées sur un anneau quelconque. Cette formalisation est une expérience de développement d'une théorie avec des types dépendants. En plus de la description formelle de l'anneau des matrices, nous montrons comment passer de la représentation inductive des vecteurs à une représentation par produit cartésien.

8.1.1 Une formalisation opérationnelle des matrices carrées

8.1.1.1 Ensemble support

Notre formalisation des matrices est générique, c'est-à-dire que ses opérations et leurs propriétés sont définies de façon indépendante de l'ensemble support et de ses opérations.

L'ensemble support est défini par un module suivant :

```
Module Type Carrier.
Parameters A :  Set.
Parameters Aopp :  A ->  A.
Parameters Aplus : A -> A ->  A; Amult : A -> A ->  A.
Parameters A0 : A; A1 : A.
Parameters Aeq :  A -> A ->  bool.

Axiom A_ring :  (Ring_Theory Aplus Amult A1 A0 Aopp Aeq).

Add Abstract Ring A Aplus Amult A1 A0 Aopp Aeq A_ring.

End Carrier.
```

Les vecteurs et les matrices seront définis comme des foncteurs prenant en argument ce module. Ainsi on obtient une description générique des vecteurs et des matrices que l'on pourra instancier avec n'importe quel ensemble support.

8.1.1.2 Opérations sur les vecteurs

Les vecteurs sont représentés par un type inductif dépendant :

```
Inductive vect [A:Set] : Set :=
          vnil : (vect A 0)
        | vcons : (n:nat)A->(vect A n)->(vect A (S n)).
```

A partir de cette définition inductive, on montre en utilisant les techniques d'inversion et l'égalité dépendante sur les familles inductives que

$$\forall v : (\mathsf{vect}\ 0).\ v = \mathsf{vnil} \tag{8.1}$$

$$\forall n : \mathsf{nat};\ v : (\mathsf{vect}\ (\mathsf{S}\ n)).\ \exists n : \mathsf{nat}; v' : (\mathsf{vect}\ n).\ v = (\mathsf{vcons}\ n\ a\ v') \tag{8.2}$$

Les opérations sur les vecteurs sont définies par analyse par cas et récursion structurelle. Par exemple, la fonction d'addition de deux vecteurs peut être définie en Coq de manière suivante :

```
Fixpoint addvect [ n : nat; v : (vect A n) ] : (vect A n) ->  (vect A n) :=
 <[k : nat] (vect A k) ->  (vect A k)> Cases v of
     vnil => [v' : ?]  (vnil A)
    | (vcons n1 x1 v1) =>
        [v' : (vect A (S n1))]
         (<[k : nat] [v' : (vect A k)] k = (S n1) ->  (vect A k)> Cases v' of
              vnil => [h : ?]  (vnil A)
             | (vcons n2 x2 v2) =>
                 [h : ?]
                  (vcons
                   A n2 (Aplus x1 x2)
                   (addvect
                    n2
                    (eq_rec
                     nat n1 [n : nat]  (vect A n) v1 n2
                     (eq_add_S_tr n1 n2 (sym_eq ? ? ? h))) v2))
          end (refl_equal nat (S n1)))
 end.
```

Le type de cette fonction d'addition est

$$\forall n : \mathsf{nat}.\ (\mathsf{vect}\ A\ n) \to (\mathsf{vect}\ A\ n) \to (\mathsf{vect}\ A\ n)$$

8.1.1.3 Induction sur les vecteurs

Afin de pouvoir raisonner aisément sur les vecteurs, on est amené à construire des principes d'induction prenant en compte le fait que l'on ne peut additionner que des vecteurs de même taille. Ainsi pour montrer que

$$\forall n : \mathsf{nat}; v, w : (\mathsf{vect}\ A\ n).\ (\mathsf{addvect}\ v\ w) = (\mathsf{addvect}\ w\ v),$$

il est habituel de procéder par double induction sur v puis sur w pour destructurer ces deux vecteurs. On peut remplacer tout ce raisonnement par l'application du principe d'induction suivant :

$\forall P : \forall n : \mathsf{nat}.\ (\mathsf{vect}\ A\ n) \rightarrow (\mathsf{vect}\ A\ n) \rightarrow \mathsf{Prop}.$
$(P\ \mathsf{O}\ (\mathsf{vnil}\ A)\ (\mathsf{vnil}\ A)) \rightarrow$
$(\forall n : \mathsf{nat}; v, v' : (\mathsf{vect}\ A\ n).\ (P\ n\ v\ v') \rightarrow \forall a, b : A.\ (P\ (\mathsf{S}\ n)\ (\mathsf{vcons}\ A\ n\ a\ v)\ (\mathsf{vcons}\ A\ n\ b\ v'))) \rightarrow$
$\forall n : \mathsf{nat}; v, v' : (\mathsf{vect}\ A\ n).\ (P\ n\ v\ v')$

Ce théorème se démontre par induction sur n, puis en utilisant les propriétés du type vecteur données par les équations (8.1) et (8.2).

8.1.1.4 Opérations sur les matrices et propriétés d'anneau

Les matrices sont définies comme des vecteurs de vecteurs. Une matrice n, m est un vecteur à n composantes dont chacune est elle-même un vecteur de longueur m. En d'autres termes, cela signifie que les matrices sont définies ligne par ligne.

```
Definition Lmatrix := [n, m : nat]  (vect (vect A n) m).
```

Ensuite à partir des définitions des opérations sur les vecteurs, on définit les opérations sur les matrices. Par exemple, le produit de matrices se définit à l'aide du produit scalaire entre vecteurs. Nous obtenons finalement un module implantant l'anneau des matrices. La signature de ce module est donnée dans la figure 8.1.

8.1.2 Changements de Représentation

Dans cette section, nous étudions comment transformer la définition de la fonction map lors d'un changement de représentation. Supposons que l'on veuille prendre une nouvelle représentation des listes dépendantes sous forme de produit cartésien. Le type du produit cartésien de n fois A est défini par la fonction récursive suivante :

```
Fixpoint cp [A:Set;n:nat] : Set :=
Cases n of
| O    => unit
|(S m) => A*(cp A m)
end.
```

La transformation de la fonction map suit les mêmes règles que pour la fonction fact (voir le chapitre 7). Nous devons tout d'abord passer d'une définition par point-fixe à une définition par ordre bien fondé.

Nous allons voir comment modifier la définition de la fonction map en une définition bien fondée pour pouvoir en déduire une fonction map sur la représentation sous forme de produits cartésiens des vecteurs. Avec le type inductif des listes dépendantes, la fonction map s'écrit simplement avec la construction de point-fixe :

```
Fixpoint map [ A, B : Set; f : A ->  B; n : nat; v : (vect A n) ] :
 (vect B n) :=
 <[p : nat]  (vect B p)> Cases v of
     vnil => (vnil B)
    | (vcons p x v') => (vcons B p (f x) (map A B f p v'))
 end.
```

```
Module Type TMatrices.

Parameters A :  Set.
Parameters Aopp :  A ->  A.
Parameters Aplus : A -> A ->  A; Amult : A -> A ->  A.
Parameters A0 : A; A1 : A.
Parameters Aeq :  A -> A ->  bool.
Parameters A_ring : (Ring_Theory Aplus Amult A1 A0 Aopp Aeq).

Parameters Lmatrix : nat->nat->Set.
Parameters addmatrix : (n,m:nat)(Lmatrix n m)->(Lmatrix n m)->(Lmatrix n m).
Parameters prodmat : (n,m,p:nat)(Lmatrix m n)->(Lmatrix p m)->(Lmatrix p n).
Parameters oppmatrix : (n,m:nat)(Lmatrix n m)->(Lmatrix n m).
Parameters o: (n,m:nat)(Lmatrix n m).
Parameters I: (n : nat)(Lmatrix n n).

Axiom addmatrix_sym :
 (n,m:nat) (w,w':(Lmatrix n m)) (addmatrix n m w w')=(addmatrix n m w' w).

Axiom addmatrix_assoc : (n,m:nat) (w,w',w'':(Lmatrix n m))
(addmatrix n m (addmatrix n m w w') w'')=(addmatrix n m w (addmatrix n m w' w'')).

Axiom addmatrix_oppmatrix :
 (n,m:nat) (w:(Lmatrix n m)) (addmatrix n m w (oppmatrix n m w))=(o n m).

Axiom addmatrix_zero_l : (n,m:nat)(w:(Lmatrix n m))(addmatrix n m (o n m) w )=w.

Axiom I_mat: (m, n : nat) (w : (Lmatrix n m)) (prodmat m m n (I m) w) = w.
Axiom mat_I: (m, n : nat) (w : (Lmatrix n m)) (prodmat m n n w (I n)) = w.

Axiom prodmat_distr_l:
 (n, m, p : nat) (a, b : (Lmatrix m n)) (w : (Lmatrix p m))
  (prodmat n m p (addmatrix m n a b) w) =
  (addmatrix p n (prodmat n m p a w) (prodmat n m p b w)).

Axiom prodmat_distr_r:
 (n, m, p : nat) (a, b : (Lmatrix p m)) (w : (Lmatrix m n))
  (prodmat n m p w (addmatrix p m a b)) =
  (addmatrix p n (prodmat n m p w a) (prodmat n m p w b)).

Axiom prodmat_assoc:
 (n, m, p, q : nat) (a : (Lmatrix m n)) (b : (Lmatrix p m)) (c : (Lmatrix q p))
  (prodmat n p q (prodmat n m p a b) c) = (prodmat n m q a (prodmat m p q b c)).

End TMatrices.
```

FIG. 8.1 – La signature du foncteur des matrices en COQ

8.1.2.1 Accessibilité dépendante et opérateur de récursion associé

Nous allons définir une nouvelle relation d'accessibilité pour les types dépendants de la forme de vect. Ensuite, nous construisons par analogie avec la bibliothèque de propriétés sur l'ordre bien fondé WellFounded une nouvelle bibliothèque sur la récursion bien-fondée pour les types dépendants avec un seul degré de dépendance.

Accessibilité Nous commençons par définir une nouvelle relation d'accessibilité pour les types dépendants de la forme $\forall x : A.\ (B\ x)$. Cette définition est pratiquement la même que celle de l'accessibilité Acc de Coq pour les types simples. On a simplement remplacé A par $\forall x : A.\ (B\ x)$.

```
Variable A : Set.
Variable B : A ->  Set.
Variable R : (x : A) (B x) -> (y : A) (B y) ->  Prop.

Inductive Acc' : (x : A) (B x) ->  Prop :=
  Acc'_intro:
    (x : A)
    (y : (B x)) ((z : A) (t : (B z)) (R z t x y) ->  (Acc' z t)) ->  (Acc' x y) .
```

A partir de Acc', on définit la notion de relation bien fondée. Une relation R sur une famille $(B\ x)$ indexée par $x : A$ est bien-fondée si et seulement si tout élément y de type $(B\ x)$ est accessible.

```
Definition well_founded' := (x : A) (y : (B x))  (Acc' x y).
```

Opérateur de récursion associé A la suite de ces définitions, on définit un opérateur de récursion bien fondée suivant la relation R. Ce principe well_founded_induction' est défini par récursion structurelle sur la preuve d'accessibilité de l'entrée $y : (B\ x)$ avec $x : A$. Son type est le suivant :

$$\begin{array}{l}\forall A : \mathsf{Set}.\ \forall B : A \to \mathsf{Set}.\ \forall R : (\forall x : A.\ (B\ x) \to \forall y : A.\ (B\ y) \to \mathsf{Prop}).\\ (\mathsf{well\_founded'}\ A\ B\ R) \Rightarrow\\ \forall P : \forall x : A.\ (B\ x) \to \mathsf{Set}.\\ \boxed{\forall x : A.\ \forall y : (B\ x).\ (\forall z : A.\ \forall t : (B\ z)(R\ z\ t\ x\ y) \to (P\ z\ t)) \to (P\ x\ y)} \Rightarrow\\ \forall x : A.\ \forall y : (B\ x).\ (P\ x\ y)\end{array}$$

La partie encadrée correspond au type de la fonctionnelle à fournir pour construire une fonction récursive suivant l'ordre bien fondé R.

8.1.2.2 Définitions par ordre bien-fondé

Ordre Le premier pas vers une définition par ordre bien fondé consiste, tout d'abord, à choisir un ordre sur les données considérées (ici les vecteurs). L'ordre sera l'ordre lt sur la longueur des vecteurs. Grâce à la propriété wf_inverse_image transposée dans le cas dépendant, on montre facilement que la famille inductive vect peut être équipée avec un ordre bien-fondé.

```
Definition Vlength := [n : nat] [v : (vect n)]  n.

Definition Vlt :=
   [n : nat] [v : (vect n)] [m : nat] [w : (vect m)]
```

```
   (lt (Vlength n v) (Vlength m w)).

Theorem wf_Vlt: (well_founded' nat [n : nat]  (vect n) Vlt).
```

Construction de la fonctionnelle F_map La fonctionnelle F_map prend en arguments le type du vecteur d'entrée (A), le type du vecteur de sortie (B) et la fonction à appliquer à tous les éléments du vecteur initial pour construire le vecteur final. En plus de ces arguments, la fonctionnelle F_map prend en argument un vecteur vn de longueur n et une fonction rf permettant de faire les calculs récursifs pour tous les vecteurs qui sont plus petits que vn suivant l'ordre Vlt (c'est-à-dire dont la longueur est inférieure à celle de vn).

La définition de F_map se fait ensuite par analyse par cas sur vn. Si vn est de la forme $vnil$, alors on obtient (vnil B). Sinon, c'est-à-dire si l'entrée est de la forme (vcons $n0$ $a0$ $a0s$), la valeur retournée est construite par l'application du constructeur vcons avec pour tête le terme (f $a0$) et pour queue le terme calculé par rf pour la queue du vecteur donné en entrée.

```
Definition F_map := [A,B:Set; f:(A->B)]
[n:nat; vn:(vect A n)]
 <[n0:nat; v:(vect A n0)]
   ((z:nat; t:(vect A z))(Vlt A z t n0 v)->(vect B z))->(vect B n0)>
 Cases vn of vnil             => [rf:?](vnil B)
          | (vcons n0 a0 a0s) =>
                   [rf:?] (vcons B n0 (f a0) (rf n0 a0s (le_n (S n0))))
    end.
```

La nouvelle version wf_map de la fonction map est donnée par la définition suivante :

$$\begin{array}{l}\mathsf{Definition\ wf\_map}\ :\ \forall n : \mathsf{nat}.\ \forall v : (\mathsf{vect}\ A\ n).\ (\mathsf{vect}\ B\ n) := \\ \quad (\mathsf{well\_founded\_induction'}\ \ \mathsf{nat}\ \ \lambda n : \mathsf{nat}.\ (\mathsf{vect}\ A\ n)\ \ (\mathsf{Vlt}\ A)\ \ (\mathsf{wf\_Vlt}\ A) \\ \qquad \forall n : \mathsf{nat}.\ \forall v : (\mathsf{vect}\ A\ n).\ (\mathsf{vect}\ B\ n)\ \ \mathsf{F\_map})\end{array}$$

8.1.2.3 Preuve de l'équation de point-fixe

On souhaite disposer d'une équation de point-fixe pour pouvoir raisonner sur la fonction wf_map. Pour cela, on reprend les travaux sur les preuves d'équations de point-fixe pour les fonctions récursives générales dans le calcul des constructions. On reprend la démonstration du théorème de transfert donné en Coq à la fin de l'article de A. Balaa et Y. Bertot [BB00]. La généralisation à un type dépendant comme les vecteurs est assez facile. Seules les quelques parties de la démonstration faites automatiquement sont à reprendre. Étant un peu plus techniques, elles ne peuvent être montrées automatiquement dans Coq. Finalement, on obtient le théorème de transfert suivant :

```
Theorem new_wf_transfer:
 ((x : A)(v:(B x)) (f' : (y : A) (w: (B y)) (V y w))
          (g : (y : A)(w:(B y)) (R y w x v) -> (V y w))
          ((y : A)(w:(B y)) (h : (R y w x v))  (g y w h) = (f' y w))
   -> (F x v [y : A] [w:(B y)] [h : (R y w x v)]  (g y w h)) =
      (F x v [y : A] [w:(B y)] [_ : (R y w x v)]  (f' y w)))
 -> (x : A) (v:(B x)) (f x v) = (F x v [y : A] [w:(B y)] [h : (R y w x v)] (f y w)).
```

Grâce à ce théorème, on obtient l'équation de point-fixe associée à la définition de wf_map. Finalement, on en déduit les inégalités suivantes :

$(\mathsf{wf\_map}\ \mathsf{O}\ (\mathsf{vnil}\ A)) = (\mathsf{vnil}\ B)$
$\forall n : \mathsf{nat}.\ \forall a : A.\ \forall v : (\mathsf{vect}\ A\ n).\ (\mathsf{wf\_map}\ (\mathsf{S}\ n)\ (\mathsf{vcons}\ A\ n\ a\ v)) = (\mathsf{vcons}\ B\ n\ (f\ a)\ (\mathsf{wf\_map}\ n\ v))$

8.2 Utilisation d'isomorphismes au niveau des types

Dans cette section, nous présentons une approche alternative à la méthode décrite dans ce document pour gérer les changements de représentation concrète des données dans une théorie développée formellement.

8.2.1 Exemple

Nous allons montrer sur l'exemple des entiers naturels comment passer d'une représentation à une autre des données en utilisant directement l'isomorphisme les reliant.

On commence par donner sous forme de paramètres les homologues en binaire des opérations de base sur les entiers de Peano.

```
Parameter bin:Set.
Parameter b0:bin.
Parameter bS: bin->bin.
Parameter bpred : bin->bin.
Parameter bplus:bin->bin->bin.
Parameter bmult:bin->bin->bin.
Parameter bminus:bin->bin->bin.
```

Ces paramètres peuvent très bien être instantiés par les opérations décrites dans le chapitre précédent. A partir de cette instantiation, on peut définir des fonctions de conversion n2b et b2n et démontrer qu'elles sont inverses l'une de l'autre :

$$\forall n : \mathsf{nat}.\ (\mathsf{b2n}\ (\mathsf{n2b}\ n)) = n \qquad \forall n : \mathsf{bin}.\ (\mathsf{n2b}\ (\mathsf{b2n}\ n)) = n$$

L'étape suivante consiste à établir des propriétés de morphismes pour les opérations sur les entiers O, S, pred, plus, mult, minus...

$\mathsf{morphism\_O} : \mathsf{b0} = (\mathsf{n2b}\ \mathsf{O})$
$\mathsf{morphism\_S} : \forall n : \mathsf{nat}.\ (\mathsf{bS}\ (\mathsf{n2b}\ n)) = (\mathsf{n2b}\ (\mathsf{S}\ n))$
$\mathsf{morphism\_pred} : \forall n : \mathsf{nat}.\ (\mathsf{bpred}\ (\mathsf{n2b}\ n)) = (\mathsf{n2b}\ (\mathsf{pred}\ n))$
$\mathsf{morphism\_plus} : \forall n, m : \mathsf{nat}.\ (\mathsf{bplus}\ (\mathsf{n2b}\ n)\ (\mathsf{n2b}\ m)) = (\mathsf{n2b}\ (\mathsf{plus}\ n\ m))$
$\mathsf{morphism\_mult} : \forall n, m : \mathsf{nat}.\ (\mathsf{bmult}\ (\mathsf{n2b}\ n)\ (\mathsf{n2b}\ m)) = (\mathsf{n2b}\ (\mathsf{mult}\ n\ m))$
$\mathsf{morphism\_minus} : \forall n, m : \mathsf{nat}.\ (\mathsf{bminus}\ (\mathsf{n2b}\ n)\ (\mathsf{n2b}\ m)) = (\mathsf{n2b}\ (\mathsf{minus}\ n\ m))$

Dans la suite, on suppose que tous ces théorèmes sont stockées dans une base de réécriture appelé morphisms.

Imaginons que l'on cherche à traduire la preuve de distributivité de la multiplication sur l'addition.

$$\mathsf{mult\_plus\_distr} : \forall n, m, p : \mathsf{nat}.(\mathsf{mult}\ (\mathsf{plus}\ n\ m)\ p) = (\mathsf{plus}\ (\mathsf{mult}\ n\ p)\ (\mathsf{mult}\ m\ p))$$

Dans la représentation binaire, l'énoncé du théorème s'écrit :

$$\forall n, m, p : \mathsf{bin}.(\mathsf{bmult}\ (\mathsf{bplus}\ n\ m)\ p) = (\mathsf{bplus}\ (\mathsf{bmult}\ n\ p)\ (\mathsf{bmult}\ m\ p))$$

On va chercher à transformer cet énoncé en une instance du théorème mult_plus_distr. Pour cela, on va remplacer chacune des occurrences de variables v de type bin présentes dans le contexte par (n2b (b2n v)). Ensuite à l'aide des règles de morphismes données plus haut, on va transformer l'énoncé du théorème en la formule suivante (on suppose que l'on a introduit tous les produits) :

$$(\mathsf{n2b}\ (\mathsf{mult}\ (\mathsf{plus}\ (\mathsf{b2n}\ n)\ (\mathsf{b2n}\ m))\ (\mathsf{b2n}\ p))) = (\mathsf{n2b}\ (\mathsf{plus}\ (\mathsf{mult}\ (\mathsf{b2n}\ n)\ (\mathsf{b2n}\ p))\ (\mathsf{mult}\ (\mathsf{b2n}\ m)\ (\mathsf{b2n}\ p))))$$

A ce moment-là, on a besoin de la propriété que n2b est une fonction, à savoir

$$\mathsf{f\_equal\_n2b} : \forall n, m : \mathsf{nat}.\ n = m \rightarrow (\mathsf{n2b}\ n) = (\mathsf{n2b}\ m).$$

De là, on réduit le but à :

$$(\mathsf{mult}\ (\mathsf{plus}\ (\mathsf{b2n}\ n)\ (\mathsf{b2n}\ m))\ (\mathsf{b2n}\ p)) = (\mathsf{plus}\ (\mathsf{mult}\ (\mathsf{b2n}\ n)\ (\mathsf{b2n}\ p))\ (\mathsf{mult}\ (\mathsf{b2n}\ m)\ (\mathsf{b2n}\ p)))$$

Ce but est une instance du théorème mult_plus_distr, il suffit donc d'appliquer ce théorème pour conclure.

En Coq, la preuve complète est donnée par :

```
Lemma new_mult_plus_distr :
   (n,m,p:bin)(bmult (bplus n m) p)=(bplus (bmult n p) (bmult m p)).
Intros n m p.
Rewrite <- (n2b_b2n n).
Rewrite <- (n2b_b2n m).
Rewrite <- (n2b_b2n p).
Repeat Rewrite morphism_mult.
Repeat Rewrite morphism_plus.
Rewrite morphism_mult.
Apply f_equal_n2b.
Apply mult_plus_distr.
Qed.
```

8.2.2 Démonstration automatique de propriétés

A partir de l'exemple précédent, il est facile de voir que l'on peut automatiser le procédé utilisé. Pour cela, nous allons utiliser le langage de tactiques proposé par David Delahaye [Del00, Del01] pour Coq. Nous allons construire une tactique IsoSolve qui fonctionne de la manière suivante :

1. On commence par écrire une tactique VIntro qui introduit la première variable liée et effectue quelques pas de réécriture. Quand le but est de la forme $\forall b$: bin. ..., on introduit la variable b et on réécrit de droite à gauche avec le terme (n2b_b2n b) qui exprime que (n2b (b2n b)) = b. Ensuite on applique la tactique de réécriture automatique AutoRewrite sur les morphismes (morphism_plus...).

   Dans le cas d'un produit $\forall v : V.\ T$ où V est différent de bin, on se contente d'introduire cette nouvelle variable sans aucun autre traitement.

```
Meta Definition GrepHypId trm :=
Match Context With | [ id:trm |- ? ] -> id.

Tactic Definition VIntro :=
Match Context With
 | [|-(id:bin)?] ->
   Intro ; Let v = (GrepHypId bin) In Rewrite <- (n2b_b2n v) ;
   AutoRewrite [morphisms]
 | [|-(id:?)?] -> Intro.
```

2. A partir de la définition de VIntro, on définit la tactique de résolution des buts IsoSolve, cette tactique commence par essayer de remplacer le symbole b0 par (n2b O) dans le but. Ensuite on itère l'application de VIntro jusqu'à ce qu'il ne reste plus de variables à introduire. Il ne reste qu'à appliquer la tactique Auto avec pour arguments les bases arith et transform. La base arith contient la représentation en arithmétique de Peano du théorème que l'on cherche à montrer. Cette tactique de démonstration automatique utilise aussi la base transform qui contient les théorèmes f_equal_n2b et n2b_one_to_one.

```
Tactic Definition IsoSolve :=
Try (Rewrite morphism_O) ;
Repeat VIntro ;
Auto with arith transform.
```

Cette tactique a permis de démontrer automatiquement de nombreux énoncés de la bibliothèque Arith traduits vers la représentation binaire des entiers. Nous verrons cependant qu'il l'enrichir pour pouvoir traduire la majeure partie de cette bibliothèque.

Traduction des énoncés La traduction des énoncés se fait au niveau du code ML. On utilise exactement la même méthode de substitution des symboles que dans le cas de la traduction structurelle des termes de preuve d'une représentation à une autre.

```
Hint Rewrite [ morphism_O morphism_S morphism_pred ] in morphisms.
Hint Rewrite [ morphism_plus morphism_mult morphism_minus ] in morphisms.

Axiom n2b_one_to_one : (n,m:nat)(n2b n)=(n2b m)->n=m.

Lemma f_equal_n2b : (x,y:nat)x=y->(n2b x)=(n2b y).
Exact(f_equal ? ? n2b).
Qed.
```

Démonstrations Il est généralement nécessaire de transformer, en plus de la conclusion, les prémisses d'un but traduit vers l'arithmétique binaire afin de se ramener à une instance d'un théorème connu dans l'arithmétique de Peano. Pour cela, nous avons développé des tactiques plus sophistiquées comme StrongIsoSolve. Avant de décrire le corps de cette nouvelle tactique, on commence par ajouter deux nouveaux théorèmes permettant de relier ble et le dans la base transform.

```
Theorem ble_n2b : (n,m:?)(le n m) -> (ble (n2b n) (n2b m)).
Theorem ble_b2n : (n,m:?)(ble (n2b n) (n2b m))->(le n m).
Hints Resolve ble_n2b ble_b2n : transform.
```

La tactique StrongVIntros transforme les prémisses de l'énoncé traduit (celui que l'on veut démontrer) en les prémisses de l'énoncé initial. La tactique StrongIsoSolve consiste simplement à répéter les transformations sur toutes les variables au fur et à mesure qu'on les introduit.

```
Tactic Definition StrongVIntros :=
Match Context With
| [id:(ble (n2b ?1) (n2b ?2))|-?] ->
  Generalize (ble_b2n ?1 ?2 id); Clear id; Intro
| [id:(bge (n2b ?1) (n2b ?2))|-?] ->
  Generalize (bge_b2n ?1 ?2 id); Clear id; Intro
| [id:(blt (n2b ?1) (n2b ?2))|-?] ->
  Generalize (blt_b2n ?1 ?2 id); Clear id; Intro
| [id:(bgt (n2b ?1) (n2b ?2))|-?] ->
  Generalize (bgt_b2n ?1 ?2 id); Clear id; Intro.

Tactic Definition StrongIsoSolve :=
Try (Rewrite morphism_O) ;
Repeat VIntros ;
Repeat StrongVIntros ;
Auto with arith transform.
```

Les tactiques IsoSolve et StrongIsoSolve ont deux homologues EIsoSolve et EStrongIsoSolve qui utilisent la tactique EAuto plutôt que la tactique Auto pour terminer la preuve. Le corps de la tactique StrongVIntros doit être étendu à chaque fois que l'on ajoute un nouveau prédicat dans la théorie initiale.

De plus, au fur et à mesure des expériences, il s'est avéré nécessaire de parcourir structurellement les énoncés à transformer. Par exemple, la preuve d'une disjonction de la forme $(\mathsf{lt}\ n\ m) \vee n = m$ nécessite de destructurer la disjonction, appliquer les morphismes et ensuite reconstruire la disjonction. Cette méthode nécessite donc une analyse structurelle du type du théorème de la même manière qu'il était nécessaire de faire une analyse structurelle du terme de preuve pour le traduire dans la méthode présentée plus tôt dans ce document.

Implantation dans Coq Le procédé de traduction des énoncés et de démonstration automatique des nouveaux énoncés est implanté dans le système Coq en utilisant le langage de tactiques proposé par David Delahaye [Del00, Del01] pour Coq. Ce langage a pu être utilisé dès lors que l'analyse du contexte par Match Context With se faisait de l'hypothèse la plus récemment introduite vers l'hypothèse la plus ancienne (c'est le cas dans la version 7.4. de Coq disponible depuis février 2003).

Comme nous l'avons vu, seule la traduction syntaxique des énoncés des théorèmes d'un type de données à l'autre a nécessité l'écriture de code directement en Caml. Cet outil a été expérimenté avec succès pour traduire une centaine de théorèmes de la bibliothèque Arith de Coq. Il apparaît *a priori* comme une alternative sérieuse (mais de complexité équivalente dans sa mise en œuvre) à la technique de transformation des termes de preuve présentée dans cette thèse.

8.3 Conclusion

Dans ce chapitre, nous avons étudié un cas où les données sont représentées par des types dépendants. Dans ce cadre, nous avons montré comment les techniques de réinstantiation des pa-

ramètres par des fonctions effectives peuvent être réutilisées en présence de types dépendants. Nous avons seulement étudié le cas d'un type dépendant avec un seul niveau de dépendances. La question de la généricité de cette méthode pour des types dépendants arbitraires reste ouverte et mériterait d'être étudiée. De plus, la redéfinition des fonctions dans une nouvelle représentation est seulement une étape du travail de transformation d'une théorie développée formellement lors du changement de la représentation concrète d'une des données mises en jeu. L'application des techniques proposées dans les deux chapitres précédents dans le cas des types dépendants reste à étudier.

En plus de cette extension aux types dépendants de nos travaux, nous avons également décrit une approche alternative pour la réutilisation des preuves dans le cas d'un changement de représentation de certaines données. Cette approche apparaît comme une alternative crédible à la méthodologie de transformation des termes de preuve proposée dans ce document.

Chapitre 9

Conclusions

Nous voilà arrivés au terme de ce document. Nous allons en rappeler les principales contributions. Nous donnerons ensuite quelles perspectives de recherche issues de ces travaux.

9.1 Travaux accomplis

Dans cette thèse, nous avons réalisé deux études sur le développement, la maintenance et la réutilisation de preuves formelles dans un système d'aide à la preuve comme Coq. Nous avons fourni une preuve complète à plusieurs niveaux d'abstraction de la correction de l'algorithme de calcul de la racine carrée de Gmp ainsi que de son implantation impérative en C. Nous avons aussi proposé de nouveaux outils pour faciliter le passage d'une représentation inductive à une autre d'un type de données. Pour cela, nous avons proposé un algorithme de transformation des termes de preuves, permettant d'extraire les étapes implicites de raisonnement par calcul et de les rendre explicites par l'application de théorèmes d'égalités simulant ces réductions. Après avoir prouvé à la main les propriétés caractéristiques des fonctions dans la nouvelle représentation, on dispose d'un outil automatique pour traduire les preuves de l'ancienne vers la nouvelle représentation.

9.2 Perspectives de recherche

Les travaux réalisés sur les changements de représentation des données dans une théorie développée formellement s'appuient sur l'isomorphisme existant entre les deux représentations inductives considérées. D'un autre côté, le passage de la description fonctionnelle à la description impérative du programme de calcul de la racine carrée de Gmp peut aussi être vu comme un changement de représentation. En effet, les données manipulées ne sont plus des entiers relatifs $\mathbb{Z}$, mais des entiers bornés.

9.2.1 Types de données non isomorphes

On peut noter trois contextes dans lesquels on pourrait vouloir changer de représentation sans pour autant que les deux types sont sémantiquement isomorphes :

- le passage des entiers relatifs $\mathbb{Z}$ vers les entiers bornés $\mathbb{Z}_n$,
- le passage des entiers naturels $\mathbb{N}$ vers les entiers relatifs $\mathbb{Z}$,
- le passage des nombres réels vers les nombres flottants.

9.2.1.1 Des entiers relatifs vers les entiers bornés

Lors d'un passage de la description fonctionnelle à la description impérative, nous avons non seulement changé de paradigme de programmation, mais aussi la représentation des données manipulées par l'algorithme. Alors que dans la description fonctionnelle, les entiers étaient représentés par l'ensemble $\mathbb{Z}$ des entiers relatifs, ils sont représentés sous forme de segments d'entiers bornés. Cette transformation peut donc être vue comme un changement de représentation. Cependant, les deux représentations considérés : l'ensemble des entiers relatifs $\mathbb{Z}$ et l'ensemble des segments d'entiers bornés que l'on peut identifier à $\mathbb{Z}_n$ pour un n donné ne sont pas isomorphes. Par conséquent les énoncés de théorèmes dans la première représentation $\mathbb{Z}$ doit être précisé avant de pouvoir être démontrées dans la nouvelle représentation. L'énoncé sur $\mathbb{Z}$ ne doit traiter que des entiers représentables dans *l'ensemble d'arrivée* de la transformation.

9.2.1.2 Des entiers naturels vers les entiers relatifs

On se retrouve confronté au même problème lorsque l'on souhaite passer de la représentation à la Peano des entiers naturels à la représentation binaire (sous forme d'entiers relatifs $\mathbb{Z}$). Par exemple, l'énoncé du théorème plus_sym, à savoir

$$\forall n, m : \mathsf{nat}.\ (\mathsf{plus}\ n\ m) = (\mathsf{plus}\ m\ n)$$

doit être transformé en

$$\forall n, m : \mathbb{Z}.\ 0 \leq n \rightarrow 0 \leq m \rightarrow (\mathsf{Zplus}\ n\ m) = (\mathsf{Zplus}\ m\ n)$$

En effet, la preuve de plus_sym ne contient aucune information à propos du comportement de l'addition sur les entiers négatifs. Cette modification des énoncés avec des pré-conditions de positivité sur les entiers est assez proche de ce que l'on a fait en considérant le type Zpos à la place de nat. Dans ce cas, les conditions de positivité étaient cachées dans un Σ-type.

9.2.1.3 Des réels vers les flottants

Passer de la description mathématique d'un problème à sa réalisation pratique revient souvent à passer d'un univers abstrait (où tout peut être décrit de façon exacte) à un univers concret (où certains objets ne peuvent être décrits qu'approximativement). C'est le cas lors du passage de la représentation mathématique des nombres (généralement des nombres réels) à la représentation informatique de ces nombres (généralement des flottants). Les flottants sont généralement représentés par des mots de 32 ou 64 bits. Par conséquent, ils ne permettent pas toujours de représenter de façon exacte tous les nombres réels. Certains nombres réels (qu'ils soient trop grands ou trop petits) ne peuvent être représentés qu'approximativement.

Pourtant on aimerait pouvoir utiliser les nombreux résultats établis sur les réels (par exemple dans [May01]) pour démontrer des propriétés d'algorithmes sur les flottants à l'aide du système Coq. Pour cela, il faudrait développer des outils de transformation de démonstrations dans le cas où les représentations initiale et finale ne sont pas isomorphes.

9.2.2 Représentation des fonctions

Nous avons vu dans ce document que certaines fonctions comme la factorielle et la fonction de Fibonacci doivent être définies en utilisant une construction par ordre bien-fondé. Dans sa thèse [Bal02], A. Balaa propose une méthode alternative à la définition par ordre bien-fondé des fonctions

récursives générales. Elle propose de faire des définitions par itération dont le nombre d'itérations est bornée. Cette méthode apparaît comme plus facilement utilisable dans le système Coq. Il serait donc intéressant d'étudier comment décrire les fonctions avec ce nouveau cadre et comment changer la représentation des données dans le corps de ces fonctions.

9.2.3 Types dépendants

A la fin de cette thèse, nous avons abordé le cas d'un changement de représentation d'une donnée décrite par un type dépendant. Il serait intéressant de poursuivre cette étude en particulier en étudiant les problèmes issus du changement de représentation des indices d'une famille inductive. Par exemple, le type des vecteurs pourrait être transformé afin de remplacer le type nat par le type bin. Dans ce cas, l'énoncé

$$\forall v : (\mathsf{vect}\ A\ n).\ \forall v' : (\mathsf{vect}\ A\ (\mathsf{pred}\ (\mathsf{S}\ n))).\ v =_{(\mathsf{vect}\ A\ n)} v'$$

est bien formé puisque (pred (S n)) et n sont convertibles dans nat. Cependant, dans la représentation binaire, (bpred (bS n)) et n ne sont pas convertibles. Par conséquent deux termes égaux propositionnellement ne sont plus dans le même type après la transformation de nat vers bin. Une solution (hors de portée au sein du système Coq actuel) serait d'identifier les égalités propositionnelle et définitionnelle comme l'a proposé M. Hofmann dans [Hof96]. Une autre technique consisterait à relier entre elles deux instances d'une même famille inductive par des cœrcions [Saï97]. Z. Luo et S. Soloviev ont d'ailleurs proposé dans [LS99] des moyens d'ajouter des cœrcions entre un type (par exemple les listes polymorphes) et une famille de types inductifs dépendants (par exemple les vecteurs de longueur n).

Bibliographie

[Alv02] C. Alvarado. *Réflexion pour la Réécriture dans le Calcul des Constructions Inductives*. PhD thesis, Université Parix XI - Orsay, 2002.

[And94] P. Anderson. Representing Proof Transformations for Program Optimization. In *Proceedings of the 12th International Conference on Automated Deduction*, volume 814, pages 575–589. LNAI, Springer-Verlag, 1994.

[Bal02] A. Balaa. *Fonctions récursives générales dans le calcul des constructions*. PhD thesis, Université de Nice Sophia-Antipolis, 2002.

[Bar95] G. Barthe. Implicit Coercions in Type Systems. In S. Berardi and M. Coppo, editors, *Proceedings of Types'95*, volume 1128 of *LNCS*, pages 1–15, 1995.

[BB00] A. Balaa and Y. Bertot. Fix-point Equations for Well-Founded Recursion in Type Theory. In Harrison and Aagaard [HA00], pages 1–16.

[BC03] Y. Bertot and P. Casteran. *Le Coq'Art*. To appear, 2003.

[Bla01] F. Blanqui. *Théorie des Types et Récriture*. PhD thesis, Université Paris XI, Orsay, France, 2001. Available in english as "Type theory and Rewriting".

[BMZ02] Y. Bertot, N. Magaud, and P. Zimmermann. A Proof of GMP Square Root. *Journal of Automated Reasoning*, 29 :225–252, 2002. Special Issue on Automating and Mechanising Mathematics : In honour of N.G. de Bruijn.

[Bon02] D. Bondyfalat. Certification d'un Algorithme de Division pour les Grands Entiers. Unpublished, 2002.

[Bou97] S. Boutin. Using Reflection to Build Efficient and Certified Decision Procedures. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Science*, volume 1281. LNCS, Springer-Verlag, 1997.

[BP01] G. Barthe and O. Pons. Type Isomorphisms and Proof Reuse in Dependent Type Theory. In F. Honsell and M. Miculan, editors, *FOSSACS'01*, volume 2030, pages 57–71. LNCS, Springer-Verlag, 2001.

[BRB95] G. Barthe, M.P.J. Ruys, and H.P. Barendregt. A Two-Level Approach Towards Lean Proof-Checking. In *Types for Proofs and Programs, International Workshop TYPES'95*, pages 16–35, Torino, 1995.

[CAB+86] R. L. Constable, S. F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D. J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986. freely available from http ://www.cs.cornell.edu/Info/Projects/NuPrl/book/doc.html.

[Cap99] V. Capretta. Universal Algebra in Type Theory. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, volume 1690 of *LNCS*, pages 131–148. Springer, 1999.

[CH88] Th. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76, 1988.

[CLMP00] P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors. volume 2277 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[Coq02] Coq development team, INRIA and LRI. *The Coq Proof Assistant Reference Manual*, May 2002. Version 7.3.

[Cos00] Y. Coscoy. *Explication Textuelle de Preuves pour le Calcul des Constructions Inductives*. Thèse d'université, Université de Nice-Sophia-Antipolis, September 2000.

[Cre02] J. Creci. Certification d'algorithmes d'arithmétique réelle exacte dans le système Coq. Rapport de DEA, Université Paris-Sud, Orsay, France, September 2002.

[CT96] C. Cornes and D. Terrasse. Inverting Inductive Predicates in Coq. In *BRA Workshop on Types for Proofs and Programs (TYPES'95)*, volume 1158 of *LNCS*, 1996.

[Cur95] R. Curien. *Outils pour la Preuve par Analogie*. PhD thesis, Université Henri Poincaré Nancy I, INRIA-Lorraine, 1995.

[Del00] D. Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island*, volume 1955, pages 85–95. Springer-Verlag LNCS/LNAI, November 2000.

[Del01] D. Delahaye. *Conception de Langages pour Décrire les Preuves et les Automatisations dans les Outils d'Aide à la Preuve*. PhD thesis, Université Paris 6, 2001.

[Den00] E. Denney. A Prototype Proof Translator from HOL to Coq. In Harrison and Aagaard [HA00], pages 108–125.

[DGJ03] C. Dubois, J. Grandguillot, and M. Jaume. Réutilisation de Preuves Formelles : une Etude pour le Système FoC. In *Journées Francophones pour les Langages Applicatifs*, January 2003.

[DR02] D. Deharbe and S. Ranise. BDD-Driven First-Order Satisfiability Procedures. Rapport technique 4630, INRIA, Novembre 2002.

[DRT01] M. Daumas, L. Rideau, and L. Théry. A Generic Library for Floating-Point Numbers and Its App lication to Exact Computing. In *Theorem Proving in Higher Order Logics : 14th International Conference*, number 2152 in LNCS. Springer-Verlag, September 2001.

[FH97] A. Felty and D. Howe. Hybrid Interactive Theorem Proving using Nuprl and HOL. In *Fourteenth International Conference on Automated Deduction*, volume 1249 of *LNCS*. Springer, 1997.

[Fil99] J.-C. Filliâtre. *Preuve de Programmes Impératifs en Théorie des Types*. Thèse, Université Paris-Sud, July 1999.

[Fil01a] J.-C. Filliâtre. Formal Proof of a Program : Find. *Science of Computer Programming*, 2001. To appear.

[Fil01b] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 2001. English translation of [Fil99]. To appear.

[Fil03] J.-C. Filliâtre. Why : a multi-language multi-prover verification tool. Submitted to FME 2003, March 2003.

[FM99] J.-C. Filliâtre and N. Magaud. Certification of Sorting Algorithms in the Coq System. In *Theorem Proving in Higher Order Logics : Emerging Trends*, 1999.

[Göd58] K. Gödel. Über eine bisher noch nicht benutze erweitrung des finiten standpunktes. *Dialectica*, 12, 1958.

[Göd86] K. Gödel. *Collected Works, Volumes I and II.* Oxford University Press, 1986.

[GF03] Groupe Foc. Présentation du Projet foc, 2003. http ://www-caslfor.lip6.fr/ foc.

[Gim98] E. Giménez. A Tutorial on Recursive Types in Coq. Rapport technique 221, INRIA, May 1998. disponible à http ://coq.inria.fr/doc-eng.html.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL : a theorem proving environment for higher-order logic.* Cambridge University Press, 1993.

[Gra02] T. Granlund. *The GNU Multiple Precision Arithmetic Library*, 2002. Edition 4.0.1.

[GWZ00] H. Geuvers, F. Wiedijk, and J. Zwanenburg. Equational reasoning via partial reflection. In Harrison and Aagaard [HA00], pages 163–179.

[HA00] J. Harrison and M. Aagaard, editors. *Theorem Proving in Higher Order Logics : 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[Har98] J. Harrison. HOL Light : A Small and Idealistic, Yet Fairly Powerful, Theorem Prover, 1998. available at http ://www.cl.cam.ac.uk/users/jrh/hol-light/.

[Har99] J. Harrison. A machine-checked theory of floating point arithmetic. In *Theorem Proving in Higher Order Logics : 12th International Conferen ce*, number 1690 in LNCS. Springer-Verlag, September 1999.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10) :576–580, 1969.

[Hof96] M. Hofmann. Conservativity of Equality Reflection over Intensional Type Theory. In *BRA Workshop on Types for Proofs and Programs (TYPES'95)*, volume 1158, pages 153–165. Springer-Verlag LNCS, 1996.

[Inf92] Information Management and Technology Division. Patriot Missile Defense : Software Problem led to System Failure at Dhahran, Saudi Arabia. Report B-247094, United States General Accounting Office, 1992.

[Jac01] C. Jacobi. Formal verification of a theory of IEEE rounding. In Richard J. Boulton and Paul B. Jackson, editors, *TPHOLs 2001 : Supplemental Proceedings*, 2001. Informatics Research Report EDI-INF-RR-0046, Univ. Edinburgh, UK.

[KMM00a] M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning : ACL2 Case Studies.* Kluwer Academic Publishers, June 2000. ISBN 0-7923-7849-0.

[KMM00b] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning : An Approach.* Kluwer Academic Publishers, June 2000. ISBN 0-7923-7744-3.

[L+96] J.-L. Lions et al. Ariane 5 Flight 501 Failure Report by the Inquiry Board. Technical report, European Space Agency, Paris, France, 1996.

[Loi97] P. Loiseleur. Formalisation en Coq de la Norme IEEE 754 sur l'Arithmétique à virgule flottante. Master's thesis, LIP, ENS Lyon, 1997.

[LP92] Z. Luo and R. Pollack. LEGO proof development system : User's manual. Technical Report ECS-LFCS-92-211, LFCS, Computer Science Dept., University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, May 1992. Updated version. Available by anonymous ftp with LEGO distribution.

[LS99] Z. Luo and S. Soloviev. Dependent Coercions. In *8th Inter. Conf. on Category Theory in Computer Science (CTCS'99)*, volume 29 of *Electronic Notes in Theoretical Computer Science*, pages 23–34, Edinburgh, Scotland, 1999. Elsevier.

[Luo99] Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1) :105–130, 1999.

[Mad89] P. Madden. The specialization and transformation of constructive existence proof s. In N.S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 131–148. Morgan Kaufmann, 1989. Also available from Edinburgh as DAI Research Paper 416.

[Mag95] L. Magnusson. *The Implementation of ALF - a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitutions*. PhD thesis, Chalmers University of Technology / Göteborg University, 1995.

[Mag03] N. Magaud. Changing Data Representation within the Coq System. In *TPHOLs'2003*, volume To appear. LNCS, Springer-Verlag, 2003.

[Map03] Maplesoft. Maple 9, 2003. http ://www.maplesoft.com/.

[May01] M. Mayero. *Formalisation et Automatisation de Preuves en Analyses Réelle et Numérique*. PhD thesis, Université Paris 6, 2001.

[MB00] N. Magaud and Y. Bertot. Changing Data Structures in Type Theory :A Study of Natural Numbers. In Callaghan et al. [CLMP00], pages 181–196.

[MB01] N. Magaud and Y. Bertot. Changement de représentation des structures de données en Coq : le cas des entiers naturels. In P. Casteran, editor, *Journées Francophones pour les Langages Applicatifs*, January 2001. Egalement disponible comme rapport de recherche INRIA RR-4039.

[McB99] C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[Min95] P.S. Miner. Defining the IEEE-854 Floating-Point Standard in PVS. NASA Technical Memorandum 110167, NASA Langley Research Center, Hampton, Virginia, June 1995.

[Miq02] A. Miquel. Axiom $\forall P, Q : Prop.$ (P $\leftrightarrow$ Q)$\rightarrow$(P==Q) is safe. Communication in the coq-club list, November 2002.

[MM94] V. Ménissier-Morain. *Arithmétique Exacte, Conception, Algorithmique et Performances d'une Implémentation Informatique en Précision Arbitraire*. Thèse, Université Paris 7, December 1994.

[MM04] C. McBride and J. McKinna. The View from the Left. *Journal of Functional Programming*, 14 :1–43, 2004. To appear.

[Mul95] J.-M. Muller. Algorithmes de Division pour Microprocesseurs : Illustration à l'aide du "Bug" du Pentium. *Technique et Science Informatiques*, 14(8), 1995.

[MW99] E. Melis and J. Whittle. Analogy in inductive theorem proving. *Journal of Automated Reasoning*, 22 :117–147, 1999.

[NPS90] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory : An Introduction*. Oxford University Press, 1990.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[PM93] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, 1993. LIP research report 92-49.

[PM96] C. Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, Décembre 1996.

[Pol99] PolKA project, INRIA Lorraine and LORIA. *The Mulitple Precision Floating-Point Reliable Library*, June 1999. Edition 1.0.

[Pon99] O. Pons. *Conception et Réalisation d'Outils d'Aide au Développement de Grosses Théories dans les Systèmes de Preuves Interactifs*. PhD thesis, Conservatoire National des Arts et Métiers, 1999.

[Pon00] O. Pons. Generalization in Type Theory based Proof Assistants. In Callaghan et al. [CLMP00], pages 217–232.

[Ric95] J. Richardson. Automating Changes of Data Type in Functional Programs. In *Proceedings of KBSE-95, Boston USA, 12-15 November 1995*. IEEE Computer Society, 1995. Extended version available from Edinburgh as DAI Research Paper 767.

[RR03] C. Raffalli and P. Rozière. The PhoX Proof Checker Documentation. Draft version 0.83, 2003. available at http ://www.lama.univ-savoie.fr/˜RAFFALLI/phox.html.

[Rus99] D. M. Russinoff. A Mechanically Checked Proof of IEEE Compliance of A MD K5 Floating Point Square-Root Microcode. *Formal Methods In System Design*, 14(1) :75–125, January 1999.

[Saï97] A. Saïbi. Typing Algorithm in Type Theory with Inheritance. In *POPL'97*. ACM, 1997.

[SG02] J. Sawada and R. Gamboa. Mechanical Verification of a Square Root Algorithnm Using Taylor's Theorem. In *Proceedings of FMCAD 2002*, volume 2517, pages 274–291. Springer-Verlag LNCS, 2002.

[SOR93] N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93 : Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.

[Tho91] S. Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.

[Wad87] P. Wadler. Views : A Way for Pattern Matching to Cohabit with Data Abstraction. In *14th Symposium on Principles of Programming Languages POPL'87*. ACM, 1987.

[Wol03] Wolfram. Mathematica 5, 2003. http ://www.wolfram.com/.

[Zim99] P. Zimmermann. Karatsuba Square Root. Technical Report 3805, INRIA, november 1999.

[Zim00] P. Zimmermann. A Proof of GMP Fast Division and Square Root Implementations. Unpublished, september 2000.

[Zur94] D. Zuras. More On Squaring and Multiplying Large Integers. *IEEE Transactions on Computers*, 43(8) :899–908, 1994.

Résumé

Nous étudions comment faciliter la réutilisation des preuves formelles en théorie des types. Nous traitons cette question lors de l'étude de la correction du programme de calcul de la racine carrée de GMP. A partir d'une description formelle, nous construisons un programme impératif avec l'outil Correctness. Cette description prend en compte tous les détails de l'implantation, y compris l'arithmétique de pointeurs utilisée et la gestion de la mémoire.
Nous étudions aussi comment réutiliser des preuves formelles lorsque l'on change la représentation concrète des données. Nous proposons un outil qui permet d'abstraire les propriétés calculatoires associées à un type inductif dans les termes de preuve. Nous proposons également des outils pour simuler ces propriétés dans un type isomorphe. Nous pouvons ainsi passer, systématiquement, d'une représentation des données à une autre dans un développement formel.

Mots clés : Outils d'aide à la preuve, théorie des types, racine carrée, preuves formelles de programmes, GMP, arithmétique en précision arbitraire, types inductifs, transformation automatique de preuves.

Abstract

We study proof reuse in a type theory. We investigate this issue by studying the correctness of GMP square root. From a formal description, we build a complete imperative program using the Correctness tool. This description gives a detailed account of the actual implementation, including pointer arithmetics and memory management.
We also study how to reuse formal proofs when the concrete representation of some data changes. We propose tools to abstract away computational properties of a given inductive data type in proof terms. We also propose new tools to simulate these properties in an isomorphic data type. Together, these tools allow us to shift, almost automatically, from one representation to another in a formal development.

Keywords : Proof assistants, type theory, square root, formal proofs of programs, GMP, arbitrary large numbers, inductive types, automatic proof transformation.