HABILITATION À DIRIGER DES RECHERCHES

présentée

A L'UNIVERSITÉ PARIS-SUD

par

Benjamin WERNER

Discipline : Informatique

Titre :

Faire simple pour pouvoir faire compliqué

Contributions à une Théorie des Types pratique

Présenté le 18 avril 2008 devant la Commission d'examen composée de

M.	**Claude Kirchner**	**Président**
MM.	**Andrea Asperti**	**Rapporteurs**
	Pierre-Louis Curien	
	Carlos Simpson	
Mme.	**Christine Paulin-Mohring**	**Examinateurs**
M.	**Gérard Huet**	

Première partie

Introduction

Chapitre 1

De la question des fondements à l'architecture logicielle

Positionnement

La logique mathématique se confond aujourd'hui, pour une large part, avec des pans de l'informatique. Cette intersection recouvre évidemment des travaux très divers ; en simplifiant, on peut reconnaître deux courants :

- L'un cherche à découvrir des structures nouvelles dans les démonstrations vues elles-mêmes comme des objets mathématiques.
- L'autre cherche à construire des outils informatiques pour manipuler, construire et vérifier des faits mathématiques.

Bien sûr, les deux approches se rejoignent pour appliquer la rigueur et la méthode mathématique à l'étude des démonstrations elles-mêmes. Dans le premier cas on traite la logique comme une discipline mathématique à part entière : on peut faire des (vraies) mathématiques même lorsque l'on fait de la logique. Dans le second cas, on cherche à montrer que des mathématiques complexes peuvent être formalisées : on peut faire de la logique même en faisant des (vraies) mathématiques.

Évidemment, présenter les choses ainsi est simplificateur et on peut trouver des travaux procédant de ces deux volontés à la fois. Je peux toutefois dire que ce qui est présenté dans ce mémoire relève du second courant : le but ici est d'aider à rendre effective la vérification formelle du raisonnement mathématique.

1.1 Les règles du jeu mathématiques

Les mathématiques sont la discipline de la vérité objective et la logique mathématique est sans doute née de la volonté d'expliciter cette constatation. Ce que Cantor, Frege, Peano, Zermelo, Russell et leur collègues ont cherché, c'est donc de proposer un ensemble de règles syntaxiques et non ambigües, définissant ce qu'est une démonstration correcte.

Ce n'est sans doute pas un hasard si cette idée est née, ou au moins a été mise en œuvre, avec la révolution industrielle, à l'âge où voyaient le jour des machines de plus en plus grandes et évoluées, qui mettaient en œuvre des *mécanismes* de plus en plus complexes. De fait, au cours du XXsiècle les notions d'algorithmes et de machines se sont progressivement imposées en logique, explicitant ainsi ce qui était d'abord implicite : une fois le formalisme fixé, la vérification de la correction d'une preuve formelle est *décidable*, c'est-à-dire qu'elle peut être effectuée mécaniquement, sans plus faire intervenir la perception ou l'intelligence humaine. Cette particularite reste une caractéristique propre des mathématiques. Peut-être en viendra-t-on même à considérer que c'est là la bonne définition des mathématiques : la discipline où les raisonnements peuvent être vérifiés par une mécanique simple.

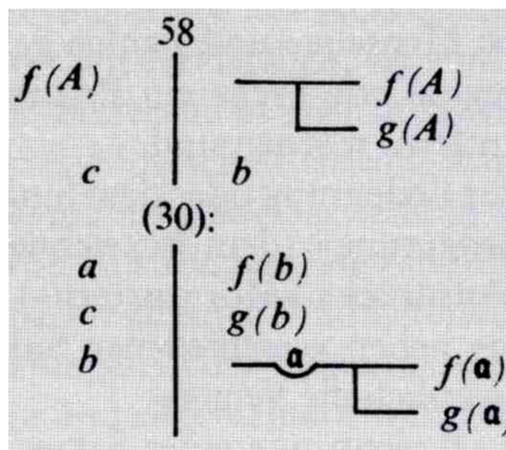


Fig. 1.1 – Un détail de la *Begrifsschrift* (écriture mathématique) de Frege (1872).
Si elle n'a pas eu de descendance directe, on considère qu'elle marque la naissance de la logique moderne. On ne peut qu'être frappé par la nature informatique de ces arbres syntaxiques.

En définissant formellement les critères de correction d'une preuve mathématique, on établit la vérité mathématique comme une notion précisément définie. Comme la vérité "la mieux définie". Aujourd'hui encore l'interjection "c'est mathématique" sert à signifier qu'il ne sert à rien de discuter, que l'on a à faire a un raisonnement sans faille.

La logique mathématique c'est donc d'abord l'établissement des règles du jeu mathématique. Définir précisément quels sont les coups permis pour prouver un théorème ou, plus prosaïquement pour l'étudiant, pour répondre à l'exercice.

Les choses ont certainement changé dans la vision philosophique que l'on a du rôle de ces règles. Pour les précurseurs qu'étaient Boole puis Frege, le but de la logique était d'établir les règles de la "pensée pûre" (respectivement *pure thought* et *reines Denken*) et pas uniquement les règles des mathématiques. De même, on trouve chez eux l'idée alors relativement répandue qu'il y a *une* vérité mathématique et que le rôle du logicien était d'en *découvrir* les *fondements*. Ce genre de conception paraît maintenant plus désuet, ne serait-ce que pour le théorème d'incomplétude de Gödel. Pour citer Jean van Heijenoort[1], on est ainsi passé de l'*absolutisme* à un certain *relativisme* dans la vision des règles logiques. En particulier, on accepte aujourd'hui plus volontiers comme une donnée la pratique mathématique existante.

D'un autre coté, l'idée qu'une démonstration correcte peut, *en principe* être ramenée à une telle démonstration formelle semble s'être imposée aujourd'hui. Dans la pratique on peut donc comprendre les textes mathématiques modernes comme la description informelle d'une preuve formelle. En revanche, la construction de véritables preuves formelles a longtemps été considérée comme un exercice vain à plusieurs égards : Dès que l'on considère des démonstrations non-triviales, la taille de leurs formalisations fait qu'elles deviennent inintelligibles et il apparaît qu'à partir d'un certain degré de détail, formaliser ne fait qu'augmenter la possibilité de cacher des erreurs, c'est bien sûr le contraire de ce qui est recherché.

1.2 De vraies machines

Comme dans bien d'autres domaines, c'est l'arrivée de l'ordinateur qui change la donne. D'une part, l'ordinateur est essentiellement la machine capable *effectivement* de construire, manipuler et surtout vérifier une preuve formelle. Il faut encore mettre en avant le rôle pionnier joué par Nicolas Gauvert de Bruijn qui, avec son équipe, développe le premier *système de preuves*, Automath [35], dans les années 1960.

[1]Tel que cité par Jean-Yves Girard dans *La mouche dans la bouteille*[51]

Fig. 1.2 – N. G. de Bruijn

Avec Automath arrivent les premières preuves formelles non triviales, en particulier la célèbre formalisation des *Grundlagen der Analysis* de Landau [75], par Bert Jutting, qui valident une première fois l'approche.

Mais aussi, Automath apporte un lot d'innovations conceptuelles importantes :

- Avec le formalisme d'Automath, De Bruijn crée, avant la lettre l'isomorphisme de Curry-Howard et les types dépendants. De fait, les preuves en Automath sont des λ-termes typés dans une variante du système plus tard appelé $\lambda\Pi$ ou LF. On voit donc que dès l'origine, les systèmes de preuve ont influencé la conception des formalismes qu'ils implémentaient.
- De Bruijn a donc bien compris l'intérêt de considérer les preuves comme des objets du formalisme. Surtout, il considère cette caractéristique du point de vue de l'architecture logicielle : la syntaxe des preuves étant précisément définie, seule la partie du logiciel qui vérifie la correction des preuves achevées est *critique* pour la confiance que l'on peut avoir dans les résultats formalisés en Automath. C'est l'idée du *noyau* du système de preuves sur laquelle nous revenons dans le paragraphe suivant.

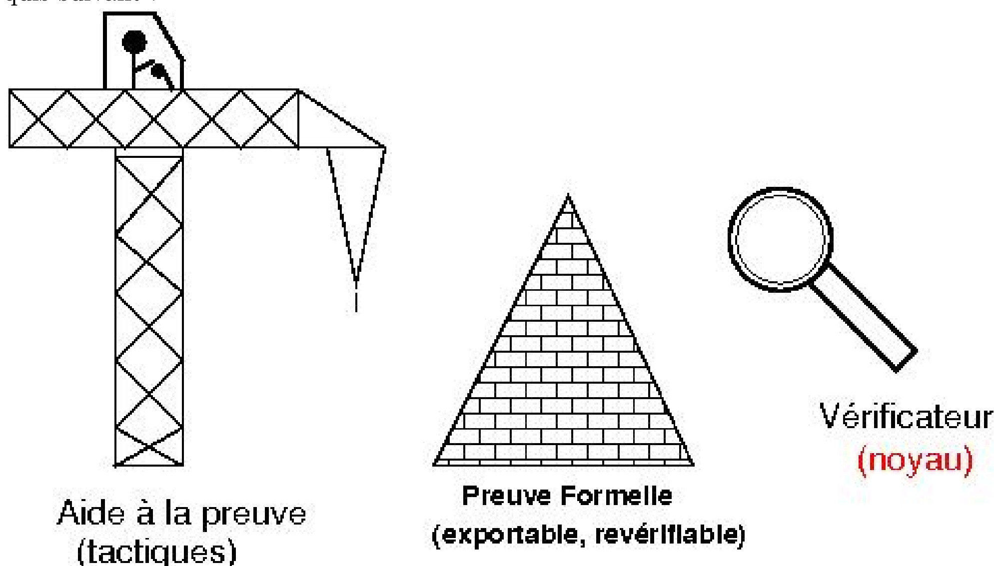
Les années 1960 voient également la naissance de la *démonstration automatique*, c'est-à-dire l'étude de comment des algorithmes exécutables peuvent automatiser en partie la tâche première du mathématicien : la recherche de preuves. Même si Automath évitait de faire appel à la démonstration automatique, on comprend bien comment cette technologie peut s'insérer dans un tel système de preuves : en fournissant à l'utilisateur un certain nombre d'outils l'aidant à construire la preuve formelle.

1.3 Les deux rôles de l'ordinateur

De fait, aujourd'hui encore, tous les systèmes implémentant des théories des types, et donc Coq en particulier, reprennent l'architecture principale d'Automath. D'un coté toute la partie *d'aide à la preuve*, à laquelle on demandera d'être aussi performante que possible, de l'autre le *vérificateur* ou *noyau* auquel on demandera d'être aussi sûr et fiable que possible.

Le formalisme logique est alors ce qui définit la forme et les propriétés de l'objet-preuve, c'est-à-dire

exactement l'interface entre ces deux parties. Depuis 1968, on peut voir les systèmes de preuve à travers le croquis suivant :



Il est essentiel de bien comprendre que l'ordinateur joue un rôle différent dans ces deux parties du système :

– Dans la partie d'aide à la preuve (à gauche), l'ordinateur est là pour aider l'utilisateur. *Il joue donc le rôle du "gentil policier"* des romans noirs. On peut exploiter pleinement sa puissance de calcul ; d'une certaine façon, tous les coups sont permis pour arriver au résultat, à savoir la preuve formelle.
– Dans la partie vérificateur (à droite), l'ordinateur devient le *méchant policier* ; celui qui interroge le suspect et ne le laissera passer que s'il est tout-à-fait assuré de son innocence (ici la correction de sa preuve). Dans ce cas, l'ordinateur n'aide pas l'utilisateur, au contraire il le contraint à rester dans le cadre ultra-réglementé de la logique formelle.

On peut expliquer partiellement cette dichotomie en remarquant que l'on exploite des caractéristiques différentes de la machine dans chaque cas : pour la construction on exploite la capacité de la machine à *calculer vite*. Dans le second, on compte sur la capacité de la machine à exécuter des algorithmes de manière *bien définie*, *prédictible* et *reproductible*.

Cette dialectique entre les deux visages de l'outil informatique est aussi le fil rouge des travaux présentés ici.

Avant de poursuivre, remarquons encore que :

– Cette séparation en deux rôles n'est pertinente que si elle est effective dans *l'architecture logicielle* du système.
– Là encore de Bruijn a fait preuve d'une clairvoyance tout-à-fait remarquable en établissant dès alors le concept de partie critique du logiciel. Cette notion est aujourd'hui omniprésente dans toute l'informatique ayant trait à la sécurité, sous la dénomination *trusted computing base*.

1.4 Vérités nouvelles

L'aide que l'ordinateur peut apporter au mathématicien ne se limite évidemment pas aux preuves formelles. De par sa puissance de calcul et sa capacité de stockage, il permet d'accéder à de nouvelles vérités, inconnues jusqu'alors.

Dans certains cas, l'ordinateur aide simplement la mathématicien à formuler le résultat qui est ensuite prouvé par des méthodes traditionnelles. Dans son remarquable petit article *Mathematics : an Experimental Science* [108], Herbert Wilf montre comment cela peut être le cas à travers un certain nombre d'exemples concrets. Le premier, pour en mentionner un, utilise la *base de données des séquences d'entiers* qui, à partir de $1, 2, 25, 543, 29281\ldots$ va reconnaître la suite des nombres de graphes orientés acycliques à n sommets.

Un autre exemple simple et frappant est celui des grands nombres premiers. Le dernier record du plus grand nombre premier établit "à la main" date de 1951 ; c'était le nombre $(2^{148} + 1)/17$ qui comporte 44 chiffres en notation décimale. En 2005 on a établit qu'un nombre de 7.816.230 chiffres, $2^{25964951} - 1$, était premier. La raison première de ce progrès vertigineux est évidemment la puissance de calcul des machines modernes. Il faut toutefois noter que l'arrivée de ces machines a été à l'origine de recherches nouvelles pour établir comment utiliser au mieux cette puissance de calcul, typiquement dans la cas de la primalité. Un ordinateur moderne aurait, évidemment, été utile aux mathématiciens de 1951, mais ces derniers n'auraient pas été capables de l'utiliser pour aboutir aux mêmes résultats que ceux d'aujourd'hui[2]. Il lui manquerait pour cela toute une littérature moderne, par exemple les résultats techniques liant primalité et courbes elliptiques. Cet exemple montre que l'arrivée de l'outil informatique est elle-même à l'origine de nouvelles mathématiques intéressantes.

On voit ici une différence fondamentale entre cet exemple et ceux de Wilf. Pour l'essentiel, ces derniers illustrent comment l'ordinateur aide à *formuler* un résultat de manière donc indépendante de la question de la *vérification* qui est celle qui nous intéresse au premier chef. Mais on compte également un nombre croissant de résultats mathématiques dont on ne sait assurer la validité qu'à travers l'exécution d'un programme informatique.

En d'autres termes, l'exemple des grands nombres premiers montre que certaines des découvertes dues à l'ordinateur (lorsqu'il joue le rôle de "gentil policier") posent des nouvelles questions quant à leur vérification formelles (lorsque l'ordinateur devient le "méchant policier").

1.5 De nouvelles preuves

Même si une phrase comme "1789 est premier" est presque le prototype de la proposition mathématique, l'utilisation de la machine pour établir de telle vérités mathématiques n'a pas suffit à changer la manière de penser des mathématiciens qui n'étaient pas concernés au premier chef par cette irruption de la technologie dans ce qui relevait jusqu'alors le leur seule intime conviction : distinguer la vérité. Cela est sans doute du à la nature particulièrement calculatoire de la notion de primalité. Aussi, l'impact et l'émoi épistémologique a-t-il été bien plus grand lorsque fut pour la première fois prouvé, "avec l'ordinateur", un résultat célèbre dont l'énoncé ne semblait le désigner comme devant reposer sur des calculs particulièrement complexes. Il s'agit évidemment du théorème des quatre couleurs. Je consacre plus loin un chapitre à ce théorème et ne rentre pas ici dans les détails techniques. Mais cet exemple, et plus près de nous celui de la conjecture de Kepler, montre que même des branches a priori très abstraites et peu calculatoires des mathématiques peuvent avoir besoin de recourir aux calculs mécanique.

Se pose alors la question de la validation de ces nouveaux arguments mathématiques. Elle se pose même très concrètement, comme le montre la discussion longue et difficile quant au statut et et la publication de la preuve de la conjecture de Kepler par Thomas Hales. Ces discussions se justifient si l'on considère que :

1. lorsqu'une preuve repose, par exemple, sur une énumération de plusieurs centaines de millions de cas, comme pour les quatre couleurs, on ne peut plus parler de *compréhension* comme on le fait pour une preuve traditionnelle. Il est difficile de répondre à la question de "pourquoi" quatre couleurs suffisent à colorier une carte. Au plus peut-on dire que l'on a vérifié que c'était vrai.

2. A partir de là, une telle preuve est beaucoup plus sensible aux "petites" erreurs qu'une preuve traditionnelle. Une erreur de typographie sera corrigée en passant par le lecteur d'une texte mathématique traditionnel. Si elle survient dans un programme informatique elle compromet évidemment tout l'édifice.

Mais ces preuves posent également des questions pour ce qui est des systèmes de preuves. On peut ainsi imaginer instrumenter les programmes en jeu dans la vérification des résultats de telle manière à ce qu'ils produisent, "en passant" une preuve formelle du résultat qu'ils sont en train de vérifier. Mais un rapide

[2]Dans un autre domaine, Gilles Kahn citait, dans une conférence à l'académie des sciences, le cas de l'algorithme de simplex ; dans la même période de temps, les progrès dans la compréhension de l'algorithmique du problème on permis de gagner un facteur de temps supérieur à ce que la loi de Moore a fait gagner en vitesse de calcul (plus de 800).

calcul d'ordre de grandeur montre rapidement que l'objet-preuve ainsi construit dépasserait également les capacités de l'ordinateur. Je dois à Jean-Louis Krivine la remarque que la preuve du théorème des quatre couleurs, si elle était déroulée de manière traditionnelle, serait sans doute plus longue que l'ensemble des textes mathématiques écrits jusqu'à présent. Il apparaît donc que l'on ne peut se passer, dans l'exposition d'une telle preuve, d'écrire le programme exécuté. La question qui se pose alors au formaliste ou au logicien est : dans quel *langage* une telle preuve est écrite, qui mêle arguments déductifs et calcul informatique ?

1.6 Les règles du jeu informatique

Il est remarquable que la famille de formalismes qui réponde à cette question n'a en fait pas du tout, au départ, été conçue pour cela. Per Martin-Löf a présenté, puis développé, dans les années 1970, la Théorie des Types moderne. Ses motivations étaient encore d'ordre largement philosophique ; en particulier il s'agissait pour lui de proposer un cadre fondateur pour les mathématiques constructives. De notre point de vue actuel, nous pouvons dire sommairement qu'il s'agit d'une extension du formalisme d'Automath par un langage de programmation fonctionnel et (évidemment) typé. Disposer de ces programmes dispense alors de recourir à des axiomatisations : contrairement à Automath la Théorie des Types est conçue pour permettre des développements sans aucune hypothèse supplémentaire.

Rajouter des programmes au formalisme devient véritablement intéressant lorsque l'on identifie ces derniers modulo la calcul. Dans le cas des λ-calculs typés, cela revient à introduire la *règle de conversion* :

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash t : B} \ (\textsc{si}\ A =_\beta B)$$

En théorie des types, les fonctions sont construites comme des programmes ; qui dit programme dit calcul, ainsi l'objet mathématique $2+2$ est un *calcul en attente* dont le résultat est 4. On dit que $2+2$ *se réduit* ou *s'évalue* en 4, ce que l'on note $2+2 \triangleright 4$. La relation d'équivalence $=_\beta$ de la règle de conversion étant simplement la clôture congruente, transitive, réflexive et symétrique de $\triangleright$. On voit que les propositions $2+2=4$ et $4=4$ sont logiquement identifiées : elles ont les mêmes preuves. On prouve donc $2+2=4$ en une seule étape logique, et sans avoir besoin de faire référence aux propriétés logiques de l'addition : son seul comportement calculatoire suffit. Surtout, la preuve est plus courte que dans des formalismes non-calculatoires comme la logique du premier ordre. Cet avantage en taille augmente évidemment avec les nombres additionnés : il est plus important lorsque l'on prouve $200+200=400$ en une seule étape, et plus encore pour $20.000+20.000=40.000$.

On va voir dans ce mémoire comment on utilise cette caractéristique pour formaliser effectivement des preuves qui ne peuvent être établies que par le calcul : la théorie des types est un langage dans lequel on peut écrire, à taille humaine, des preuves de primalité de grands nombres ou la démonstration du théorème des quatre couleurs.

Il faut noter que si l'on commence à ouvrir la porte du formalisme logique aux programmes informatiques, il se pose alors la question de jusqu'à quel point on la laisse ouverte :

- Quel langage de programmation : simplement fonctionnel pur ou accepte-t-on certains traits impératifs et lesquels ?
- Veut-on donner un statut particulier aux nombres, utiliser les possibilités matérielles du microprocesseur pour traiter rapidement les nombres entiers bornés ? ou les nombres flottants ?
- Quel modèle d'exécution choisit-on ? des programmes interprétés ou compilés ?

La première de ces trois questions relève très essentiellement du formalisme, c'est-à-dire, du point de vue de l'implémentation de la spécification du noyau. La seconde est déjà à cheval entre des questions logiques (le statut des nombres) et essentiellement informatiques (à quelle implémentation des opérations arithmétiques fait-on confiance). La dernière est essentiellement extérieure aux règles logiques mais le formalisme sera évidemment reflété dans le modèle d'exécution choisi. On voit enfin que dans toutes ces questions ont doit faire un compromis entre d'un coté une certaine simplicité, garante de sûreté, et de l'autre l'efficacité qui permet d'incorporer aux preuves des programmes plus complexes et donc de prouver plus de résultats.

1.7 Un choix d'Architecture

Dans tous ces choix, le formalisme doit garantir la cohérence des mathématiques formalisées. Mais il est aussi la pierre angulaire de l'architecture logicielle du système. De Bruijn a toujours prêché pour un formalisme et une implémentation du noyau aussi simples que possible : *"It should hold on one slide"*. Nous voyons que nous sommes obligés de faire un peu plus compliqué si nous voulons pouvoir traiter des preuves calculatoires : un mécanisme de compilation et d'exécution raisonnable ne peut s'écrire en quelques lignes. Mais on peut garder du principe de de Bruijn originel plusieurs idées fondamentales :

- La notion d'objet preuve dans la formalisme, qui permet d'identifier un *noyau critique* dans le logiciel dont le formalisme définit le comportement attendu de manière précise.
- A défaut de faire tout tenir sur une page, l'idée d'essayer de garder le formalisme et donc son langage de programmation aussi simples que possibles.

On se garde ainsi en particulier la possibilité d'appliquer les critères de vérification les plus rigoureux au noyau, en particulier de le valider en utilisant à nouveau des méthodes formelles. On ramène ainsi l'incertitude quant à la confiance à accorder au système le plus près possible du minimum irréductible que nous impose le théorème d'incomplétude. Mais il est important de souligner encore une fois qu'à partir du moment où l'on s'attaque à des résultats qui ne peuvent être établis qu'à travers des calculs importants, on n'a pas d'autre choix que de faire, à un moment dans le processus de vérification, confiance "aveuglément" à un mécanisme de calcul évolué donc relativement compliqué.

1.8 Des Mathématiques comme une science expérimentale

Georges Gonthier fait la remarque très juste que si l'on veut encore minimiser les possibilités d'erreur, on peut encore relancer la vérification du même objet-preuve en utilisant un autre processeur, un autre système d'exploitation, une autre implémentation du noyau, etc...Cela revient à faire varier les conditions expérimentales dans lesquelles on procède à la vérification de la preuve. On retrouve donc l'idée des mathématiques comme d'une science expérimentale, mais pas seulement, comme décrit par Wilf, parce que les résultats sont parfois obtenus par essai et erreur, mais aussi parce que la vérification même de la correction d'une preuve ne repose plus sur la *perception* du lecteur, mais sur un processus matériel reproductible.

Si l'on s'aventure quelques instants sur le terrain de l'épistémologie, on ne peut manquer de voir là une certaine ironie. Car les pionniers à l'origine de la logique mathématique se situaient, pour une large part, dans une perspective positiviste pour laquelle existe *une vérité* dont les règles logiques ne sont que le reflet, et où laquelle la *perception* de vérité joue un rôle fondamental. Or c'est finalement ces mêmes règles qui, après quelques ajustements et à travers quelques lignes de code informatique permettent de s'affranchir de cette perception dans le processus de validation d'une preuve mathématique.

De même, lorsque Boole puis Frege conçoivent les lois logiques comme celles de la *pensée pûre*, on peut les rattacher au courant *idéaliste* : c'est la pensée qui pré-existe. Or ce sont justement leur outils qui nous permettent aujourd'hui de comprendre les démonstrations mathématiques comme des jugements matériels portant sur le succès d'un processus effectif de vérification.

On se retrouve au final dans une situation habituelle pour une discipline scientifique telle que décrite par Popper : d'une part la cohérence d'un formalisme est un postulat qui ne peut jamais être considéré comme acquis (Gödel), mais il est effectivement *falsifiable* par l'expérience. Ce serait la cas si un mathématicien arrivait à construire un paradoxe dont la vérification serait faite sur une machine (processus reproductible s'il en est).

1.9 Organisation du mémoire

Ceux de mes travaux qui sont rassemblés dans le corps de ce mémoire sont ceux liés directement à la théorie des types et plus particulièrement à Coq et son formalisme. Dans le chapitre suivant je rappelle sommairement quelques caractéristiques essentielles de ce formalisme.

Une première moitié de ce mémoire reprend des travaux qui décrivent de nouvelles preuves. Il s'agit donc d'illustrer l'idée que l'alliance entre calcul et déduction est une force de la théorie des types. Les deux chapitres sont consacrés à deux exemples mentionnés ci-dessus. Le premier reprend, en français, et sous une forme un peu différente, un article écrit avec Benjamin Grégoire et Laurent Théry qui montre comment le calcul permet de prouver la primalité de grands nombres en Coq ; j'ai essayé d'adapter en partie le contenu à ce mémoire. Le chapitre suivant présente certains aspects de la preuve du théorème des quatre couleurs, en insistant particulièrement sur le rôle du calcul et la manière dont il est traité dans la preuve en Coq.

Dans la partie suivante, on trouve des travaux dont le formalisme est *l'objet de l'étude*. Je m'y attache particulièrement à relier la Théorie des Types aux mathématiques plus usuelles à travers la théorie des Ensembles. Ces travaux se situent donc plus directement dans le prolongement de ma thèse de doctorat. Par rapport à ma thèse, une idée sous-tend ces articles, est de simplifier la sémantique de la théorie des types en *limitant l'imprédicativité*. On obtient ainsi une sémantique ensembliste très simple de la théorie des types, qui permet aisément de valider des axiomes habituels en mathématiques, comme l'extentionalité pour les fonctions, le schéma de remplacement ou l'axiome du choix. Le premier présente cette sémantique et montre essentiellement l'équi-consistance de la théorie correspondante et de la théorie des ensembles. Le second, bien plus court, décrit quelques difficultés rencontrées avec une version du formalisme implémentée par des anciennes versions de Coq.

1.10 Le reste du dossier HDR

Ce mémoire reprend le contenu de divers articles, comme indiqué en tête des chapitres correspondants. Parmi ceux-ci, deux font explicitement partie du dossier HDR : *The not-so-simple proof-irrelevant model of CC* avec Alexandre Miquel, et *On the power of proof-irrelevant Type Theories*.

Les trois autres articles sont indépendants de ce mémoire mais portent sur des sujets liés :

- *Proof Normalization Modulo*, avec Gilles Dowek, qui porte sur la théorie de la démonstration en Déduction Modulo, famille de formalismes logiques voisins de la Théorie des Types.
- *Simple Types in type theory : deep and shallow embedings*, avec François Garillot qui décrit une preuve formelle où le calcul joue également un rôle et qui peut ouvrir la voie à un traitement nouveau des lieurs dans les preuves formelles.
- *Choice in Dynamic Linking* avec Martin Abadì et Georges Gonthier, sur un langage fonctionnel avec une nouvelle forme de liaison dynamique, inspirée d'une interprétation à la Curry-Howard de l'opérateur ε de Hilbert. Ce travail est d'abord motivé par des applications aux langages de programmation, mais pourrait également être utilisé pour améliorer le traitement de la modularité en Théorie des Types.

Je reviens sur ces différentes applications possibles de ces travaux dans le chapitre de conclusion.

Petit historique

Parmi mes contributions, celle dont je suis le plus satisfait n'apparaît pas explicitement dans les publications. C'est d'avoir été parmi les premiers à reconnaître que l'on pouvait exploiter plus largement les possibilités de la règle de conversion en écrivant des programmes en Coq, puis en prouvant leur correction et ensuite les exécuter à l'intérieur du système pour établir de nouveaux théorèmes. J'avais ainsi contribué à souffler cette idée à Samuel Boutin qui l'a, une première fois, développée dans sa thèse et baptisée cette méthode "reflexion calculatoire". A peu près en même temps Henk Barendregt a baptisé cette approche *two-level approach*. Dans le premier cas il s'agissait d'abord d'optimiser et d'automatiser des simplifications algébriques (la tactique `ring`), l'équipe de Barendregt, avec Martin Oostdijk, a été la première à l'utiliser pour des résultats numériques. Enfin Georges Gonthier, qui avait étudié la preuve du théorème des quatre couleurs pour proposer un sujet de programmation aux élèves de l'École Polytechnique, a eu l'idée de tenter de l'utiliser pour formaliser la preuve des quatre couleurs en Coq. J'ai ainsi eu la chance de participer à cette aventure scientifique.

Chapitre 2

Rappels sur la Théorie des Types

Coq implémente le Calcul des Constructions (Co-)Inductives (CCI), une théorie des types dans la lignée des formalismes introduits par Per Martin-Löf. Je ne vais pas chercher ici à faire une introduction pédagogique. D'une part cela sortirait du cadre de ce mémoire, et d'autre part il existe maintenant une littérature de qualité, à commencer par le livre de Pierre Casteran et Yves Bertot [14]. Je ne vais pas non plus détailler complètement toutes les règles formelles de CCI ; cette seule tâche remplit déjà une bonne part du mémoire d'Habilitation de Christine Paulin-Mohring [89]. Je vais donc simplement essayer de reprendre quelques points essentiels en insistant sur ceux qui seront importants par la suite.

2.1 Les types dépendants : une implémentation des règles logiques

Même si l'aspect constructif de la théorie de types n'est pas central dans ce qui va suivre, un premier point important est qu'une théorie des types est toujours une *matérialisation de la sémantique de Heyting* [65]. Les preuves sont des programmes, les propositions des types.

- Une preuve canonique de $A \wedge B$ est une paire formée d'une preuve de A et d'une preuve de B. La conjonction est donc définie par le produit cartésien sur les types.
- Une preuve de $A \Rightarrow B$ est un programme transformant une preuve de A en preuve de B. L'implication est construite avec le type $\rightarrow$ des fonctions.
- Une preuve de $A \vee B$ est soit une preuve de A soit une preuve de B, à chaque fois accompagnée d'un "drapeau" indiquant dans quel cas on se trouve. C'est l'union disjoint sur les types.
- La proposition fausse correspond à un type vide.

Puisque l'on veut traiter également le calcul des prédicats, des objets peuvent apparaître dans les types. L'expression *types dépendants* signifie simplement que les types peuvent dépendre des valeurs des objets. La construction primitive première est le produit des fonctions dépendantes : $\Pi x : A.B$ est un généralisation de $A \rightarrow B$ où l'argument x peut apparaître dans le type B du résultat.

Cette construction permet de coder la quantification universelle toujours exactement comme prévu par Heyting.

On voit un peut plus bas, au paragraphe 2.2.3 comment est représentée la quantification existentielle.

2.1.1 Sortes

Un rapide rappel sur la notion de *sorte*. Lorsque l'on utilise des types dépendants, les notions syntaxiques de termes et de types sont mêlées. Il devient alors naturel d'introduire une classe de constantes particulières qui seront les *types des types*. Par exemple le prédicat "être pair" portant sur les entiers naturels pourra avoir pour type even : $\mathsf{nat} \rightarrow \mathsf{Prop}$, dans un système où Prop est une sorte servant à représenter des propositions.

Dans la présentations de théories des types "à la Martin-Löf" on utilise pas vraiment de sortes et on recourt à la place à l'utilisation *d'univers* qui permettent de désigner un fragment des types de la théorie. La notion de sortes est en fait née du Calcul des Constructions [103], est devenue plus explicite lorsqu'on

a étendu celui-ci avec une hiérarchie cumulative d'univers[1]. La notion de sorte a acquis un statut reconnu avec l'émergence des *systèmes de types purs* (PTS, par Berardi, Barendsen, Barendregt).

Nous ne redonnerons pas les règles de ces systèmes de types dans le détail. Rappelons simplement que dans ces systèmes :

- Un type bien formé dans un contexte est un objet dont le type est une sorte : $\Gamma \vdash T : s$.
- Chaque système est paramétré par des relations d'axiomes qui donnent le(s) type(s) d'une sorte $s_1 : s_2$,
- et surtout par des *règles* qui indiquent quelles sont les types fonctionnels qui peuvent être construits ; un règle est de la forme :

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma(x : A) \vdash B : s_2}{\Gamma \vdash \Pi x : A.B : s_3}$$

  on dit alors qu'on a la règle (s_1, s_2, s_3).

2.1.2 Imprédicativité

Une sorte est *imprédicative* lorsqu'on peut former un objet de cette sorte en quantifiant sur tous les objets de cette sorte. Donc s est imprédicative si :

- on a un axiome $s : s'$,
- une règle de la forme : (s', s, s).

Pour peu que l'on puisse former des types fonctionnels dans s (c'est-à-dire que l'on ait la règle (s, s, s) ce qui est toujours la cas pour les théories dont on parlera), on peut donc plonger le système F de Girard [50] dans le système de types en question.

2.1.3 Les sortes de Coq

Les théories des types implémentées par Coq peuvent être décrites ainsi :

- Une sorte imprédicative Prop de type Type_1
- Une hiérarchie de sortes prédicatives : $\mathsf{Type}_1 : \mathsf{Type}_2 : \ldots \mathsf{Type}_i : \mathsf{Type}_{i+1} : \ldots$
- Une sorte Set également de type Type_1.

De plus, des définitions inductives[2] sont autorisées dans toutes ces sortes.

2.2 Description sommaire du système

Le système de types de Coq est appelé *Calcul des Constructions (co-)Inductives* ou CCI. On commence par décrire le système de types pur sous-jacent.

2.2.1 le PTS dans CCI

On a déjà indiqué quelles sont les sortes du système. Ce qui suit est donc totalement standard, avec une petite exception : on marque les variables pour indiquer lesquelles appartiennent à la sorte Prop et lesquelles appartiennent aux sortes Type_i. Cela va permettre d'une part d'éviter l'écueil découvert par Alexandre Miquel et mentionné ci-dessus dans la preuve de correction de la sémantique ensembliste. Cela permettra également de définir une extension de ce système en intégrant la *proof-irelevance* à la règle de conversion ; je reviendrai sur ce dernier point, étudié dans l'article [107] dans la section 5.4.

[1] Ces derniers sont effectivement des sortes dans le sens que nous essayons de décrire ici. Ils sont donc techniquement différents des univers "à la Martin-Löf", même s'il y a des liens, au moins intuitifs, évidents.

[2] Et aujourd'hui également co-inductives.

La syntaxe : sortes, marques, termes et contextes

$$s ::= \mathsf{Prop} \mid \mathsf{Type}(i) \qquad \mathsf{s} ::= * \mid \diamond$$
$$t ::= s \mid x_{\mathsf{s}} \mid \lambda x_{\mathsf{s}} : t.t \mid (t\ t) \mid \Pi x_{\mathsf{s}} : t.t$$
$$\Gamma ::= [] \mid \Gamma(x : t).$$

Nous écrirons parfois x pour $x_{/}s$ en omettant le marquage s lorsqu'il n'est pas relevant ou peut-être déduit du contexte.

La liaison des variables suit les règles usuelles. Nous écrivons $t[x \setminus u]$ pour la substitution des occurrences libres de la variable x dans t par u. Nous suivons la coutume de ne pas traiter l'α-conversion, en laissant ouvert le choix entre variables nommées et indices de de Bruijn.

Nous suivons également la pratique habituelle d'écrire $A \to B$ pour $\Pi x : A.B$ lorsque x n'a pas d'occurrences libres dans B.Nous écrivons aussi $\Pi x, y : A.B$ (resp. $\lambda x, y : A.t$) pour $\Pi x : A.\Pi y : A.B$ (resp. $\lambda x : A.\lambda y : A.t$).

Conversion

La β-réduction $\rhd_\beta$ est définie comme d'habitude. On note $\rhd_\beta^*$ la clôture reflexive-transitive et $=_\beta$ la clôture reflexive-symétrique-transitive. La propriété de Church-Rosser est évidemment vérifiée.

Pour traiter de l'inclusion entre les Type_i on définit un sous-typage syntaxique co-variant :

1. si $A =_\beta B$ alors $A \leq B$,
2. $\mathsf{Type}_i \leq \mathsf{Type}_{i+1}$,
3. Si $A \leq B$ alors $\Pi x : C.A \leq \Pi x : C.B$,
4. enfin $\leq$ est transitive : si $A \leq B$ et $B \leq C$ alors $A \leq C$.

2.2.2 Les règles

Les règles sont donc données ici dans un style PTS [9] ; la particularité concerne les régles où de nouvelles variables sont "poussées" dans le contexte.

Dans la règle Prod, max est le maximum de deux sortes pour l'ordre $\mathsf{Prop} < \mathsf{Type}(0) < \mathsf{Type}(1) < \dots$

2.2.3 Types inductifs

Les types inductifs sont une caractéristique essentielle de CCI. Ils servent, entre autres à définir des types de données, des connecteurs logiques. Ils sont présentés dans une forme proche de celle utilisée encore aujourd'hui Par Thierry Coquand et Christine Paulin [102]. La description précise est toujours assez longue et technique. J'ai choisi de n'utiliser ici que des exemples.

Notations

Un type inductif est le plus petit type clos par une liste de constructeurs. L'exemple le plus connu est peut-être celui des entiers de Peano :

$$\begin{array}{rl} \text{Inductive } \mathsf{nat} : \mathsf{Type}_1 := & \mathsf{O} : \mathsf{nat} \\ \mid & \mathsf{S} : \mathsf{nat} \to \mathsf{nat}. \end{array}$$

Cette *commande Coq* défini trois objets nat, O et S des types attendus. Pour chaque type inductif, on obtient également des opérateurs d'analyse de cas *(pattern-matching)* et de récurrence structurelle.

Par exemple l'addition :

$$(\textsc{Prop})\frac{\Gamma \vdash \mathrm{wf}}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}(i)} \qquad (\textsc{Type})\frac{\Gamma \vdash \mathrm{wf}}{\Gamma \vdash \mathsf{Type}(i) : \mathsf{Type}(i+p)}$$

$$(\textsc{Base})\frac{}{[] \vdash \mathrm{wf}} \qquad (\textsc{Var})\frac{\Gamma \vdash \mathrm{wf}}{\Gamma \vdash x : A}\text{if } (x : A) \in \Gamma$$

$$(\textsc{Cont})\frac{\Gamma \vdash A : \mathsf{Type}(i)}{\Gamma(x_\diamond : A) \vdash \mathrm{wf}} \quad (\textsc{Cont*})\frac{\Gamma \vdash A : \mathsf{Prop}}{\Gamma(x_* : A) \vdash \mathrm{wf}}$$

$$(\textsc{Conv})\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s}{\Gamma \vdash t : B}\text{if } A \leq B$$

$$(\textsc{Prod})\frac{\Gamma \vdash A : s \qquad \Gamma(x_{\mathsf{s}} : A) \vdash B : \mathsf{Type}(i)}{\Gamma \vdash \Pi x_{\mathsf{s}} : A.B : \max(s, \mathsf{Type}(i))}$$

$$(\textsc{Prod*})\frac{\Gamma \vdash A : s \qquad \Gamma(x_{\mathsf{s}} : A) \vdash B : \mathsf{Prop}}{\Gamma \vdash \Pi x_{\mathsf{s}} : A.B : \mathsf{Prop}}$$

$$(\textsc{Lam})\frac{\Gamma(x : A) \vdash t : B}{\Gamma \vdash \lambda x : A.t : \Pi x : A.B} \qquad (\textsc{App})\frac{\Gamma \vdash t : \Pi x : A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash (t\ u) : B[x \setminus u]}$$

FIG. 2.1 – Le fragment PTS

```
Fixpoint plus (n m : nat) {struct n} : nat :=
  match n with
  | O => m
  | S p => S (p + m)
  end
```

Bien sûr, une telle construction est munie de ses règles de réduction propres.

Comme le typage de ces opérateurs inclue les types dépendants, il permet de construire des opérateurs correspondant aux principes de récurrence :

$$\Pi P : \mathsf{nat} \to s.(P\ \mathsf{O}) \to (\Pi n : \mathsf{nat}.(P\ n) \to (P\ (\mathsf{S}\ n))) \to \Pi n : \mathsf{nat}.(P\ n)$$

où s est une sorte Prop ou Type_i.

Ces schémas possèdent alors leur propres règles de réduction, à l'image du récurseur du système T. De toute façon, nous utiliserons surtout la notation Coq qui est inspirée par celle de ML.

Restrictions

Deux restrictions principales assurent qu'une définition inductive fasse sens, et ne mette pas en danger la consistance et la normalisation du système. La première est bien connue : c'est la condition de stricte positivité sur les occurrences récursives dans le type des constructeurs ; on peut voir [102] pour les détails. Typiquement, la définition suivante est prohibée à cause de la première occurrence de foo dans le type de C :

Inductive $\mathsf{foo}\ :\ \mathsf{Type}_i\ :=$

$C : (\mathsf{foo} \to \mathsf{foo}) \to \mathsf{foo}.$

Sémantiquement, par exemple dans la sémantique ensembliste décrite plus loin, cette condition assure que la définition inductive corresponde au plus petit point-fixe d'un opérateur *monotone*.

La seconde restriction est plus subtile et essentielle à ce qui suit. Si le type inductif est de type Type_i (en fait, plus généralement si le type de son type est Type_{i+1}), alors tous les arguments de ses constructeurs doivent habiter le même univers ou plus bas. C'est-à-dire, en utilisant la cumulativité, les types des argument des constructeurs doivent aussi être de type Type_i. Relâcher cette condition permettrait par exemple de simuler le système paradoxal $\mathsf{Type} : \mathsf{Type}$. On ne sera pas surpris non plus de devoir utiliser cette restriction dans la construction du modèle plus bas. On voit une première conséquence de cette restriction dans le prochain paragraphe.

Deux versions de l'existentielle

La restriction ci-dessus correspond en fait à la prédicativité pour les types inductifs : elle signifie, on va le voir tout de suite, à restreindre la quantification existentielle dans Type_i aux univers inférieurs ou égal à i. En revanche, dans Prop on peut quantifier sur tous les univers également pour le quantificateur existentiel. En fait, on n'a pas fondamentalement besoin de définitions inductives dans Prop ; certaines propositions sont bien sûr définies inductivement, mais elles peuvent être construites en utilisant un codage imprédicatif. L'exemple clé est le quantificateur existentiel. Étant donnés $A : \mathsf{Type}_i$ et $P : A \to s$ on définit la proposition $\exists a : A.(P\ a)$ par :

$$\text{Inductive } \exists a : A.(P\ a) : \mathsf{Prop} \ := \quad \text{exi} : \Pi a : A.(P\ a) \to \exists a : A.(P\ a).$$

Mais cette définition est exactement équivalente à

$$\text{Definition } \exists a : A.(P\ a) := \Pi X : \mathsf{Prop}.(\Pi a : A.(P\ a) \to X) \to X.$$

Le point clé est qu'ici la restriction est sur les règles d'élimination. On ne peut pas éliminer des propositions inductives vers Type_i*.*

Il y a toutefois une définition alternative pour le quantificateur existentiel, à savoir le Σ-type habitant l'univers Type_i :

$$\text{Inductive } \Sigma a : A.(P\ a) : \mathsf{Type}_i \ := \quad \sigma : \Pi a : A.(P\ a) \to \exists a : A.(P\ a).$$

Chacune de ces deux définitions a ses avantages et ses limitations. La première habite la base de la hiérarchie d'univers et on peut donc toujours quantifier par rapport à elle. De l'autre coté, elle ne permet pas d'extraire le *témoin* d'une preuve existentielle : avoir prouvé $\exists a : A.(P\ a)$ ne signifie pas que nous pouvons exhiber le terme correspondant de type A. Plus exactement, on ne peut pas, en général, prouver $\exists a : A.(P\ a) \to \Sigma a : A.(P\ a)$.

Dans le cas du Σ-type, nous pouvons définir π tel que $(\pi\ (\sigma\ a\ p)) \triangleright a$. Mais nous ne pouvons pas parler de ce Σ-type dans des types habitants de Type_j inférieurs.

Remarquons encore que si l'on utilise pas le niveau imprédicatif, le fragment obtenu est, dans l'esprit, très similaire à la théorie des types prédicative de Martin-Löf [81]. La manière dont on conçoit le niveau imprédicatif ressemble beaucoup à la logique d'ordre supérieur de Church : les objets habitent le niveaux prédicatifs et Prop permet d'exprimer des faits à propos de ces objets ; en revanche on ne peut pas vraiment construire des objets à partir de preuves. Cette idée apparaît clairement dans la sémantique ensembliste et *proof-irrelevante.*

Inductifs dans Prop

On peut évidemment autoriser les inductifs dans Prop. C'est d'ailleurs ce qui est fait dans l'implémentation de Coq. Néanmoins, dès le "baptême" de cette sorte, l'idée est qu'elle puisse être utilisée pour *l'extraction de programmes* (voir 5.4 par exemple). C'est-à-dire qu'il doit être possible d'effacer les parties d'un terme correspondant au niveau Prop, tout en gardant un programme exécutable[3].

[3] Nous allons voir par la suite que cette propriété peut également être utile pour d'autres raisons que la génération de programmes certifiés. Par exemple pour introduire la non-relevance des preuves dans la théorie (section 5.4 ou simplement pour pouvoir admettre des axiomes plus fins, ne portant que sur les preuves non-calculatoires.

La conséquence est qu'un programme (c'est-à-dire ici un terme dont le type du type n'est pas Prop) ne doit jamais dépendre d'une preuve (un terme dont le type du type est Prop). Aussi, l'élimination des inductifs de type Prop ne doit être autorisée que vers la sorte Prop. C'est exactement ce qui se passe dans le paragraphe précédent : on ne peut utiliser une preuve de $\exists a : A.P$ pour retrouver le témoin $a : A : \mathsf{Type}$. C'est pourquoi, en première approximation, on peut considérer que l'on se passe d'inductifs dans Prop et que ces derniers sont codés en utilisant l'imprédicativité. Par exemple (avec A, $B : \mathsf{Type}_i$ et $, b : A$) :

$$\begin{array}{rcl} A \vee B & \equiv & \Pi P : \mathsf{Prop}.(A \rightarrow P) \rightarrow (B \rightarrow P) \rightarrow P \\ A \wedge B & \equiv B & \Pi P : \mathsf{Prop}.(A \rightarrow B \rightarrow P) \rightarrow P \\ a = b & \equiv & \Pi P : A \rightarrow \mathsf{Prop}.P\ a \rightarrow P\ b. \end{array}$$

Toutefois, si l'on regarde le détail des règles implémentées, on peut voir que dans certains cas particuliers, on veut autoriser une élimination de Prop vers Type_i. C'est le cas si la preuve éliminée ne recèle pas d'information calculatoire qui ne puisse pas être retrouvée. Plus précisément :

- Si la définition inductive a (au plus) un constructeur,
- si tous les arguments de ce constructeur sont eux-mêmes de sorte Prop.

Les deux exemples typiques sont la conjonction et surtout l'égalité. En Coq on a donc une élimination calculatoire de l'égalité de Leibniz :

$$\text{eq_ rec} : \Pi A\ a\ b.\Pi P : A \rightarrow \mathsf{Type}_i.P\ a \rightarrow a = b \rightarrow P\ b.$$

L'élimination forte

Une caractéristique importante des types inductifs est la possibilité de construire de nouveaux types par analyse de cas et par récurrence sur les objets inductifs. C'est cette caractéristique qui permet d'avoir des types "vraiment" dépendants. Un exemple simple est la construction qui permet de prouver que 0 et 1 sont différents. On construit un prédicat sur les entiers (c'est-à-dire une fonction de type $\mathsf{nat} \rightarrow \mathsf{Prop}$) par analyse de cas :

$$\begin{array}{l} \text{Definition discr} := \text{fun } x : nat \Rightarrow \text{match } x \text{ with} \\ \qquad |0 \Rightarrow \text{True} \\ \qquad |(S\ _) \Rightarrow \text{False} \\ \qquad end. \end{array}$$

En utilisant la règle de conversion, on voit immédiatement que (discr 0) est identifié à True et (discr $(S\ 0)$) à False. On en déduit que $0 = (S\ 0)$ implique False, c'est-à-dire que $0 \neq 1$.

2.3 Élimination forte et imprédicativité

Dans ma thèse, j'avais prouvé la normalisation forte d'un fragment important de la théorie des types de Coq. En fait, pour être précis, il faudrait dire de la théorie des types du Coq *tel qu'implémenté de l'époque*. En effet la caractéristique principale de la thèse était la conjonction de deux caractéristiques distinctes : Imprédicativité ou polymorphisme d'une part, élimination forte de l'autre.

La différence essentielle porte sur l'imprédicativité de la sorte Set. A l'époque elle était imprédicative ; aujourd'hui elle est prédicative par défaut et on peut retrouver l'ancien comportement en appelant Coq avec une option `-impredicative-set`. Dans le comportement par défaut de Coq, Set peut donc être simplement compris comme un nom particulier donné à une sorte Type_0 au pied de la hiérarchie prédicative des Type_i.

Dans la plus grande partie de ce document, on travaille avec la théorie courante, donc sans Set imprédicatif. Seul le chapitre 6 est consacré à l'exposition d'un certain nombre de difficultés liées à "l'ancienne" théorie.

Deuxième partie

Preuves calculatoires formelles

Chapitre 3

Traitement des certificats de Pocklington en Théorie des Types

1783 est premier. Une assertion comme celle-ci sera souvent considérée comme l'archétype de la proposition mathématique et à bien des égards, c'est justifié : cette affirmation se laisse traduire de manière très simple en de nombreux formalismes, dont, évidemment, l'arithmétique.

La manière dont on peut vérifier qu'un nombre est premier (nous dirons à partir de maintenant vérifier *la primalité* d'un nombre) a changé au cours du temps. Ces changements reflètent très largement les progrès du savoir mathématique et des outils de calcul dont l'homme dispose : calcul manuel "comme à l'école", crible d'Erathosthène, puis l'utilisation de théorèmes de théorie des nombres élémentaire, ensuite l'arrivée du calcul mécanique et électronique, enfin l'alliance d'ordinateurs modernes avec des résultats mathématiques dédiés qui ont été développés pour exploiter mieux la puissance de calcul des machines modernes.

Ce chapitre est essentiellement une reprise de l'article écrit avec Benjamin Grégoire et Laurent Théry en 2006 [13]. J'ai gardé volontairement certains aspects techniques pour illustrer les questions qui se posent lors de la construction d'une preuve calculatoire. J'ai également essayé de détailler un peu plus des preuves de primalité plus élémentaires, afin d'illustrer comment la frontière entre raisonnement et calcul bouge et suit l'évolution de la puissance des moyens de calculs à disposition du mathématicien. J'indique enfin les travaux récents d'autres informaticiens qui ont depuis prolongé ce travail.

3.1 Preuves de primalité élémentaires

3.1.1 La méthode de l'instituteur

La justification de la correction de cette proposition variant largement suivent le contexte, imaginons que nous devions la justifier simplement muni d'un tableau et d'une craie. On vérifiera par exemple :

1. Après quelques essais, on peut remarquer que $43 \times 43 > 1783$. Il suffit donc de vérifier que 1783 n'a pas de diviseur compris entre 2 et 43.
2. 1783 n'est pas divisible par 2, 3, 5. On vérifie ensuite qu'il en est de même pour 7, 11, 13, 17, 19.
3. Si l'on n'est plus sûr des nombres premiers suivants, on peut considérer les nombres impairs suivant : 21 n'est pas premier, 23 ne divise pas 1783, 25 et 27 ne sont pas premiers (connu), 29 et 31 ne divisent pas 1783, 33 et 35 ne sont pas premiers, 37 ne divise pas 1783, 39 n'est pas premier (divisible par 3 puisque la somme de ses chiffres est divisible par 3), 41 ne divise pas 1783.

On a ici effectué un certain nombre de divisions, utilisé notre connaissance des tables de multiplication, et les critères habituels pour voir qu'un nombre est divisible par 2, 3 ou 5.

Il est évidemment possible de reproduire ce cheminement dans un système de preuves. Ce serait toutefois relativement pénible, en particulier il faudrait formaliser, comme une proposition mathématique, le fait que

l'on a bien énuméré tous les nombres impairs compris entre 3 et 43. Or c'est un exemple typique de fait qui apparaît évident au lecteur mais qui n'est pas immédiat à formaliser en vue d'une vérification mécanique.

3.1.2 Le programmeur BASIC

La manière la moins fatigante de vérifier la primalité d'un nombre pas trop grand est d'écrire le programme naïf qui essaye de diviser n par tous les nombres compris entre 2 et $n-1$. Cela s'effectue tout aussi facilement en Coq. Voici le code de la fonction :

```
Fixpoint test (n:nat) (d:nat) {struct d} :=
match d with
| O => true
| (S O) => true
| (S ((S c) as e)) => (negb (dvdn d n))&&(test n e)
end.
```

Il est alors facile de prouver que `test` vérifie bien la primalité ; formellement :

$$\forall n > 1, \texttt{prime}(n) \iff \texttt{test}(n, n-1) = \texttt{true}.$$

La preuve de ce lemme n'est pas particulièrement intéressante et de fait, `test` pourrait presque servir de définition à `prime`.

Prenons maintenant un nombre premier quelconque et appelons n sa représentation en Coq. On sait alors que $\texttt{test}(n, n-1)$ se réduit vers `true` et donc l'axiome de réflexivité appliqué à `true` est aussi une preuve de $\texttt{test}(n, n-1) = \texttt{true}$.

En appliquant ce "résultat" au lemme précédent, on construit immédiatement une preuve de $\texttt{prime}(n)$. De plus la taille de cette preuve est quasiment constante ; elle contient simplement une occurrence de n.

Cette méthode est facilement implémentée en Coq. Une à deux pages de code permettent de prouver la primalité de nombres comme 1783 et un peu plus grands. De plus, on remarque que la taille de l'objet preuve ne dépend que très peu du nombre traité : tout juste doit on faire apparaître sa représentation dans un système qu'on choisira (binaire, décimal, ou autre).

Avec cette méthode, le facteur limitant n'est donc pas la taille de la preuve ou la mémoire nécessaire, mais bien le temps de calcul.

3.2 Calculer en autarcie ou accepter des certificats

Lorsque l'on parle de calcul et de preuves, on se trouve confronté à une dichotomie entre les calculs ayant lieu à l'intérieur et à l'extérieur du système :

- Les calculs internes font vraiment partie de la justification du résultat final. Ils sont donc considérés comme sûrs, car ils sont effectués par le mécanisme d'exécution du vérificateur de preuve. D'un autre coté, ce mécanisme d'exécution sera en général moins efficace que des programmes spécialisés pouvant exploiter librement les ressources de l'ordinateur.
- Les calculs externes où "tous les coups sont permis" pour obtenir le résultat aussi rapidement que nécessaire. En revanche, ces résultats n'auront pas force de preuve tant qu'ils ne seront pas revérifiés par le prouveur.

Une possibilité est donc de se restreindre et d'effectuer tous les calculs à l'intérieur du système. C'est l'approche *autarcique*, nommée ainsi par Barendregt [10]. C'est exactement le cas des preuves de primalité décrites dans la section précédente.

Une autre possibilité est de remarquer que, dans certains cas, une première exploration calculatoire du problème peut donner une information utile, même si elle doit être soumise à vérification. On peut donc utiliser les résultats obtenus à l'aide d'un logiciel spécialisé, à condition que ces résultats soient exportés vers le système de preuve sous forme d'une *trace*. Cette trace joue d'une certaine façon le rôle du fil d'Ariane qui permet de sortir plus rapidement du labyrinthe ; elle apporte des informations sur les calculs qui doivent être

effectués au moment de la vérification formelle et fait donc partie de l'objet-preuve. C'est ce que Barendregt appelle l'approche *sceptique*. Une première occurrence de cette approche peut être trouvée dans le travail de Harrison et Théry avec un interfaçage entre HOL et Maple [60] ; c'était toutefois pour un problème assez différent, où la vérification du résultat fourni par Maple était beaucoup plus simple.

L'utilisation de certificat de Pocklington est un exemple typique de l'approche sceptique. L'idée d'utiliser le théorème de Pocklington dans un cadre formel revient à Arjeh Cohen qui indiqua cette possibilité à Henk Barendregt ce qui donna ensuite lieu au travail de Caprotti et Oostdijk [22]. Dans notre travail, nous avons pu ré-utiliser en particulier la preuve du théorème de Pocklington proprement dit.

3.3 Les certificats de Pocklington

3.3.1 Le théorème

Le théorème de Pocklington [91] remonte à 1914 et fournit une condition suffisante à la primalité d'un nombre :

Théorème 1 *Soit un entier naturel $n > 1$, un témoin a et une séquence de paires d'entiers $(p_1, \alpha_1), \ldots, (p_k, \alpha_k)$. Pour que n soit premier, il suffit que les quatre conditions suivantes soient vérifiées :*

$$p_1 \ldots p_k \quad \textit{sont} \quad \textit{premiers} \tag{3.0}$$
$$(p_1^{\alpha_1} \ldots p_k^{\alpha_k}) \quad | \quad (n-1) \tag{3.1}$$
$$a^{n-1} \quad = \quad 1 (\mathsf{mod}\ n) \tag{3.2}$$
$$\forall i \in \{1, \ldots, k\}\ \mathsf{gcd}(a^{\frac{n-1}{p_i}} - 1, n) \quad = \quad 1 \tag{3.3}$$
$$p_1^{\alpha_1} \ldots p_k^{\alpha_k} \quad > \quad \sqrt{n}. \tag{3.4}$$

Il faut remarquer qu'il n'y a pas *un* théorème clairement défini dans l'ouvrage de Pocklington. Aussi, la littérature mentionne souvent des variantes un peu moins puissantes sous la même dénomination. Dans tous les cas, on peut faire trois observations simples mais essentielles :

- La première est, qu'étant donné n, il faut plus de calculs pour trouver des entiers $a, p_1, \alpha_1, \ldots, q_k, \alpha_k$ que pour vérifier que ces nombres vérifient effectivement les conditions 1 à 4 ci-dessus. C'est pourquoi l'on dit que $a, p_1, \alpha_1 \ldots, p_k, \alpha_k$ forment un *certificat de Pocklington* pour n. Caprotti et Oostdijk en ont justement conclu qu'il s'agit là d'un cas typique où l'approche sceptique convient : le certificat est construit à l'extérieur de Coq par un programme dédié et seule la vérification du certificat est effectuée à l'intérieur du système de preuve.
- La seconde observation est qu'étant donné n et un certificat $p_1, \ldots, p_k$ et a, vérifier la primalité de n se réduit à :
  1. la vérification des conditions 1-4 ce qui correspond uniquement à des calculs numériques,
  2. La vérification de la condition 0, ce qui peut être faite récursivement.
- La dernière observation est que le théorème 1 est le seul théorème qu'il est nécessaire de formaliser. Pour on utilise implicitement sa contraposée : si un nombre impair n est premier, il est toujours possible de trouver un certificat. En effet, étant donné une factorisation partielle suffisante de $n-1$, un générateur du groupe multiplicatif $\mathbb{Z}/n\mathbb{Z}$ est un bon candidat pour a. Or un tel générateur existe lorsque n est premier, car alors $\mathbb{Z}/n\mathbb{Z}$ est cyclique.

Le théorème 1 est celui utilisé par Caprotti et Oostdijk [22]. La condition 4 indique qu'il est nécessaire de factoriser partiellement $n-1$ jusqu'à sa racine carrée pour construire un certificat. Pour notre expérience, Laurent Théry a ensuite utilisé une variante légèrement plus puissante du théorème, qui fut proposée par Brillhart, Lehmer et Selfridge en [19]. Grâce à cette variante, on peut limiter la factorisation à la racine cubique de $n-1$; comme cette factorisation est la partie la plus difficile de la construction du certificat, c'est une amélioration notable. La théorème formalisé en Coq est le suivant :

Théorème 2 *Étant donné un nombre n, un témoin a et une séquence de paires s $(p_1, \alpha_1), \dots, (p_k, \alpha_k)$ où tous les p_i sont premiers, soient :*

$F_1 = p_1^{\alpha_1} \dots p_k^{\alpha_k}$
$R_1 = (n-1)/F_1$
$s = R_1/(2F_1)$
$r = R_1 \mathsf{mod}\ (2F_1)$.

Il est alors suffisant pour que n soit premier que les conditions suivantes soient vérifiées :

$$F_1 \text{ est pair, } R_1 \text{ est impair, et } F_1 R_1 = n - 1 \quad (3.5)$$
$$(F_1 + 1)(2F_1^2 + (r-1)F_1 + 1) > n \quad (3.6)$$
$$a^{n-1} = 1(\mathsf{mod}\ n) \quad (3.7)$$
$$\forall i \in \{1, \dots, k\}\ \mathsf{gcd}(a^{\frac{n-1}{p_i}} - 1, n) = 1 \quad (3.8)$$
$$r^2 - 8s \text{ n'est pas un carré ou bien } s = 0. \quad (3.9)$$

Les remarques faites à propos du théorème 1 restent valides pour le théorème 2. L'existence d'un certificat pour tout nombre premier impair n'est pas déduite directement du théorème 2 mais peut être vérifiée à partir du théorème précisément donné dans [19] qui est un peu plus fort :

Si les conditions 5 à 8 sont vérifiées, alors n est premier si et seulement si la condition 9 est vérifiée.

3.3.2 Les certificats

Un certificat ne se compose pas uniquement des nombres a,$p_1, \alpha_1, \dots, p_k, \alpha_k$. Pour pouvoir être autosuffisant, il faut adjoindre d'autres certificats pour les p_i, ainsi que pour les facteurs apparaissant dans ces nouveaux certificats.

On aboutit ainsi immédiatement à la définition récursive suivante. Un certificat pour un nombre entier n est donné par le n-uplet $c = \{n, a, [c_1^{\alpha_1}; \dots ; c_k^{\alpha_k}]\}$ où les $c_1, \dots, c_k$ sont respectivement des certificats pour les nombres $p_1, \dots, p_k$. Cela revient à voir les certificats comme des arbres dont les branches sont elles-mêmes des certificats correspondant aux facteurs premiers. La vérification se réduit uniquement à des calculs numériques.

Une telle structure est définie facilement en Coq comme un type inductif. Un certificat est soit :

- un n-uplet de la forme : $c = \{n, a, [c_1^{\alpha_1}; \dots ; c_k^{\alpha_k}]\}$[1]
- une paires (n, ψ) formée d'un nombre n est d'une preuve ψ attestant que ce nombre est premier.

Le second cas est ajouté pour autoriser des preuves de primalité qui ne reposent pas sur le théorème de Pocklington. C'est utile en particulier pour la primalité de 2 (qui ne peut être prouvée en utilisant Pocklington) mais aussi pour s'autoriser d'autres méthodes qui peuvent, au moins ponctuellement être plus pratiques pour certains nombres.

Avec cette représentation, un certificat[2] possible pour 127 est :

$$\{127, 3, [\{7, 2, [\{3, 2, [(2, \mathsf{prime2})]\}; (2, \mathsf{prime2})]\};$$
$$\{3, 2, [(2, \mathsf{prime2})]\};$$
$$(2, \mathsf{prime2})]\}$$

où prime2 est la preuve que 2 est premier.

On remarque alors que cette représentation duplique certains certificats (ici ceux de 2 et 3) qu'il faudra donc vérifier plusieurs fois. On autorise donc du partage en aplatissant la représentation, en remplaçant les arbres par des listes. Dans le cas présenté, on aboutit ainsi à :

$$[\{127, 3, [7; 3; 2]\}; \{7, 2, [3; 2]\}; \{3, 2, [2]\}; (2, \mathsf{prime2})].$$

[1]Dans la suite, on notera simplement c_i pour c_i^1.

[2]Il s'agit là d'un exemple illustratif ; le certificat qui sera effectivement généré pour 127 est plus concis : $\{127, 3, [\{3, 2, [(2, \mathsf{prime2})]\}; (2, \mathsf{prime2})]\}$.

Cela se traduit immédiatement par la définition Coq suivante :

```
Definition dec_prime := list (positive*positive).

Inductive pre_certif : Set :=
 | Pock_certif : forall n a : positive, dec_prime -> pre_certif
 | Proof_certif : forall n : positive, prime n -> pre_certif.

Definition certificate := list pre_certif.
```

Nous introduisons d'abord la notion de factorisation partielle (`dec_prime`), qui est une liste de facteurs premiers à chaque fois munis de leur exposant. Ensuite nous définissons la notion de pré-certificat qui est soit une paire formé d'un nombre et de sa preuve de primalité (cas `Proof_certif`), ou d'un triplet formé par un nombre premier n, un témoin a et d'une factorisation partielle de $n-1$ (cas `Pock_certif`). Il manque encore les certificats des éléments de la factorisation partielle.

Un certificat complet est une liste de pré-certificats. Le premier élément de la liste est en général un triplet (`Pock_certif` n a d), et le reste de la liste contient les pré-certificats pour les éléments de la factorisation d, et ainsi de suite.

D'une certaine manière, on peut voir ces certificats comme une mini-base de données, où l'on peut trouver toute l'information nécessaire pour démontrer que les différents éléments sont premiers.

3.4 La vérification des certificats

La définition précédente étant essentiellement une structure libre, l'existence d'un objet de type `certificate` ne suffit pas à garantir quelque propriété que ce soit. Nous détaillons maintenant la fonction C qui vérifie qu'un certificat est bien valide, ainsi que la preuve de correction de cette fonction.

Il s'agit donc de définir C comme une fonction des certificats vers les booléens, de manière à ce que lorsqu'elle retourne `true`, on peut garantir la primalité de tous les nombres contenus dans le certificat. Remarquons qu'ici, la *complétude* de la fonction C n'est pas nécessaire ; il nous faut simplement le lemme de correction suivant :

$$\mathsf{Pock_refl} : \forall c, l,\ C\ (c :: l) = \mathit{true} \Rightarrow \mathsf{prime}\ (n\ c)$$

où $(n\ c)$ est le nombre premier sur lequel porte le certificat c. Plus précisément, il nous faudra le lemme plus général :

$$\forall l,\ C\ l = \mathit{true} \Rightarrow \forall c \in l,\ \mathsf{prime}\ (n\ c)$$

La fonction C parcourt la liste l récursivement. Si la liste est vide, le résultat est `true`. Si la liste est de la forme $(l = c :: l')$, la fonction commence par vérifier récursivement la validité de l' puis, le cas échéant, la validité de c. Il y a là deux cas :

- Si c est obtenu à partir d'une preuve de la forme $c = (n, \psi)$, il n'y a rien à faire ; le typage de Coq suffisant à garantir que ψ est une preuve de primalité pour n.
- Si c est un pré-certificat de Pocklington $c = \{n, a, [p_1^{\alpha_1}; \ldots; p_k^{\alpha_k}]\}$, la fonction commence par vérifier que chacun des facteurs $p_1, \ldots, p_k$ possède bien un certificat dans l' (ce qui signifie qu'ils sont tous premiers). Si c'est bien le cas, la fonction vérifie les conditions 5 à 9. Cette dernière tâche est effectuée par une fonction dédiée C_c.

3.4.1 Vérification des conditions calculatoires

La fonction C_c commence par calculer les nombres $F_1,\ R_1,\ s,\ r$ telles que définies dans les théorème 2. La vérification des conditions 5 et 6 est simple. Pour ce qui est des conditions 7 et 8, la difficulté est de calculer rapidement a^{n-1} mod n and $\mathsf{gcd}(a^{\frac{n-1}{p_i}} - 1, n)$ pour $i = 1 \ldots k$. Il est essentiel de ne pas calculer explicitement a^{n-1} et $a^{\frac{n-1}{p_i}}$, qui peuvent être gigantesques. Pour cela, nous calculons toujours modulo n, ce

qui est possible puisque $\mathsf{gcd}(b, n) = \mathsf{gcd}(b \;\mathsf{mod}\; n, n)$. De plus, il est possible de ne calculer qu'un seul pgcd en remarquant que $\mathsf{gcd}(b_1 \dots b_l, n) = 1$ si et seulement si pour tout $i = 1 \dots l$, on a $\mathsf{gcd}(b_i, n) = 1$.

Nous définissons donc les fonctions suivantes, qui toutes travaillent modulo n :

- une fonction prédesséceur (`Npred_mod`) ;
- une fonction de multiplication (`times_mod`) ;
- une fonction d'exponantiation (`Npow_mod`) (en utilisant l'algorithme d'exponantiation rapide ou *repeated square-and-multiply algorithm*) ;
- une fonction de multiplication d'exposants (`fold_pow_mod`) qui à partir de a, $l = [q_1; \dots ; q_r]$ et n calcule $a^{q_1 \cdots q_r} \;\mathsf{mod}\; n$.

Une autre optimisation est de partager en partie les calculs des

$$a^{\frac{n-1}{p_1}} \;\mathsf{mod}\; n,\ \dots,\ a^{\frac{n-1}{p_k}} \;\mathsf{mod}\; n,\ a^{n-1} \;\mathsf{mod}\; n.$$

Soit $m = (n - 1)/(p_1 \dots p_k)$; si ces calculs sont effectués séparément, $a^m \;\mathsf{mod}\; n$ est calculé $k + 1$ fois, $(a^m \mathsf{mod}\; n)^{p_1} \;\mathsf{mod}\; n$ est calculé k fois, et ainsi de suite.

Pour partager ces calculs, on peut définir la fonction :

```
Fixpoint all_pow_mod (P A : N) (l:list positive) (n:positive)
                                                {struct l}: N*N :=
  match l with
  | nil => (P,A)
  | p :: l =>
    let m := Npred_mod (fold_pow_mod A l n) n in
    all_pow_mod (times_mod P m n) (Npow_mod A p n) l n
  end.
```

Alors, si P et A sont des nombres positifs inférieurs à n et l est la liste $[q_1; \dots ; q_r]$, la fonction `all_pow_mod` retourne la paire :

$$(P \prod_{1 \leq i \leq r} A^{\frac{q_1 \cdots q_r}{q_i}} \;\mathsf{mod}\; n, A^{q_1 \cdots q_r} \;\mathsf{mod}\; n).$$

On peut alors remarquer que l'application de cette fonction à $P = 1$, $A = a^m \;\mathsf{mod}\; n$ et $l = [p_1; \dots ; p_k]$ conduit au résultat recherché. On remarque aussi que l'ordre dans lequel les éléments apparaissent dans la liste l est important pour la vitesse de calcul. A^{p_1} est calculé une seule fois, mais les exponantiations des éléments de queue dans la liste l sont calculés plusieurs fois. Aussi, le calcul sera plus rapide si l est triée en ordre décroissant.

Finalement, la fonction C_c vérifie la condition 9 (à savoir que ($s = 0 \vee r^2 - 8s$ n'est pas un carré). Si $n \neq 0$ et $r^2 - 8s \geq 0$ il faut vérifier que $r^2 - 8s$ n'est pas un carré. Pour ce faire, nous adaptons légèrement la définition des certificats en ajoutant la racine carré de l'entier le plus petit[3] sqrt, et il suffit alors de vérifier que

$$\mathsf{sqrt}^2 < r^2 - 8s < (\mathsf{sqrt} + 1)^2$$

En conclusion, la définition finale des pré-certificats est donc :

```
Inductive pre_certif : Set :=
 | Pock_certif : forall n a sqrt: positive, dec_prime -> pre_certif
 | Proof_certif : forall n : positive, prime n -> pre_certif.
```

3.4.2 Définitions des opérations arithmétiques en Coq

La représentation des nombres est évidemment cruciale pour l'efficacité des calculs. Dans le langage purement fonctionnel de Coq, la manière standard de définir des structures de données et d'utiliser des types inductifs sur lesquels on peut calculer par récursion structurelle. Cette restriction, qui assure la normalisation

[3]Lorsque $r^2 - 8s < 0$ nous ajoutons simplement l'entier 1, puisque $r^2 - 8s$ n'est trivialement pas un carré.

forte du calcul est décrite par ailleurs. Elle impose parfois certaines contorsions pour arriver à écrire des versions efficaces des algorithmes recherchés.

Dans le cas des certificats de Pocklington, nous avons besoin des opérations de base sur les entiers. Le plus simple est d'utiliser la bibliothèque standard de Coq. On y trouve une définition des entiers sous forme de listes de bits :

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

`xH` représente 1, `xO` x représente $2x$, `xI` x représente $2x+1$. Ainsi 6 est représenté par `(xO (xI xH))`. Toutes les opérations sont purement fonctionnelles (on utilise donc pas les opérations arithmétiques du processeur et pas d'effets de bord). En conséquence, chaque fois qu'une fonction retourne un nombre, ce nombre doit être construit et la mémoire correspondante est allouée dans le tas. De fait, dans l'exemple décrit dans ce chapitre, l'allocation de mémoire et le glanage de cellules (*garbage collecting*) sont particulièrement coûteux.

Pour minimiser l'allocation de mémoire, nous avons du ré-implémenter certaines fonctions. Par exemple en écrivant une version du reste de division entière qui ne passe pas par le calcul du quotient. De même on a proposé une version optimisée du carré.

3.4.3 Amélioration récentes

Au cours des 18 mois qui ont suivi ce travail des progrès importants ont été fait pour ce qui est de la représentation des nombres en Coq :

- D'une part, Benjamin Grégoire et Laurent Théry [59] ont construit une représentation plus efficace des entiers, sous forme d'arbres binaires. Cette bibliothèque permet de construire des représentations d'entiers de tailles arbitraires à partir d'un type permettant de représenter des entiers bornés.
- Pour exploiter encore mieux cette bibliothèque, il apparaissait désirable d'utiliser les capacités arithmétiques du processeur pour effectuer le plus rapidement possible les opérations sur ces entiers bornés. Lors d'un stage sous ma direction, Arnaud Spiwack [97] a implémenté une version de Coq où le calcul sur une définition particulière des entiers 31 bits est effectivement délégué au routines de bas niveaux de C lors du type-checking.

La combinaison de ces deux travaux augmente très largement le taille des nombres effectivement manipulables par Coq et donc des nombres dont on peut prouver la primalité en Coq. Nous revenons sur ces points dans la conclusion de ce mémoire.

3.5 La construction de certificats

On décrit maintenant comment sont construit des certificats pour des grands nombres premiers. L'étape critique est en général la factorisation partielle de $n-1$.

Comme expliqué ci-dessus, cette factorisation est naturellement effectuée en utilisant des outils externes au système de preuves. Le logiciel construisant le certificat joue donc le rôle d'un oracle dont la prédiction est toutefois soigneusement vérifiée. Ce logiciel a été écrit par Benjamin Grégoire sous forme d'un programme C utilisant les bibliothèques ECM [109] et GMP [100]. Étant donné un nombre n, ce programme engendre un fichier en syntaxe Coq dont les lemmes ont la forme suivante :

```
Lemma prime_n : prime n.
Proof.
 apply (Pock_refl (Pock_certif n a d sqrt) l).
 exact_no_check (refl_equal true).
Qed.
```

où a est le témoin du certificat de Pocklington, d une factorisation partielle de $n-1$, sqrt la racine carrée de r^2-8s (lorsque r^2-8s est positif et 1 dans le cas contraire) et l une liste de pré-certificats prouvant que chacun des éléments de l est premier. La preuve commence par une application du théorème calculatoire Pock_refl. Au moment de la vérification de la preuve, le système calcule $C((n,a,d,\mathsf{sqrt})::l)$ et vérifie ainsi si la proposition $C((n,a,d,\mathsf{sqrt})::l) = \mathsf{true}$ est convertible avec la proposition $\mathsf{true} = \mathsf{true}$. De fait, refl_equal true est simplement la preuve canonique d'égalité de $\mathsf{true} = \mathsf{true}$.

Cette dernière étape est réellement la partie calculatoire où la déduction est remplacée par du calcul. On peut bien voir que les étapes de calcul n'apparaissent pas dans le terme de preuve. Même si ce dernier est au final un peu plus grand que celui décrit dans la section 3.1.2, il ne grandit lui aussi que très peu avec n. De fait, un petit facteur constant mis à part, la taille de la preuve est essentiellement celle du certificat $(n,a,d,\mathsf{sqrt})::l$.

Un point technique propre à Coq : lors de la construction certificat on demande au système de ne pas vérifier tout de suite la convertibilité de $C((n,a,d,\mathsf{sqrt})::l) = \mathsf{true}$ avec $\mathsf{true} = \mathsf{true}$. Cette vérification n'est faite qu'une seule fois lors de l'étape de validation finale de la preuve (Qed). La capacité du noyau de Coq de calculer $C((n,a,d,\mathsf{sqrt})::l)$ est cruciale en particulier pour le temps de vérification.

3.5.1 Construction de certificats pour des nombres premiers arbitraires

La tâche difficile étant la factorisation partielle, l'oracle prend des options en arguments, indiquant quelle "recette" utiliser. Les options principales sont :

`pocklington [-v] [-o` *filename*`] [` *prime* `| -next` *num* `]`

`pocklington` est le programme-oracle qui engendre un certificat pour le nombre *prime* ou le plus petit nombre premier supérieur à *num*. L'option `-v` enclenche le mode verbeux, `-o` permet de choisir le fichier de sortie (un nom est crée si aucun nom est donné).

Le programme vérifie d'abord si le nombre n a de bonnes chances d'être premier, puis il essaye de trouver une factorisation partielle de son prédécesseur. Pour obtenir cette factorisation, il essaye d'abord d'obtenir tous les petits facteurs par des divisions par 2, 3, 5 et 7 puis par tous les nombres suivant qui ne sont pas multiples de 2, 3, 5 et 7. Cette étape s'arrête au maximum de un million et $\log_2(n)^2$. Ensuite, si la factorisation partielle obtenue alors est encore inférieure à la racine cubique de n, l'oracle tente de trouver d'autres facteurs en utilisant la bibliothque ECM.

La librairie ECM propose trois méthodes pour trouver des facteurs premiers. L'oracle tente d'appliquer ces méthodes suivant une heuristique que nous ne détaillons pas ici.

Lorsqu'une factorisation partielle suffisante a été trouve, l'oracle tente de déterminer un témoin a en essayant simplement les entiers successifs 2, 3, 4...Enfin, il essaye récursivement de construire des certificats pour les entiers de la factorisation dont la primalité n'est pas encore certifiée.

Pour éviter de trop répéter les mêmes tâches, les certificats des 5000 premiers entiers (de 2 à 48611) sont construits une fois pour toute dans un fichier séparé `BasePrimes.v`.

Avec ces techniques, l'oracle est capable d'engendrer des certificats pour la plupart des nombres premiers jusqu'à 100 chiffres décimaux et pour certains nombres plus grands.

3.5.2 Construction de certificats pour les nombres de Mersenne

Les nombres de Mersenne sont ceux de la forme 2^n-1. Ils ne sont pas tous premiers ; les plus petits correspondent à $n = 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127$. Actuellement, on connaît seulement 44 nombres de Mersenne premiers et c'est une question ouverte de savoir s'il en existe une infinité ou pas. Le plus grand nombre premier connu actuellement (découvert en 2006) est $2^{32582657}-1$ dont l'écriture comporte 9808358 chiffres décimaux...

Le théorème de Pocklington est bien adapté aux nombres de Mersenne. Une première remarque est qu'un nombre de Mersenne s'écrit $1111\dots1111$ en base 2. Une seconde remarque est que la factorisation de son prédécesseur contient toujours 2, puisque $(2^n-1)-1 = 2(2^{n-1}-1)$. Aussi, factoriser le prédécesseur d'un nombre de Mersenne revient à factoriser $2^{n-1}-1$.

On peut alors utiliser quelques propriétés arithmétiques simples pour commencer la factorisation :

- $2^{2p} - 1 = (2^p - 1)(2^p + 1)$
- $2^{3p} - 1 = (2^p - 1)(2^{2p} + 2^p + 1)$

L'oracle utilise ces astuces récursivement pour débuter la factorisation, ce qui réduit considérablement la taille du nombre à factoriser. Puisque $2^n - 1$ ne peut être premier que lorsque n est impair, on sait que $n-1$ est pair et on peut donc toujours utiliser la première remarque. Lorsqu'aucune des deux remarques ne peut être utilisée, l'oracle recourt aux méthodes génériques décrites ci-dessus pour poursuivre la factorisation.

La syntaxe pour utiliser ces "recetets" est `pocklington -mersenne` n, qui demande de chercher le certificat pour $2^n - 1$. On arrive ainsi à construire des certificats pour les 15 premiers nombres premiers de Mersenne ; le plus grand correspond à $n = 1279$ et se compose de 386 chiffres décimaux.

Lorsque n est encore plus grand, l'oracle ne suffit pas à trouver les factorisations des nombres obtenus. On peut alors indiquer à l'oracle des factorisations partielles en donnant le nom d'un fichier qui contient un nombre premier est la factorisation partielle de son prédécesseur : `pocklington -dec` *file*.

En utilisant cette possibilité et des tables de factorisation disponible publiquement[4] on arrive à construire un certificat pour les $16^{\text{ème}}$ et $17^{\text{ème}}$ nombres de Mersenne ($n = 2203$ et $n = 2281$).

3.5.3 Aller à la limite

Des difficultés similaires apparaissent pour le $18^{\text{ème}}$ nombre de Mersenne ($n = 3217$). En utilisant des astuces spécifiques, Laurent Théry et Benjamin Grégoire ont été capables de construire un certificat vérifiable par Coq pour ce nombre de 969 chiffres. Ce nombre premier avait été découvert en 1957.

On peut remarquer que Pocklington n'est pas considéré comme le moyen le plus efficace de vérifier la primalité d'un nombre de Mersenne. Le test de Lucas donne un critère plus simple qui ne nécessite pas de factorisation :

Théorème 3 *Soit (S_n) récursivement défini par $S_0 = 4$ et $S_{n+1} = S_n^2 - 2$, pour $n > 2$. $2^n - 1$ est premier si et seulement si $(2^n - 1) | S_{n-2}$.*

Ce théorème a également été formalisé en Coq pour comparer le temps de vérification avec celui des certificats de Pocklington pour les nombres de Mersenne.

A cette fin on ajoute une nouvelle entrée `Lucas_certif` dans le type des pré-certificats :

```
Inductive pre_certif : Set :=
 | Pock_certif : forall n a : positive, dec_prime -> pre_certif
 | Proof_certif : forall n : positive, prime n -> pre_certif
 | Lucas_certif : forall n p : positive, pre_certif.
```

où n doit être égal à $2^p - 1$.

Pour générer des certificats de Lucas pour les nombres de Mersenne, on ajoute une nouvelle option à l'oracle : `pocklington -lucas` p.

3.6 Performances

Tous les certificats décrits ci-dessous sont vérifiables en Coq. On peut trouver les fichiers chez Benjamin Grégoire :

`http://www-sop.inria.fr/everest/Benjamin.Gregoire/primnumber.html`

Les certificats sont engendrés par le programme pocklington (1721 lignes de C). La vérification nécessite 6653 lignes de Coq. Voici quelques temps d'exécution datant de 2006 (processeur Pentium 4 (3.60 GHz) 1Gb de RAM). On utilise bien sûr le mécanisme de compilation de Coq [57].

La figure 3.1 donne les temps pour construire les certificats des 100.000 premiers nombres premiers, la taille moyenne des certificats et le temps de vérification en Coq. En moyenne, la génération d'un certificat

[4] `http ://homes.cerias.purdue.edu/~ssw/cun/prime.php`

from - to	build	size	verify
2 - 5000	0.15s	989K	35.85s
5001 - 10000	0.17s	1012K	42.59s
10001 - 20000	0.38s	2.1M	134.14s
20001 - 30000	0.38s	2.1M	138.30s
30001 - 40000	0.38s	2.1M	145.81s
40001 - 50000	0.38s	2.2M	153.65s
50001 - 60000	0.41s	2.2M	153.57s
60001 - 70000	0.43s	2.2M	158.13s
70001 - 80000	0.39s	2.2M	160.07s
80001 - 90000	0.40s	2.2M	162.58s
90001 - 100000	0.44s	2.2M	162.03s

Fig. 3.1 – Temps de vérification pour les 100.000 premiers nombres premiers

pour un petit nombre premier est de l'ordre de 3.10^{-5} secondes, leur taille de 210 octets et leur vérification est de lordre de 0,015 secondes.

Dans le travail précédent de Oostdijk et Caprotti, un logiciel dédié construit également un certificat de Pocklington. Toutefois, la fonction de vérification n'a pas de statut explicite. A la place, le logiciel utilise le certificat pour construire une preuve (sous forme de script Coq) de primalité, qui va invoquer plusieurs fois le théorème de Pocklington. En conséquence, les preuves peuvent devenir beaucoup plus longue que chez nous, lorsque les entiers traités grandissent.

La figure 3.2 fait une comparaison avec cette approche déductive pour des nombres premiers particuliers. Le nombre $(2^{148}+1)/17$ composé de 44 chiffres était le plus grand prouvé premier en Coq jusqu'alors.

Comme on pouvait s'y attendre, l'approche calculatoire réduit considérablement la taille des preuves. Pour $(2^{148}+1)/17$, on gange un facteur de 1500 en remplaçant déduction par calcul.

Les trois dernières colonnes comparent les temps de vérification pour les différentes approches. Sans utilisation de la machine virtuelle (cad. sans compilation) l'approche calculatoire est un peu plus rapide

		size		temps		
premier	chiffres	deduc.	refl.	deduc.	refl.	refl. + VM
1234567891	10	94K	0.453K	3.98s	1.50s	0.50s
74747474747474747	17	145K	0.502K	9.87s	7.02s	0.56s
1111111111111111111	19	223K	0.664K	17.41s	16.67s	0.66s
$(2^{148}+1)/17$	44	1.2M	0.798K	350.63s	338.12s	2.77s
P_{200}	200	_	2.014K	_	_	190.98s

Fig. 3.2 – Comparaison avec la méthode non récursive

#	n	chiffres	année	découvreur	certificat	temps	temps(Lucas)
8	31	10	1772	Euler	0.527K	0.51s	0.01s
9	61	19	1883	Pervushin	0.648K	0.66s	0.08s
10	89	27	1911	Powers	0.687K	0.94s	0.25s
11	107	33	1914	Powers	0.681K	1.14s	0.44s
12	127	39	1876	Lucas	0.775K	2.03s	0.73s
13	521	157	1952	Robinson	2.131K	178.00s	53.00s
14	607	183	1952	Robinson	1.818K	112.00s	84.00s
15	1279	386	1952	Robinson	3.427K	2204.00s	827.00s
16	2203	664	1952	Robinson	5.274K	11983.00s	4421.00s
17	2281	687	1952	Robinson	5.995K	44357.00s	4964.00s
18	3217	969	1957	Riesel	7.766K	94344.00s	14680.00s
19	4253	1281	1961	Hurwitz	_	_	35198.00s
20	4423	1332	1961	Hurwitz	_	_	39766.00s

FIG. 3.3 – Temps de vérification des nombres de Mersenne

mais les temps sont du même ordre. En utilisant la compilation lors de la vérification (donc pour calculer le résultat de la fonction C_c) on gagne en temps un facteur 9 pour les petits nombres et jusqu'à 120 pour les plus grands. C'est évidemment un résultat satisfaisant pour la combinaison de la compilation et de l'approche calculatoire.

Le développement décrit ici permet de vérifier en Coq la primalité d'un nombre premier "aléatoire" de 200 chiffres :

$$\begin{aligned} P_{200} = \quad & 67948478220220424719000081242787129583354660769625 \\ & 17084497493695001130855677194964257537365035439814 \\ & 34650243928089694516285823439004920100845398699127 \\ & 45843498592112547013115888293377700659260273705507 \end{aligned}$$

en 191 secondes (en 2006) avec une preuve de taille 2K.

En pratique il est difficile de trouver des facteurs premiers de plus de 35 chiffres. La plupart des nombres de moins de 100 chiffres contiennent suffisamment de facteurs premiers de moins de 20 chiffres pour que ECM les trouve rapidement. Pour les nombres plus grands, être capables de trouver des facteurs premiers est une question de chance ; $n-1$ doit comporter de nombreux facteurs premiers de petite taille. C'est le cas de P_{200}.

La figure 3.3 donne les temps de vérification des 18 premiers nombres de Mersenne (avec des certificats de Pocklington) à l'exception des 7 premiers qui font partie des 100.000 plus petits nombres premiers. Le plus grand est composé de 969 chiffres.

Avec des certificats de Lucas on arrive à prouver la primalité du 20$^{\text{ème}}$ nombre de Mersenne (1332 chiffres décimaux).

3.7 Développements récents

Le travail décrit dans ce chapitre date d'il y a moins de deux ans, et pourtant les performances qui y sont décrits ont été récemment nettement dépassées. Je considère que c'est un très bon signe, car si ces nouveaux travaux perfectionnent ce qui est décrit ici, ils s'inscrivent tout à fait dans le prolongement. Je reviens sur ces améliorations dans le dernier chapitre ; elles-portent essentiellement sur :

- Les outils mathématiques : Laurent Théry a, avec Guillaume Hanrot, formalisé en Coq les théorèmes sur les courbes elliptiques et la primalité.
- La représentation des grands nombrs en Coq avec la librairie de Benjamin Grégoire et Laurent Théry.
- La possibilité d'exploiter l'arthmétique du processeur, avec le travail d'Arnaud Spiwack que j'ai suggéré et encadré.

Chapitre 4

Le théorème des quatre couleurs

Le théorème des quatre couleurs doit sans doute sa renommée à la simplicité et au caractère concret de son énoncé : il peut être expliqué facilement à un non-mathématicien. Cela a suffit, et suffit encore, à attiser la curiosité d'innombrables amateurs, qui ont tenté, et tentent encore, de proposer des "preuves" élémentaires. De plus, les seules preuves connues sont fortement calculatoires, ce que l' énoncé ne laisse pas présager à première vue. Ce recours à l'ordinateur, utilisé une première fois en 1976, à une époque où ces machines étaient bien moins répandues qu'aujourd'hui, à sans doute également fait beaucoup pour l' halo de mystère qui semble encore entourer ce résultat.

Ce chapitre se veut une introduction à ce résultat mathématique bien particulier en général, et à sa preuve formelle en Coq en particulier.

4.1 Historique

La première observation connue du phénomème remonte à 1852. Francis Guthrie, cartographe britannique remarque qu'il arrive à colorier toutes les cartes dont il dispose avec seulement quatre couleurs, et ce sans que deux régions contiguës ne se voient attribué la même couleur. On dit que la carte à l'origine de cette observation aurait été la carte des contés britanniques de l'époque ; on peut voir une illustration en figure 4.1. Pour être tout à fait précis :

- Une région est une partie connexe du plan,
- une carte est un ensemble de régions deux-à-deux distinctes,
- deux régions sont contiguës si l'intersection de leurs adhérences comporte au moins un point qui n'appartient pas à l'adhérence d'une autre région.

Le dernier point permet est important. Sans lui, toute tarte coupée en plus de quatre parts constituerait un contre-exemple.

De fait, on s'intéresse généralement à la formulation du théorème en termes de *graphes*. Chaque "pays" étant ramené à un sommet, une arête correspondant à une frontière commune. Le théorème peut alors être simplement formulé ainsi :

> Pour tout graphe planaire, il est possible d'assigner à chaque sommet une couleur parmi quatre, de manière à ce qu'aucune arête ne joigne deux sommets de même couleur.

Il faut toutefois noter que passer du problème de coloriage de cartes à celui de graphes n'est pas si simple que cela. Sa formalisation ayant permis d'en éclairer quelques détails délicats, facilement ignorés.

Si Guthrie est resté dans l'Histoire, c'est essentiellement parce qu'on lui doit d'avoir attiré l'attention de la communauté mathématique sur cette question. La gazette retient que :

1852 Guthrie mentionne la question à de Morgan.
1878 Première référence écrite à la conjecture par Cayley.
1879 Première preuve par Kempe [68].
1880 Variante de la preuve par Tait [98].

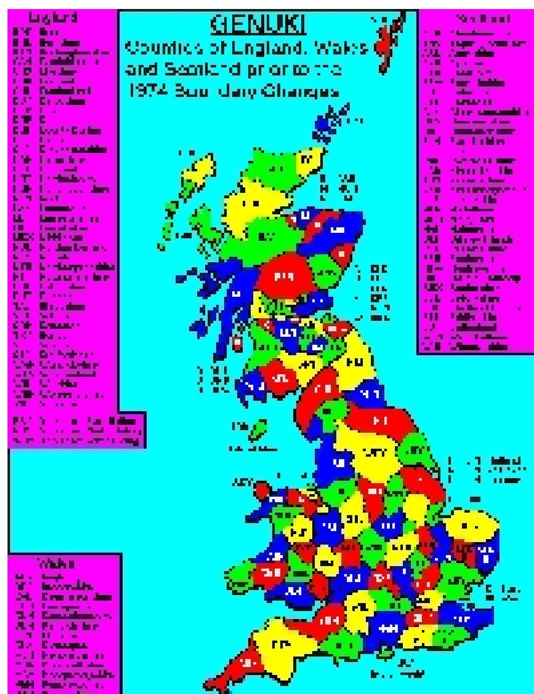


Fig. 4.1 – La carte des Contés Britanniques avant 1974 (avec l'autorisation de GENUKI).

1890 Heawood trouve une erreur dans la preuve de Kempe ; un an plus tard Petersen trouve une erreur dans celle de Tait. La preuve de Kempe reste toutefois valable pour prouver le théorème des cinq couleurs.

1913 Birkhoff propose la notion de *configuration réductible* [15]. C'est la notion centrale de toutes les preuves connues.

1925 En utilisant la notion de réductibilité, Franklin montre que tout contre-exemple est composé de plus de 25 régions. Cette borne sera progressivement améliorée au cours du siècle.

1969 Heinrich Heesch propose une approche pour résoudre le problème et suggère l'utilisation de l'ordinateur pour aboutir [61].

1976 Kenneth Appel et Wolfang Haken annoncent avoir vérifié le théorème [69, 70, 71]. La preuve considère 1476 configurations, et la vérification a demandé 1200 heures de calcul.

1995 Les théoriciens des graphes Robertson, Seymour et Sanders présentent une variante de la preuve précédente [93]. Elle ne considère que 633 configurations, et les conditions que celles-ci doivent vérifier sont un peu plus simples. La vérification prend alors environ une heure sur un PC (une dizaine de minutes aujourd'hui).

On peut remarque au passage que la plupart des contributions à la question sont dues à des mathématiciens anglo-saxons, à l'exception des allemands Heesch et Haken.

4.2 La preuve fausse de Kempe

La preuve fausse de Kempe est importante car elle comporte deux idées également essentielles aux preuves "modernes". D'une part qu'il faille trouver le "point faible" du graphe, à savoir un endroit où la densité d'arêtes est faible, et surtout l'idée des permutations de couleurs dans une composante connexe. Cette opération est d'ailleurs encore aujourd'hui désignée comme l'utilisation des "chaînes de Kempe".

4.2.1 Triangulation

La première remarque est qu'il suffit de résoudre le problème dans le cas de graphes triangulés. En effet, on peut :

1. Effacer les arêtes parallèles : lorsque plusieurs arêtes joignent les deux mêmes sommets, l'on en garde qu'une. La contrainte sur les coloriages restant évidemment inchangée.
2. chaque fois qu'une région est bordée de plus de trois sommets, on choisit parmi ceux-ci deux sommets qui ne soient pas déjà voisins, et on les joint par une arête supplémentaire. Ce faisant, on rend simplement le problème de coloriage plus difficile : tout coloriage du nouveau graphe sera aussi un coloriage du graphe original.

Si l'on préfère penser en termes de cartes, les graphes triangulaires correspondent à ce que l'on appelle les *cartes cubiques*. C'est-à-dire qu'elles ne comportent pas de "trous" et surtout qu'il n'y a pas de point où plusieurs frontières se rejoignent.

[PETIT DESSIN]

4.2.2 La formule d'Euler

En triangulant le graphe, on s'est donc placé d'emblée dans le cas le plus difficile. L'intérêt est que l'on peut alors trouver le "point faible" du graphe. On connaît en effet depuis Euler la formule reliant, pour un graphe planaire connexe, le nombre de sommets s, le nombre d'arêtes a et le nombre de faces f :

$$a + 2 = s + f$$

Or lorsqu'un graphe est triangulé, on a de plus $f = 2a/3$, puisque chaque face "voit" 3 demi-arêtes. La formule d'Euler devient alors :

$$a + 2 = s + \frac{2a}{3}$$

c'est-à-dire

$$s = \frac{a}{3} + 2$$

Le degré d'un sommet étant le nombre d'arêtes qui le joignent, le degré moyen pour un tel graphe devient $2a/s$ c'est-à-dire :

$$\overline{d} = 6 - \frac{12}{a+2}$$

Le point crucial est alors simplement que, puisque le degré moyen est strictement inférieur à 6, il existe au moins un sommet du graphe dont le degré est au plus 5. C'est ce point-là que l'on regarde de plus près.

Le déroulement de la preuve de Kempe est alors :

1. On montre par récurrence sur s que tout graphe planaire de s sommets est 4-coloriable.
2. On suppose la propriété vraie pour $s - 1$ et on considère un graphe de s sommets. On a vu que l'on peut trianguler ce graphe en gardant constant le nombre de sommets.
3. Ce graphe triangulé comporte au moins un sommet de degré inférieur ou égal à 5. On supprime ce sommet et les arêtes qui le joignent ; on colorie le graphe obtenu par hypothèse de récurrence.
4. On replace le sommet supprimé. Si son degré est inférieur ou égal à 3, il est évidemment possible de lui assigner une couleur qui n'a pas encore été attribuée à l'un de ses voisins.
5. Restent alors à traiter les cas où ce sommet a quatre ou cinq voisins ; c'est l'objet des paragraphes suivants.

4.2.3 Le sommet de degré quatre

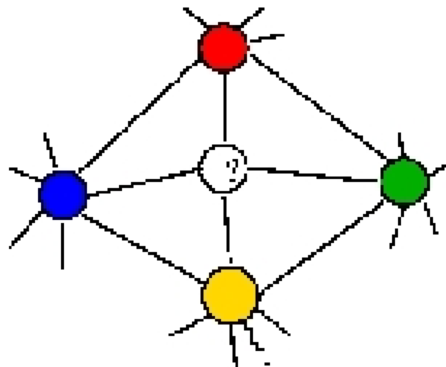
Plus précisément, le seul cas problématique est celui où le coloriage obtenu par hypothèse de récurrence assigne quatre couleurs distinctes aux quatre voisins du sommet considéré :

C'est là que se situe la seconde idée remarquable de Kempe : lorsque l'on considère une *composante connexe bicolore*, il est possible d'inverser les deux couleurs à l'intérieur de cette composante, en préservant la correction du 4-coloriage :

Définition 3.1 *On considère un graphe 4-colorié. Une composante bi-colore connexe est un sous graphe connexe tel que :*

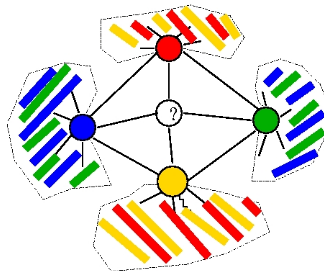
1. *ses sommets sont tous coloriés de deux couleurs, qu'on appellera A et B,*
2. *les sommets du graphes qui sont voisins d'un sommet de la composante, mais ne font pas partie de la composante ne sont pas coloriés par A ou B.*

Considérons maintenant le cas d'un graphe comportant un sommet de degré 4. Par hypothèse de récurrence, on a colorié tous les sommets sauf celui-là, et ses voisins se sont vus affecter chacune des 4 couleurs



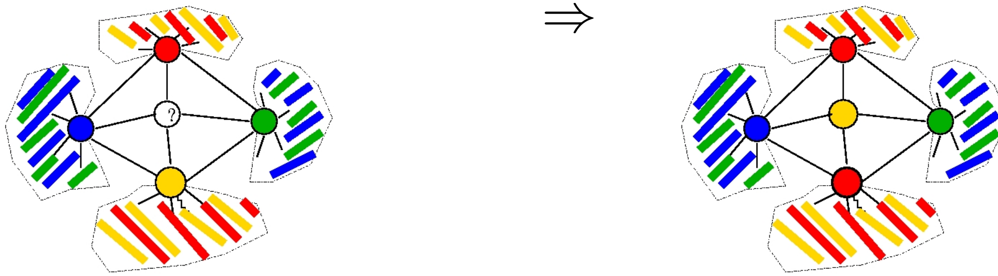
disponibles :

L'idée est alors de considérer les composantes bi-colores pour les couleurs se faisant face ; dans notre cas

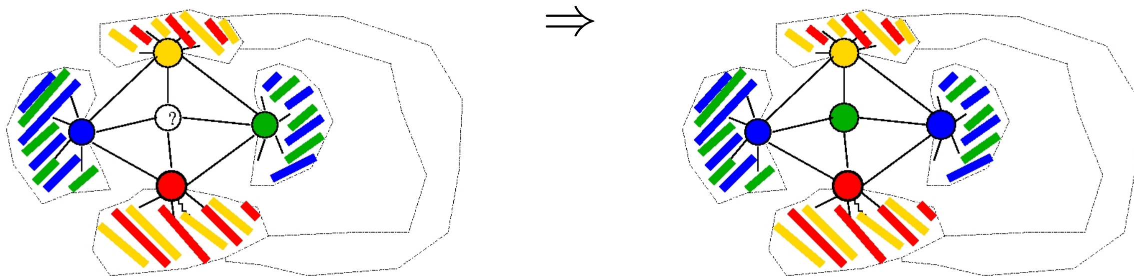


rouge et jaune d'une part, bleu et vert de l'autre.

Si les deux composantes rouge et jaune apparentes sont distinctes, on peut inverser les couleurs à l'intérieur de la première sans affecter la seconde. On "libère" ainsi une couleur pour le sommet central :



Si en revanche ces deux composantes rouge/jaune sont identiques, alors la planarité impose que les deux composantes vert et bleu ne le sont pas. On peut donc effectuer une inversion en conséquence :

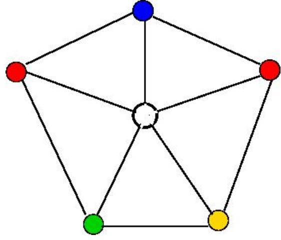


Ceci achève, correctement, le cas du sommet à degré quatre.

4.2.4 Le sommet de degré cinq

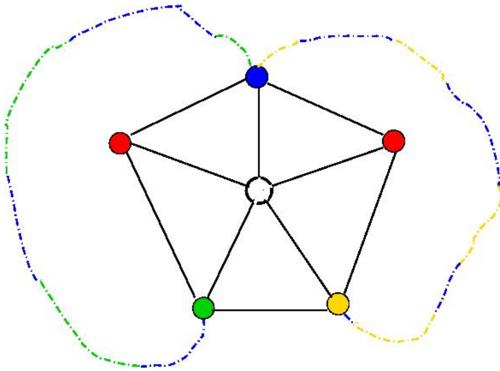
L'erreur de Kempe a été de croire qu'un raisonnement presque aussi simple permettait de traiter le cas d'un graphe dont l'un des sommets est de degré 5. Même si les détails ont surtout un intérêt historique ou anecdotique, il est amusant de regarder son argument dans le détail.

Le seul cas problématique est si les 5 voisins se sont vus affecter les 4 couleurs disponibles. Par symétrie, il est possible de se ramener au cas de figure suivant.

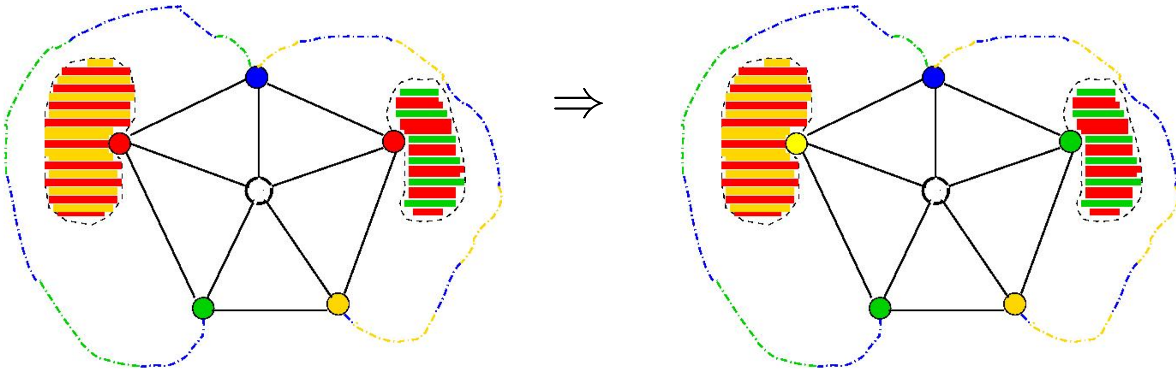


Kempe dit alors :

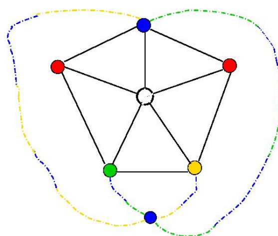
1. Si a et c ne sont pas dans la même composante rouge/jaune, on inverse les couleurs dans l'une des deux et on libère la couleur correspondante pour le sommet central.
2. Sinon, si a et d ne sont pas dans la même composante rouge/verte, on fait de même.
3. reste alors le cas correspondant au dessin suivant.



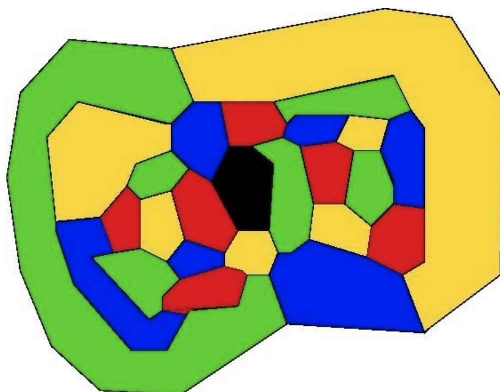
   Kempe pense alors que la composante bleu/jaune contenant e est disjointe de la composante bleu/verte contenant b. Si c'est le cas, on peut en effet inverser ces deux composantes et libérer ainsi la couleur bleu pour le sommet central.



Malheureusement, la situation 3 peut aussi correspondre au dessin suivant :



et dans ce cas les deux composantes bicolores rouge/jaune et rouge/vertes ne sont pas forcément disjointes. Les inversions prévues ne peuvent alors pas nécessairement se faire. Percy Heawood mis en évidence l'erreur de Kempe en présentant une carte coloriée à l'exception d'un sommet de degré 5, et où aucune suite d'inversions ne permet de finir le coloriage. Voici le contre-exemple de Heawood :



4.3 La contribution de Tait

Alors que la preuve de Kempe n'avait pas encore été invalidée, Tait proposa alors une variante de celle-ci, qui identifiait un outil qui allait se révéler décisif par la suite.

Définition 3.2 *Une quasi-triangulation, ou graphe quasi-triangulé, est un graphe planaire dont toutes les faces sauf, au plus, une sont bordées de trois arêtes exactement. La face qui n'est pas bordée par trois arêtes est appelée l'*extérieur *de la quasi-triangulation.*

La remarque de Tait est simplement que lorsque l'on considère un graphe planaire quasi-triangulé, c'est-à-dire dont toutes les faces, sauf éventuellement une, sont des triangles, il est équivalent de 4-colorier ses sommets et de 3-colorier les *arêtes*, de telle manière à ce que toute face intérieure soit bordée par les trois couleurs. C'est-à-dire que chacune des trois arêtes de chaque face intérieure se voit affectée une couleur différentes des deux autres.

Il suffit pour cela de définir la couleur d'une arête à partir de celles de ses extrémités par une fonction f qui soit :

– symétrique en ses deux arguments, $f(x, y) = f(y, x)$,
– telle que $x \neq y \Rightarrow f(x, y) \in \{1; 2; 3\}$,
– telle que $x \neq y \Rightarrow f(x, z) \neq f(y, z)$.

Une telle fonction est, par exemple définie comme :

$$\begin{aligned} f(0, x) &\equiv x \\ f(1, 2) &\equiv 3 \\ f(1, 3) &\equiv 2 \\ f(2, 3) &\equiv 1 \\ f(x, y) &\equiv f(y, x) \text{ pour les autres cas} \end{aligned}$$

Il est immédiat qu'un 4-coloriage des sommets induits un 3-coloriage des arêtes. Le fait qu'un 3-coloriage permette de retrouver le 4-coloriage sous-jacent est du au fait que :

1. Si l'on connait la couleur x d'un sommet et la couleur $f(x, y)$ de l'arête, alors y est fixée,
2. la somme des couleurs des arêtes d'un cycle est nulle dans $\mathbb{Z}/4\mathbb{Z}$.

Il suffit donc de connaitre $f(x, y)$ pour tous les sommets x et y (cad. toutes les couleurs des arêtes) et la couleur x d'un sommet (en fait un sommet par composante connexe) pour retrouver la couleur de chaque sommet.

Inversion dans les tri-coloriages

Un intérêt de la remarque de Tait est qu'elle permet de bien comprendre la combinatoire des inversions de couleurs dans les quasi-triangulation.

Un trois coloriage sur une quasi-triangulation induit immédiatement un trois-coloriage sur le cycle formé par les arêtes de l'extérieur, comme celui de la figure 4.3.

On peut alors choisir une *couleur pivot*, par exemple rouge. Considérons que les arêtes rouges sont des murs infranchissables.

1. Si l'on est à l'extérieur de la quasi-triangulation en face d'une arête non rouge, par exemple verte (respectivement bleue), on peut franchir cette arête.
2. On se trouve alors à l'intérieur d'une face triangulaire. bordée par une arête rouge infranchissable, une verte (respectivement bleue) déjà franchie et une bleue (respectivement verte).
3. Si l'on s'interdit également de revenir sur ses pas, on n'a donc pas le choix et l'on doit franchir l'arête bleue (respectivement verte).
4. On se retrouve alors soit à nouveau à l'extérieur, auquel cas on a finit le parcours, soit dans une autre face triangulaire, auquel cas on est à nouveau dans la situation 1 et on peut ré-itérer le processus.

Il est facile de se convaincre qu'au cours d'un tel parcours on ne va jamais passer deux fois par une même face. On finit donc toujours par retrouver l'extérieur de la quasi-triangulation. Qui plus est, on "sort" de la quasi-triangulation par une arête verte ou bleue distincte de celle par laquelle on a commencé le parcours.

On voit également que l'on peut faire le même parcours à l'envers : si l'on commence par l'arête de sortie, on terminera par l'arête d'entrée. *Autrement dit, étant donné un trois-coloriage, le choix d'une* couleur pivot *induit un appareillement deux-à-deux entre les arêtes qui ne sont pas de cette couleur pivot.*

On voit un exemple d'appareillement signé en figure 4.3.

On peut maintenant faire trois remarques cruciales :

– Les arêtes qui ne sont pas rouges sont appareillées deux-à-deux. Elles sont donc en nombre pair. Autrement dit, le nombre d'arêtes rouges a la même parité que le nombre d'arêtes du bord. On peut évidemment conclure la même chose pour le nombre d'arêtes bleues et le nombre d'arêtes vertes, en changeant la couleur pivot.
– Un parcours tel que décrit ci-dessus passe alternativement par des arêtes vertes et bleues. Il est possible d'inverser ces deux couleurs pour l'ensemble du parcours et préserver la propriété de trois-coloriage. En d'autres termes, il est possible d'inverser simultanément les couleurs de deux arêtes appareillées (pour un choix de couleur pivot donné). Ici, suivant la parité de la longueur du parcours, ces deux arêtes seront soit simultanément vertes ou bleues, ou au contraire l'une verte et l'autre bleue.
– Les parcours appareillant deux arêtes vertes et/ou bleues sont forcément distincts. *Ils ne peuvent donc se couper.* La planarité de la quasi-triangulation assure alors que les appareillements ne peuvent être alternés.

La dernière remarque permet donc de décrire l'appareillement entre arêtes qui est induit par un coloriage et le choix d'un pivot comme un *mot* bien parenthésé dans l'alphabet suivant :

$$\{'(\ ; \ ')^{-} \ ; \ ')^{+} \ ; \ '_\}$$

Dans ce mot chaque caractère correspond à une arête du bord. Le caractère _ correspondant à une arête pivot (ici rouge), les parenthèses aux arêtes appareillées (ici bleues et vertes). Le signe + ou − sur la parenthèse fermante signalant si ces arêtes sont respectivement de même couleur ou de couleurs différentes.

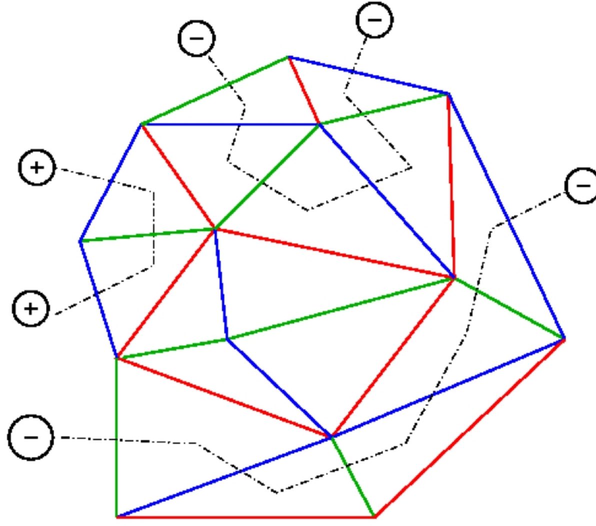


FIG. 4.2 – Appareillement signé entre arêtes bleues et vertes induit par un trois-coloriage.

Ces mots seront appelés des *chromogrammes*. Cette jolie dénomination est due à Georges Gonthier.

Bien sûr, pour passer d'un trois-coloriage à un chromogramme, il faut aussi choisir une arête sur le bord qui correspondra au premier caractère. Par exemple, le trois-coloriage de la figure 4.3 induit les chrommogramme $()^{-}(_\,_)^{-}()^{+}$ si l'on prend le sens des aiguilles d'une montre en partant de l'arête verte du haut.

Une dernière remarque importante est que les contraintes de parité font que le nombre de parenthèses $)^{-}$ a également la même parité que la longueur du mot et le nombre de _.

4.4 Recollement

Considérons deux quasi-triangulations, qui ne partagent que leur face externe. On dira qu'elles sont *complémentaires*. Leur réunion est évidemment une triangulation. De plus, cette triangulation est trois-coloriables (pour les arêtes) si et seulement si il existe un trois-coloriage pour chaque quasi-triangulation, et que ces trois-coloriages coïncident sur les arêtes de la face externe (commune).

La réductibilité, étudiée dans le paragraphe suivant correspond à une propriété forte de certaines quasi-triangulations : quelque soit la quasi-triangulation complémentaire, pourvu que cette dernière soit trois-coloriable, il est toujours possible de trouver un trois-coloriage de la réunion.

4.5 Réductibilité

Appelons n le nombre d'arêtes du bord. Un coloriage c du bord est donc un mot de longueur n composé de trois couleurs. Étant donné un chromogramme w et une couleur pivot p, on définit la relation $w \simeq_p c$ par les clauses :

1. $[] \simeq_p []$ où $[]$ désigne à la fois le coloriage et le chromogramme vide.
2. Si $w \simeq_p c$ alors $w_ \simeq_p cp$.
3. Si a est une couleur distincte de p et $w \simeq_p c$, alors $(w)^{+} \simeq_p aca$.
4. Si a et b sont deux couleurs distinctes et différentes de p et $w \simeq_p c$, alors $(w)^{-} \simeq_p acb$.

On dit alors que le coloriage c *convient* au chromogramme w pour le pivot p.

La remarque simple mais cruciale est alors que : si la quasi-triangulation admet un trois-coloriage qui, lorsque l'on choisit le pivot p, a le même appareillement par le chromogramme w, alors quelque soit c et p' tel que $w \simeq_p c$, il est possible de re-colorier la quasi-triangulation pour obtenir un trois-coloriage induisant c sur le bord.

Soit maintenant un ensemble $\mathcal{C}$ de coloriages du bord, dont on suppose qu'ils peuvent tous être obtenus à partir de trois-coloriages corrects. Soit maintenant un coloriage c tel que :

$$\exists p. \forall w. w \simeq_p c \Rightarrow \exists c' \in \mathcal{C}. w \simeq_p c'.$$

Autrement dit, on peut choisir un pivot tel que, quel que soit le chromogramme correspondant au trois-coloriage sous-jacent, il est toujours possible de se ramener à un coloriage de $\mathcal{C}$.

Alors, si l'on veut recoller la quasi-triangulation avec une autre quasi-triangulation K qui admet un trois-coloriage induisant c sur le bord, on comprend que sous la condition précédente, il est possible de re-colorier K pour aboutir à un coloriage du bord qui soit élément de $\mathcal{C}$.

On peut évidemment itérer ce processus. On arrive ainsi au définitions suivantes, étant donné une quasi-triangulation K_0 :

- Soit $\mathcal{C}_0$ l'ensemble des coloriages du bord correspondant à des trois-coloriages corrects de K_0.
- Soit $\mathcal{C}_{i+1} \equiv \{c, \exists p, \forall w, w \simeq_p c \Rightarrow \exists c' \in \mathcal{C}_i, w \simeq_p c'\} \bigcup \mathcal{C}_i$.

Comme la suite des $\mathcal{C}_i$ est croissante pour l'inclusion, elle admet évidemment un point-fixe $\mathcal{C}_\infty$.

On dit alors que la quasi-triangulation est D-réductible si $\mathcal{C}_\infty$ contient tous les coloriages possibles (vérifiant les conditions de parité).

La propriété essentielle est :

Théorème 4 *Soit deux quasi-triangulations K_1 et K_2 ayant des bords de même longueur. Si les K_1 et K_2 sont toutes les deux trois-coloriables et si K_1 est D-réductible, alors le recollement de K_1 et K_2 est trois-coloriable (et donc quatre-coloriable).*

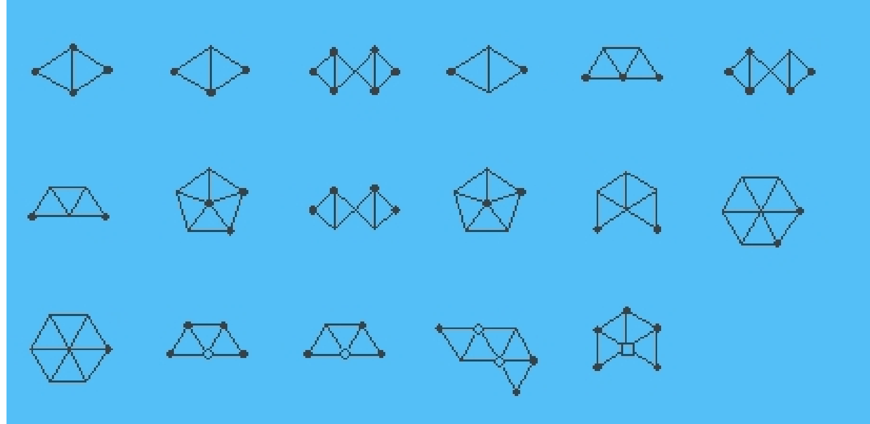
4.6 Les preuves modernes

Avec la notion de réductibilité, on a la clé des preuves modernes (et correctes) du théorème des quatre couleurs : on va exhiber un ensemble de quasi-triangulations qu'on appellera *configurations* tel que :

- Ces configurations soient toutes réductibles.
- l'ensemble des configurations soit *inévitable*, c'est-à-dire que toute triangulation contient soit une configuration, soit vérifie un autre critère garantissant qu'elle est quatre-coloriable.

L'ensemble inévitable de configurations est obtenu en ne cherchant plus le sommet de plus petit degré dans le graphe, mais celui ayant le plus petit nombres de voisins de distance 2. Formellement, ce sommet est décrit à travers un certain nombre de règles de grammaire de graphes qui, lorsqu'elles peuvent être appliquées localement vont déplacer des valeurs de poids d'un sommet à un autre. On cherche alors le sommet de plus grand poids.

L'intérêt de cette présentation sous forme de règles est qu'elle conduit à une méthode utilisable pour effectivement démontrer l'inévitabilité. L'ensemble inévitable, dans la démonstration de 1995, est composé de 633 configurations, dont voici les premières :



C-réductibilité

Pour être précis, seule une minorité de ces configurations est D-réductibles. La plupart sont C-réductibles, c'est-à-dire qu'elles sont munies d'un *contrat* correspondant à une ou deux arêtes marquées en gras sur le dessins. Lorsque l'on calcule $\mathcal{C}_\infty$ on va trouver un certain nombre de coloriages que l'on ne sait pas traiter. On remarque alors que lorsque l'on calcule l'ensemble $\mathcal{C}_\infty$ correspondant à la configuration *après contraction des arêtes du contrat*, on ne sait toujours pas traiter ces coloriages. On en déduit donc que si ces coloriages manquant permettaient de construire un contre-exemple aux quatre couleurs, alors on pourrait faire de même avec la configuration contractée. Ce qui suffit à conclure qu'il ne s'agit pas d'un contre-exemple minimal.

4.7 Réductibilité en Coq

Le programme vérifiant la réductibilité calcule donc, pour chaque configuration, les ensembles $\mathcal{C}_i$ successifs jusqu'à arriver au point-fixe.

4.7.1 Représentation des ensembles $\mathcal{C}_i$

Un coloriage est une séquence de trois couleurs, que nous noterons R, G et B, dont la longueur correspond à la taille du bord de la configuration, c'est-à-dire de 6 à 14 suivant les cas.

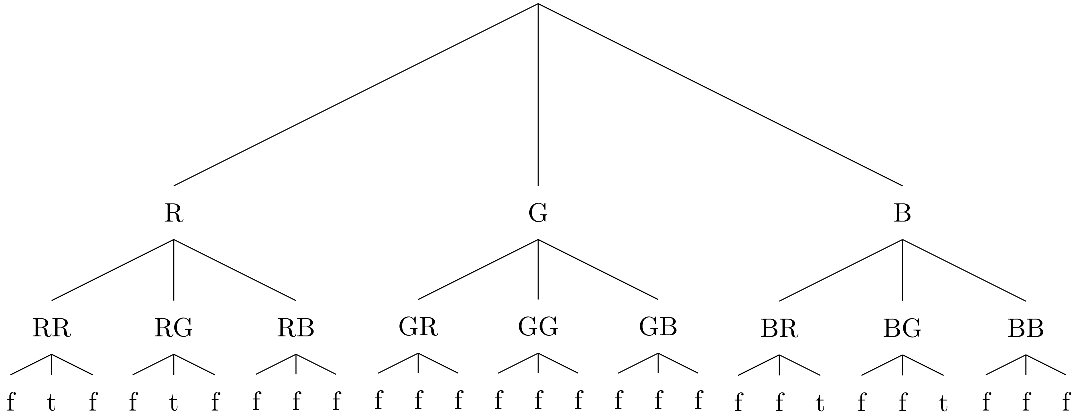
Les coloriages sont naturellement représentés comme des listes de caractères dans l'alphabet à trois lettres. Comme on est dans un cadre purement fonctionnel, on représente les ensembles de telles listes à l'aide d'arbres ternaires :

- Les feuilles sont booléennes, indiquant si le mot appartient, ou pas, à l'ensemble.
- Les noeuds pointent vers les sous-arbres correspondants aux trois débuts de suffixes possibles, R, G et B.

Par exemple l'ensemble

$$\{RRG; RGG; BGB; BRB\}$$

sera représenté par :



On optimisera par symétrie, en considérant que la première couleur est toujours R. De plus, on ne s'intéresse qu'aux coloriages vérifiant la condition de parité : le nombre de R, G et B est de même parité que la longueur du bord. Cela implique que l'on a jamais le choix de la dernière couleur, qu'il est donc inutile d'expliciter. On se ramène donc à des coloriages de longueur 4 à 12.

4.7.2 Algorithme naïf

On commence par calculer $\mathcal{C}_0$ en énumérant tous les coloriages. Il est bien sûr avantageux d'utiliser une bonne stratégie pour cela, mais il ne s'agit pas d'une étape critique. Dans la pratique, on trouve jusqu'à un peu plus de 20.000 éléments à $\mathcal{C}_0$ pour 500.000 coloriages de bonne parité.

On peut remarquer qu'il est possible de calculer $\mathcal{C}_{i+1}$ directement à partir de $\mathcal{C}_i$:

$$\mathcal{C}_{i+1} \equiv \{c, \exists p, \forall w, w \simeq_p c \Rightarrow \exists c' \in \mathcal{C}_i, w \simeq_p c'\} \bigcup \mathcal{C}_i$$

il faut pour cela énumérer tous les mots w (n'appartenant pas déjà à $\mathcal{C}_i$, tous les pivots p possibles, tous les chromogrammes w tels que $w \simeq_p c$, puis vérifier s'il existe une reconfiguration c' du coloriage suivant w telle que $c' \in \mathcal{C}_i$.

Cet algorithme est facile à implémenter et économe en mémoire ; il est en revanche très inefficace. Ce sont les deux dernières étapes qui demandent du calcul. La seconde sera la plus coûteuse, car elle doit être effectuée plus souvent.

4.7.3 Algorithme habituel

On a donc clairement intérêt à garder en mémoire non seulement l'ensemble $\mathcal{C}_i$ courant, mais aussi l'ensemble des chromogrammes correspondants. C'est-à-dire :

$$\mathcal{K}_i \equiv \{w, \exists p, \exists c \in \mathcal{C}_i w \simeq_p c\}.$$

On peut alors re-définir :

$$\mathcal{C}_{i+1} \equiv \{c, \exists p, \forall w, w \simeq_p c \Rightarrow w \in \mathcal{K}_i\} \bigcup \mathcal{C}_i$$

ce qui suggère immédiatement une implémentation plus efficace.

1. On calcule $\mathcal{C}_0$ et on initialise i à 0.
2. On parcourt C_i et pour chaque élément c, pour chaque pivot p, on ajoute tous les chromogrammes w tels que $w \simeq_p c$ à $\mathcal{K}_i$.

3. On parcourt les w que l'on vient d'ajouter ; pour chacun, on parcourt les c' tel que $w \simeq_p c'$. Parmi ceux-ci, on repère ceux tel que tous les $w' \simeq_p c'$ sont déjà éléments de $\mathcal{K}_i$. Ces coloriages c' sont ajoutés à $\mathcal{C}_i$. Une fois cette étape terminée on a calculé C_{i+1}.
4. On peut alors revenir à l'étape 2. Si jamais on n'a ajouté aucun coloriage à $\mathcal{C}_i$, le calcul est terminé.

Bien sûr, on représente alors les ensembles $\mathcal{K}_i$ comme des arbres quaternaires, puisque les chromogrammes peuvent être vus comme des mots d'un alphabet de quatre lettres. Là encore il est possible de gagner de la place mémoire en considérant que l'on a moins de liberté pour les dernières lettres.

4.7.4 Algorithme optimisé

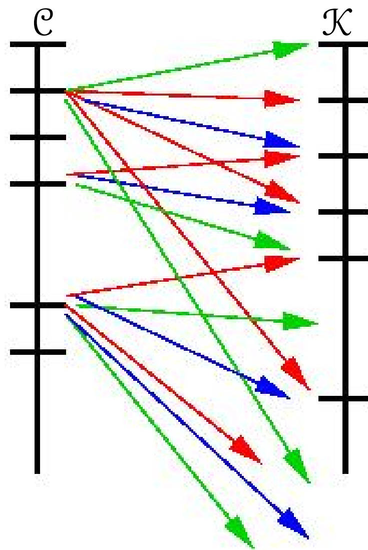
L'algorithme décrit ci-dessus et qui est utilisé, par exemple par Robertson et ses co-auteurs. C'est donc lui qui est (très minutieusement) décrit dans les écrits accompagnant leur article.

La vitesse d'exécution plus limitée de Coq[1] a conduit à chercher une optimisation supplémentaire. On remarque qu'à chaque fois qu'un nouveau chromogramme est ajouté à $\mathcal{K}_i$, il faut, pour chaque c' parcourir et tester tous les chromogrammes correspondants. On peut éviter ce parcours en remarquant qu'il est possible de *pré-calculer*, pour chaque coloriage c et chaque pivot p le nombre de chromogrammes w tels que $w \simeq_p c$. On va donc utiliser une représentation plus fine des ensembles $\mathcal{C}_i$: pour chaque c et chaque p, on va stocker le nombres de chromogrammes correspondants qui ne sont pas éléments de $\mathcal{K}_i$.

L'algorithme devient alors :

1. On calcule $\mathcal{C}_0$ et on initialise i à 0.
2. On parcourt C_i et pour chaque élément c, pour chaque pivot p, on ajoute tous les chromogrammes w tels que $w \simeq_p c$ à $\mathcal{K}_i$.
3. A chaque fois qu'un chromogrammes w est ajouté à $\mathcal{K}_i$, on parcourt les c' qui lui correspondent et on décrémente le compteur correspondant dans $\mathcal{C}_i$. Si jamais ce compteur tombe à 0, alors c' fait partie des nouveaux éléments de $\mathcal{C}_{i+1}$.
4. On parcourt les c' qui viennent d'être ajoutés pour passer de $\mathcal{C}_i$ à $\mathcal{C}_{i+1}$ et on revient à l'étape 2.

On peut décrire la combinatoire du problème initial en la décrivant par un graphe dont les sommets sont les coloriages et les chromogrammes, avec une arête entre w et c si $w \simeq_p c$:



[1]Cette optimisation a été proposée par Georges Gonthier a un moment où la compilation était seulement en cours d'intégration à Coq.

On peut remarquer qu'avec cette optimisation, l'algorithme parcourt exactement *une fois* chaque arête ; ce qui semble optimal.

Cette optimisation permet de traiter les calculs de réductibilité dans un cadre fonctionnel dans un temps presque comparable au programme C fournit par Robertson et ses co-auteurs.

4.8 Quelques remarques

Je passe ici sous silence bien d'autres aspects de la preuve formelle, qui sont décrits dans tous les détails par Georges Gonthier [52]. Les plus importants sont sans doute :

- Le traitement formel des graphes planaires orientés comme des *hypercartes*. Cette structure combinatoire, indépendemment découverte par des théoriciens de graphes comme Robert Cori, permet de définir formellement et de manière utilisable, les notions de faces, de chemins, d'extérieur et d'intérieur d'une configuration, de contraction et d'effacement d'arêtes, etc.
- Le rôle du langage de preuve. Georges Gonthier ayant développé au cours de ce développement un certain nombre de tactiques de preuves particulièrement adaptée à un certain style de formalisation. Ce que l'on peut remarquer à ce propos, et que ce style de preuve exploite lui aussi très largement la capacité de la théorie des types à raisonner *modulo le calcul*. Mais non pas à grande échelle comme dans la partie calculatoire décrite ci-dessus, mais à petite échelle. Les propositions se réécrivent petit à petit, en évitan à chaque fois le recours à des lemmes explicites.

Troisième partie

À propos du Calcul des Constructions Inductives

Chapitre 5

Consistences relatives entre Théorie des Types et Théorie des Ensembles

Ce chapitre reprend avant tout les deux codages entre théorie des types et théorie des ensembles décrits dans mon article de 1997 [106]. Il bénéficie toutefois de nouveaux travaux : l'article avec Alexandre Miquel [84] qui précise et corrige la preuve de correction du modèle ensembliste (dans le cadre du seul Calcul des Constructions), cette correction ayant été améliorée et adaptée au système complet dans [107]. Ce dernier article, décrivant une théorie des types avec preuves non-relevantes est, un peu sommairement, résumé dans la dernière partie de ce chapitre.

5.1 Sémantique ensembliste de Coq

Dans les grandes lignes, c'est un résultat connu depuis longtemps [29, 47, 66] : la possibilité d'interpréter les constructions de théories des types directement à travers leurs contreparties ensemblistes. C'est-à-dire une sémantique ensembliste simple pour (la théorie de) Coq. La simplicité de cette sémantique fait qu'on peut préférer la décrire comme une *traduction* ou un *codage* d'une théorie en l'autre.

Si de par sa simplicité, cette sémantique a longtemps été quelque peu négligée par les théoriciens de la démonstration, elle présente un certain nombre d'intérêts pour nous :

- En tant que preuve de cohérence d'abord : sa simplicité est garante de robustesse.
- De plus la simplicité de sa formulation fait qu'il est, en général, facile de justifier un axiome : il suffit de vérifier la validité de la traduction de cet axiome par la sémantique ; et justement, cette traduction est simple.
- Elle est naturellement *proof-irrelevant*. J'explique et développe ce dernier point plus loin.
- Enfin nous allons voir dans la section suivante que cette sémantique permet finalement une caractérisation finalement assez fine de l'expressivité logique de Coq. En effet, il est possible, à l'inverse, de coder la théorie des ensembles dans la théorie de Coq (éventuellement étendu par un ou deux axiomes) donnant ainsi lieu à des preuves de cohérence relatives croisées assez simples. On va voir que la contrepartie ensembliste des univers Type_i sont les cardinaux inaccessibles.

Nous allons également voir qu'il y a toutefois une difficulté longtemps ignorée dans la sémantique ensembliste qui n'est donc "pas si simple". Cette difficulté porte sur le traitement de l'imprédicativité de Prop et m'avait été signalée par Alexandre Miquel. Elle a été décrite pour la première fois dans un article commun [84].

5.1.1 Le résultat de Reynolds

Le principe de la sémantique ensembliste est très simple : un type est interprété par un ensemble. Plus exactement, comme il y a des variables dans les types, étant donnée une fonction $\mathcal{I}$ qui associe une inter-

prétation à chaque variable, on associe $|A|_{\mathcal{I}}$ à chaque type A. Surtout, si A et B sont des types, alors on interprète le type fonctionnel $A \rightarrow B$ par l'ensemble des fonctions ensemblistes :

$$|A \rightarrow B|_{\mathcal{I}} = |B|_{\mathcal{I}}^{|A|_{\mathcal{I}}}$$

ce qui est un choix on ne peut plus simple[1]. En faisant ce choix, on est immédiatement confronté au résultat de John Reynolds [92] *polymorphism is not set-theoretic* qui dit essentiellement qu'un tel modèle pour le système F est forcément trivial : chaque type à au plus un élément.

Dans le cas de Coq, cela veut dire qu'une sorte imprédicative doit être interprétée de manière booléenne : on aura donc $|\mathsf{Prop}|_{\mathcal{I}} = \{\emptyset; \mathbb{I}\}$ avec $\mathbb{I}$ bien choisi.

5.1.2 Cardinaux inaccessibles

Pour interpréter les univers de CCI, nous supposons l'existence de cardinaux inaccessibles. Cette notion de théorie des ensembles est bien connue (voir Krivine [73] ou Kunen [74] par exemple).

Définition 4.3 (Cardinal inaccessible) *Un cardinal infini λ est dit inaccessible si et seulement si :*
- *Pour tout cardinal $\alpha < \lambda$, $2^\alpha < \lambda$.*
- *Soit $(\beta_i)_{i \in I}$ une famille de cardinaux $< \lambda$ indexée par un cardinal $I < \lambda$; alors $\mathsf{sup}_{i \in i}(\beta_i) < \lambda$.*

L'idée principale derrière cette notion est qu'elle permet de construire un ensemble qui est lui-même un modèle de ZFC. Supposer l'existence d'un tel cardinal augmente donc fortement l'expressivité logique de la théorie. Les constructions suivantes sont bien connues.

On écrit $\mathcal{P}$ pour l'ensemble des parties *(powerset)*.

Définition 4.4 *Pour chaque ordinal α, on définit l'ensemble V_α par induction transfinie sur α :*
- $V_0 \equiv 0$
- $V_\alpha \equiv \bigcup_{\beta < \alpha} \mathcal{P}(V_\beta)$ *if* $\alpha > 0$.

Le résultat suivant est une conséquence bien connue de l'axiome de fondation.

Définition 4.5 (Rang d'un ensemble) *Pour tout ensemble X, il existe un plus petit ordinal α tel que $X \in V_\alpha$. α est le* rang *de X, noté $rk(X)$.*

Lemma 4.1 *Si λ est un cardinal inaccessible, alors V_λ vérifie les axiomes de ZFC. En particulier, si $A \in V_\lambda$ et pour tout $a \in A$, $B_a \in V_\lambda$, alors $\Pi_{a \in A} B_A \in V_\lambda$.*

Dans la suite on assume l'existence d'une suite croissante de cardinaux inaccessibles $(\lambda_i)_{i \in \mathbb{N}}$. Pour chaque i on notera $\mathcal{U}(\mathrm{i}) \equiv V_i$.

5.1.3 Caractérisation syntaxique des preuves

Même si l'interprétation ensembliste est simple, elle fait un différence entre les objets habitant le niveau imprédicatif et les autres. Dans le premier cas $A \rightarrow B$ est interprété de manière booléenne, dans l'autre c'est un ensemble de fonctions tel que défini ci-dessus.

Définition 4.6 *Un terme est* une preuve *s'il est de la forme :*
- x^*,
- *$(p\ t)$ où p est une preuve,*
- *$\lambda x : A.p$ où p est une preuve,*
- *une élimination d'un type inductif vers une proposition P : Prop.*

Je ne détaille pas syntaxiquement le dernier cas, dont la formulation précise dépend des détails de la syntaxe choisie pour les opérateurs d'élimination des types inductifs. Dans tous les cas, il est toutefois possible de savoir statiquement si une telle élimination se fait vers le niveau Prop ou un niveau Type_i.

[1]Voire simpliste si l'on s'intéresse à comprendre vraiment quelle sont les fonctions représentables dans la théorie des types. Jean-Yves Girard écrit par exemple qu'ainsi ces dernières sont *"lost in a sea of set-theoretic functions"*.

5.1.4 Notations

En théorie des ensembles, les fonctions sont définies de manière extentionnelle par leur *graphe*. D'habitude on code celui-ci de la manière suivante :

$$f = \{(x, y) | y = f(x)\}.$$

Ici, nous allons utiliser un codage un peu différent du, à Peter Aczel [5] :

$$f = \{(x, y) | y \in f(x)\}.$$

Cela se traduit par les notations suivantes. L'application (ensembliste) est définie comme $f(x) \equiv \{y|(x, y) \in f\}$. Aussi, la construction des fonctions ensemblistes est faite à travers l'opération d'abstraction suivante : $\alpha \in A \mapsto t$ est un raccourci pour $\{(\alpha, \beta)|\alpha \in A \wedge \beta \in t\}$. Les avantages techniques de cette approche sont discutés en détail chez Aczel et dans [84]. En bref, le point principal est qu'une fonction constante retournant toujours $\emptyset$ (sur n'importe quel domaine) est elle-même codée par $\emptyset$. Du coup, $\emptyset$ est le candidat parfait pour dénoter la "preuve canonique" de toute proposition $P : \mathsf{Prop}$.

On peut alors (re-)définir les produits dépendants ensemblistes : si A est un ensemble et $(B_a)_{a\in A}$ une famille d'ensembles indexés sur A, on note :

$$\begin{array}{rcl}\Pi_{a\in A} B_a & \equiv & \{f \in A \to \bigcup_{a\in A} B_a \mid \forall a \in A. f(a) \in B_a\} \\ \Sigma_{a\in A} B_a & \equiv & \{(a, b) \mid a \in A \wedge b \in B_a\}.\end{array}$$

5.1.5 Interprétation du fragment PTS

D'habitude, [6, 83] on définit d'abord l'interprétation comme une fonction partielle et on montre la totalité et la correction ensuite. Ici, de part le codage d'Aczel de l'application, on voit immédiatement que l'interprétation est toujours définie.

Définition 4.7 *Pour toute application $\mathcal{I}$ allant de l'ensemble des variables à $\bigcup_{i\in\mathbb{N}} \mathcal{U}_i$, nous définissons une application associant un ensemble $|t|_{\mathcal{I}}$ à tout terme t. Cette fonction est définie par récurrence sur la taille de t par les équations suivantes :*

$$\begin{array}{rcl}|t|_{\mathcal{I}} & \equiv & \emptyset \quad \textit{si } t \textit{ est une preuve syntaxique} \\ \textit{Dans les autres cas :} & & \\ |x_\diamond|_{\mathcal{I}} & \equiv & \mathcal{I}(x_\diamond) \\ |\lambda x : A.t|_{\mathcal{I}} & \equiv & \alpha \in |A|_{\mathcal{I}} \mapsto |t|_{\mathcal{I};x\leftarrow\alpha} \\ |(t\ u)|_{\mathcal{I}} & \equiv & |t|_{\mathcal{I}}(|u|_{\mathcal{I}}) \\ |\Sigma x : A.B|_{\mathcal{I}} & \equiv & \Sigma_{\alpha\in|A|_{\mathcal{I}}} |B|_{\mathcal{I};x\leftarrow\alpha} \\ |\Pi x : A.B|_{\mathcal{I}} & \equiv & \Pi_{\alpha\in|A|_{\mathcal{I}}} |B|_{\mathcal{I};x\leftarrow\alpha} \\ | < t, u >_{\Sigma x:A.B} |_{\mathcal{I}} & \equiv & (|t|_{\mathcal{I}}, |u|_{\mathcal{I}}) \\ |\pi_i(t)|_{\mathcal{I}} & \equiv & \alpha_i \quad \textit{si } |t|_{\mathcal{I}} \textit{ est une paire } (\alpha_1, \alpha_2) \\ |\pi_i(t)|_{\mathcal{I}} & \equiv & \emptyset \quad \textit{si } |t|_{\mathcal{I}} \textit{n'est pas une paire} \\ |\mathsf{Prop}|_{\mathcal{I}} & \equiv & \{\emptyset; \mathbb{I}\} \\ |\mathsf{Type}(i)|_{\mathcal{I}} & \equiv & \mathcal{U}(i) \\ |(\mathsf{Eq_rec}\ A\ P\ a\ b\ p\ e)|_{\mathcal{I}} & \equiv & |p|_{\mathcal{I}}\end{array}$$

5.1.6 Interprétation des types inductifs

L'interprétation de chaque type inductif est construite inductivement, cette fois dans un sens ensembliste. On trouve, par exemple, une description précise de ce genre de construction chez Dybjer [47]. Il n'y a toutefois rien de bien surprenant dans ces constructions, et s'il y a des difficultés elles sont d'ordre techniques. Le plus dur est finalement de donner une présentation génériques pour l'ensemble des types inductifs, qui rende néanmoins justice à la simplicité de l'idée.

Ici nous refusons cet obstacle en nous restreignant à un exemple ; considérons la définition des listes :

$$\begin{array}{rl}\text{Inductive } \mathsf{list} : \mathsf{Type}_i := & \mathsf{nil} : \mathsf{list} \\ | & \mathsf{cons} : A \to \mathsf{list} \to \mathsf{list}.\end{array}$$

où $A : \mathsf{Type}_i$. On code les constructeurs en utilisant les entiers naturels (de la théorie des ensembles) 0,1,2. . .. L'ensemble $|\mathsf{list}|$ est (s'il existe) le plus petit élément de $\mathcal{U}(\mathrm{i})$ vérifiant :

- $0 \in |\mathsf{list}|$
- si $a \in |A|$ et $l \in |\mathsf{list}|$, alors $(1, a, l) \in |\mathsf{list}|$.

Les deux clauses correspondent aux deux constructeurs ; l'interprétation de ces derniers est naturelle :

- $|\mathsf{nil}| \equiv 0$
- $|(\mathsf{cons}\ a\ l)| \equiv (1, |a|, |l|)$ ou, pour être précis, la fonction (curryfiée) qui à $a \in |A|$ et $l \in |\mathsf{list}|$ associe $(1, a, l)$.

Cette technique se généralise sans difficulté à des définitions avec plusieurs constructeurs, eux-mêmes d'arité arbitraire. Ce qui est crucial dans cette définition, c'est que les règles de CCI imposent à chaque constructeur donné un nombre fixé d'arguments.

La condition de stricte positivité assure que la définition ci-dessus correspond bien à un opérateur monotone sur les ensemble. Puisque les arguments du constructeur sont tous de type Type_i, le résultat de correction du modèle assurera que le résultat soit bien dans $\mathcal{U}(\mathrm{i})$ et donc il suffira bien de considérer le plus petit point-fixe de l'opérateur dans $\mathcal{U}(\mathrm{i})$.

L'ordre structurel des éléments du type inductif est réfléchi par l'ordre bien-fondé sur son interprétation. Cela permet d'interpréter naturellement les schémas d'élimination, ce que je ne détaille pas ici.

5.1.7 Correction du modèle

Indiquons d'abord pourquoi le fait de repérer syntaxiquement les preuves du niveau imprédicatif est utile. Ce point technique est, ou plutôt semble, nécessaire à la preuve de correction. En effet, sans ce marquage syntaxique, on peut construire le même modèle, ce modèle sera correct, mais on ne saura pas le prouver.

La difficulté se situe au niveau du traitement de la règle de conversion : il faut que deux types bien-formés et β-convertibles aient la même interprétation dans le modèle. Pour cela, on va utiliser le fait que l'interprétation est inchangée par β-réduction :

$$\text{si } T \rhd_\beta T', \text{ alors } |T|_{\mathcal{I}} = |T'|_{\mathcal{I}}.$$

Or cette propriété, longtemps considérée comme évidente, n'est en fait pas vraie pour certains termes mal-typés. Considérons en effet

$$\lambda x^* : P.x\ \mathsf{Prop}$$

ce terme est une preuve, donc son interprétation est $\emptyset$. Or il se réduit à Prop dont l'interprétation est $\{\emptyset; \mathbb{I}\}$. On pourrait avoir l'impression, avec cet exemple, que le problème est justement due à l'utilisation du marquage syntaxique des preuves dans la définition de l'interprétation. Dans [84] nous montrons qu'il n'en est rien ; plus précisément :

- Le problème vient de ce que le domaine de la fonction $\lambda x^* : P.x$ est "oublié" dans l'interprétation ; et un tel oubli est inévitable à partir du moment où certain termes sont interprétés de manière booléenne.

– De part le codage des fonctions "à la Aczel" utilisé ici, l'interprétation est toujours définie. Lorsque l'on utilise le codage conventionnel, le même type d'exemple se traduite par des termes $T \triangleright_\beta T'$ tel que $|T|_\mathfrak{I}$ est défini et $|T'|_\mathfrak{I}$ ne l'est pas.

Tous ces contre-exemples portent sur des termes mal-typés. Mais si l'on essaye de montrer la stabilité de l'interprétation par rapport à la réduction uniquement pour les termes bien-typés, on a besoin de la correction de l'interprétation. On se retrouve donc dans une boucle sans fin (ou plutôt sans début) entre la preuve de correction et celle de stabilité par rapport à la réduction.

Ce problème semble de nature technique et à première vue, il est sans doute naturelle de penser qu'il possède une solution. Après avoir passé un certain temps à chercher cette solution, je partage finalement l'opinion d'Alexandre Miquel et pense que ce problème est au final plus profond qu'il n'y paraît.

Pourquoi le marquage

L'intérêt principal du marquage apparaît alors. Le problème dans tous ces contre-exemples a lieu lorsqu'une variable interprétée de manière irrelevante est substituée par un terme qui ne l'est pas. En ayant identifié syntaxiquement les termes et les variables irrelevants, nous pouvons prohiber de telles réductions dans la preuve de correction, puis montrer que de telles réduction n'ont jamais lieu dans des termes bien-typés.

On donne l'enchaînement des lemmes principaux.

Définition 4.8 *Un terme t est* bien sorté *si pour tout t' tel que $t \triangleright_\beta^* t'$, tous les radicaux de t' sont de l'une des formes suivantes :*
- *$\lambda x^* : A.v\ u$ où u est une preuve syntaxique,*
- *$\lambda x^\diamond : A.v\ u$ où u n'est pas une preuve syntaxique.*

Lemma 4.2 *Tout terme bien typé est bien sorté. Plus précisément : si $\Gamma \vdash t : T$, alors $\Gamma \vdash T : \mathsf{Prop}$ si et seulement si t est une preuve syntaxique.*

Lemma 4.3 *Pour tout terme t bien sorté, si $t \triangleright_\beta t'$, alors $|t|_\mathfrak{I} = |t|_\mathfrak{I}$.*

Théorème 5 (Correction) *Si $\Gamma \vdash t : T$ et $\mathfrak{I} \in |\Gamma|$, alors $|t|_\mathfrak{I} \in |T|_\mathfrak{I}$.*

On en déduit immédiatement la cohérence du système : il n'y a pas de dérivation de $[] \vdash \Pi X : \mathsf{Prop}.X$.

5.1.8 Axiomes justifiés par le modèle

On l'a déjà évoqué, une des forces de ce modèles, c'est sa simplicité. En particulier :
- Il valide de nombreux axiomes qui peuvent être utiles dans certains cas,
- ces validations sont simples à vérifier, de part la simplicité de la traduction vers les ensembles.

5.1.9 Le tiers-exclu

Le tiers-exclu sur Prop est évidemment validé de part l'interprétation booléenne de Prop :

$$EM \equiv \Pi P : \mathsf{Prop}.P \vee \neg P$$

où $A \vee B$ est une abréviation de $\Pi X : \mathsf{Prop}.(A \rightarrow X) \rightarrow (B \rightarrow X) \rightarrow X$ et $\neg A$ une abréviation de $A \rightarrow \Pi x : \mathsf{Prop}.X$.

Ce que l'on peut également remarquer à ce propos, c'est que cette version du tiers-exclu vivant dans Prop, elle ne met pas en cause la constructivité des résultats établis dans Type_i.

Ce n'est pas le cas des versions plus fortes du tiers-exclu qui sont toutefois également validées par le modèle :

$$\forall A : \mathsf{Type}_i.A + \neg A$$

où $A + B$ est une définition de la disjonction (ou du type somme) dans Type_i :

```
Inductive sum (A : Type) (B : Type) : Type :=
    inl : A -> A + B
  | inr : B -> A + B.
```

Le fait que CCI permette de distinguer entre ce tiers-exclu "violent", que l'on ne pourra réaliser dans le calcul fonctionnel, et la première version beaucoup plus inoffensive est, il me semble, un point positif pour cette théorie des types.

5.1.10 Axiomes du choix

Choix intentionnel

Il y a des versions de l'axiome du choix qui sont de toute façon valides en théorie des types : lorsqu'elles portent sur des formulations du quantificateur existentiel pour lesquelles on dispose des deux projections. C'est exactement le cas pour le σ-type. On a bien :

$$\Pi AB : \mathsf{Type}_i.\Pi R : A \to B \to \mathsf{Prop}.(\Pi a : A.\Sigma b : A.(R\ a\ b)) \to \Sigma f : A \to B.\Pi a : A.(R\ a\ (f\ a)).$$

En revanche, la proposition correspondante utilisant la quantificateur $\exists$ (qui habite le niveau Prop) n'est pas prouvable :

$$\mathrm{ACIT} \equiv \Pi AB : \mathsf{Type}_i.\Pi R : A \to B \to \mathsf{Prop}.(\Pi a : A.\exists b : A.(R\ a\ b)) \to \exists f : A \to B.\Pi a : A.(R\ a\ (f\ a)).$$

Nous l'appelons ACIT (pour Axiome du Choix Intentionnel en Théorie de Types). Cette proposition est validée dans le modèle, pour peu que l'on accepte l'axiome du choix ensembliste. Il faut ici faire deux remarques :

- Là encore, il s'agit d'une proposition non-calculatoire (de type Prop) ; l'admettre ne met donc pas en danger les résultats de constructivité mentionnés ci-dessus.
- On verra dans la partie suivante du chapitre que cet axiome est en fait très puissant et donne à CCI l'expressivité de ZF. D'une certaine façon, admettre cet axiome oblige à avoir un modèle très riche : toute fonction prouvablement totale dans CCI (et l'on dispose de l'imprédicativité pour faire des preuves dans CCI) doit exister dans le modèle.

Choix extentionnel

Si l'axiome ACIT est justifié par l'axiome du choix ensembliste, on peut arguer qu'il ne s'agit pas vraiment d'un "choix" comme pour l'axiome ensembliste. En effet, si la preuve de $\Pi a : A.\exists b : A.(R\ a\ b)$ est constructive (et elle sera constructive si l'on n'admet pas d'axiomes par ailleurs), alors le "choix" du témoin est en fait effectué dans la preuve. De fait, on verra plus bas que cet axiome correspond plutôt au schéma de remplacement.

C'est pourquoi Per Martin-Löf a récemment choisi l'appellation "axiome du choix intentionnel" pour cet axiome. Par opposition, l'axiome du choix extentionnel permet vraiment de faire "un choix" de manière totalement non-constructive.

La formulation d'un tel axiome est un peu plus compliquée que celle de ACIT. En voici une possible :

$$\begin{aligned}\mathrm{ACET} \equiv \Pi A : \mathsf{Type}_i.\Pi R : A \to A \to \mathsf{Prop}.(\mathrm{equiv}\ R) \to \\ \exists f : A \to A.\Pi a : A.R\ a\ (f\ a) \wedge \Pi a\ b : A.R\ a\ b \to f\ a = f\ b\end{aligned}$$

où $(\mathrm{equiv}\ R)$ est une abréviation pour dire que R est une relation d'équivalence.

Là encore, cet axiome est évidemment validé dans le modèle par l'axiome du choix ensembliste. Il est toutefois très différent de ACIT. D'une part, on ne peut pas déduire ACIT et ACET, et vice-versa. Mais

à partir de ACET, on peut déduire le tiers-exclu, par une construction inspirée de celle de Diaconesu [36], elle-même transposée en théorie des ensembles par Goodman et Myhill [67].

Notre construction est toutefois un peu différente ; elle peut être immédiatement transposée en théorie des ensembles, et utilise une autre formulation de AC que la preuve connue. Elle est assez courte pour être facilement décrite ici.

Soit P : Prop la proposition dont on va "décider" de la validité. On définit sur le type des booléens la relation d'équivalence suivante :

$$R\ x\ y\ \equiv\ x = y \vee P$$

ACET garantit alors l'existence d'une fonction $f : \mathsf{bool} \to \mathsf{bool}$ qui choisit un élément canonique dans chaque classe d'équivalence. Il suffit alors d'éliminer le quantificateur existentiel, puis de comparer f true et f false. S'il sont égaux, alors P est vraie, sinon P est fausse. A noter, bien sûr, que l'élimination du quantificateur $\exists f. \ldots$ n'est possible que si l'on prouve une proposition habitant Prop. On ne peut pas déduire de version forte (dans Type_i) du tiers-exclu.

5.2 Coder les ensembles comme des types

La construction décrite maintenant est à mon sens l'une des plus élégantes qui soient en théorie des types. L'idée est essentiellement due à Peter Aczel qui l'a proposée et développée entre 1977 et 1985. A l'origine on retrouve la volonté de *fonder* les mathématiques sur la théorie des types et donc de montrer que la théorie des ensembles peut être retrouvée comme une construction particulière dans les théories de Martin-Löf. Comme il s'agissait essentiellement de fonder des mathématiques constructives et prédicatives, Peter Aczel n'utilisait pas l'équivalent de la sorte Prop ; en conséquence la théorie qu'il interprète est une variante constructive et prédicative de la théorie des ensembles, qu'il appelle CZF, et où la quantification est essentiellement bornée : on ne peut pas toujours écrire $\forall x.P$ mais en général seulement $\forall x \in A.P$.

Ici nous laissons de coté CZF et exploitons l'imprédicativité de Prop pour coder les théorie habituelles des ensembles : Z, ZF et ZFC. Ce travail avait été décrit une première fois dans l'article de 1997, *Sets in Types, Types in Sets* [106].

5.2.1 Les ensembles

L'idée première d'Aczel est que les ensembles peuvent être construits inductivement en suivant l'axiome de fondation : les éléments sont structurellement plus petits que l'ensemble auquel ils appartiennent. La définition inductive, paramétrée ici par un indice d'univers i est :

$$\begin{array}{l}\text{Inductive } \mathsf{Set}\ :\ \mathsf{Type}_{i+1}\ := \\ \quad \mathsf{sup} : \Pi A : \mathsf{Type}_i.(A \to \mathsf{Set}) \to \mathsf{Set}.\end{array}$$

Intuitivement, $(\mathsf{sup}\ A\ f)$ est l'ensemble des $(f\ a)$ où a parcourt le type A ; en mêlant notation ensembliste et types, on pourrait le noter $\{f(a), a : A\}$. Remarquons que $(sup\ A\ f)$ contient au plus autant d'éléments que le type A (et moins si, par exemple, f est une fonction constante).

Un bon premier exemple est la construction de la paire ensembliste qui correspond à l'axiome du même nom de la théorie des ensembles. Puisque $\{E, E'\}$ a au plus deux éléments, il est naturel et de choisir les booléens comme type de base :

$$\begin{array}{l}\text{Definition } \mathsf{Pair}\ :\ \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}\ := \\ \quad \mathsf{fun}\ E_1\ E_2 \mapsto (\mathsf{sup\ bool}\ (\mathsf{fun} \quad \mathsf{true} \quad \mapsto E_1 \\ \qquad\qquad\qquad\qquad\qquad\quad \mid \quad \mathsf{false} \quad \mapsto E_2)).\end{array}$$

Un autre exemple est l'ensemble vide, qui est construit à partir du type vide bot[2] :

$$\text{Definition } \mathsf{Empty} := (\mathsf{sup}\ \mathsf{bot}\ \mathsf{fun} : \mathsf{bot} \to \mathsf{Set}).$$

5.2.2 Les propositions

Avant de poursuivre cette série de définitions et de prouver qu'elles vérifient bien les axiomes ensemblistes, nous devons décider comment traduire les propositions de la théorie des ensembles. Il s'agit d'une théorie en logique du premier ordre avec deux symboles de prédicats (binaires) : l'appartenance et l'égalité. On définit d'abord l'égalité, comme une fonction structurellement récursive de Set vers Prop. Cette définition est bien sûre choisie pour capturer l'axiome d'extentionalité :

$$\begin{aligned}
&\text{Definition } \mathsf{Eq} \ : \ \mathsf{Set} \to \mathsf{Set} \to \mathsf{Prop} \ := \\
&\quad \mathsf{fun}\ (\mathsf{sup}\ A\ f)\ (\mathsf{sup}\ B\ g) \mapsto \quad ((\Pi a : A.\exists b : B.(\mathsf{Eq}\ (f\ a)\ (g\ b))) \\
&\qquad\qquad\qquad\qquad\qquad \wedge (\Pi b : B.\exists a : A.(\mathsf{Eq}\ (f\ a)\ (g\ b))))
\end{aligned}$$

À partir de là, on définit facilement l'appartenance :

$$\begin{aligned}
&\text{Definition } \mathsf{In} \ : \ \mathsf{Set} \to \mathsf{Set} \to \mathsf{Prop} \ := \\
&\quad \mathsf{fun}\ E\ (\mathsf{sup}\ A\ f) \mapsto \exists a : A.(\mathsf{Eq}\ E\ (f\ a)).
\end{aligned}$$

C'est dans ces deux dernières définitions que nous différons du choix originel d'Aczel : nous représentons les propositions de la théorie des ensembles comme des objets de Prop alors que lui les place, suivant les cas, dans Type_i ou Type_{i+1}.

Dans tous les cas toutefois, avec ces définitions, nous pouvons vérifier que la construction de la paire (non-ordonnée) est effectivement un témoin de l'axiome correspondant de la théorie de Zermelo. On démontre pour cela les trois lemmes suivants :

$$\begin{aligned}
&(A, B : \mathsf{Set})(\mathsf{In}\ A\ (\mathsf{Pair}\ A\ B)) \\
&(A, B : \mathsf{Set})(\mathsf{In}\ B\ (\mathsf{Pair}\ A\ B)) \\
&(A, B, C : \mathsf{Set})(\mathsf{In}\ C\ (\mathsf{Pair}\ A\ B) \to (\mathsf{Eq}\ A\ C) \vee (\mathsf{Eq}\ B\ C))
\end{aligned}$$

Il est important de comprendre que l'égalité ensembliste n'est pas l'égalité de Leibniz, qui est l'égalité propositionnelle de Coq. Il faut donc à chaque fois montrer que nos constructions sont extentionnelles. Par exemple :

$$\Pi A, A', B : \mathsf{Set}.(\mathsf{In}\ A\ B) \to (\mathsf{Eq}\ A\ A') \to (\mathsf{In}\ A'\ B).$$

5.2.3 Comparaison avec l'approche d'Aczel

D'un point de vue constructif, le défaut de notre approche est que nous ne pouvons par extraire le témoin d'une preuve existentielle[3], c'est-à-dire que prouver $\exists X : \mathsf{Set}.(P\ X)$ et effectivement exhiber un objet E de type Set ainsi qu'une preuve de $(P\ E)$ sont deux choses différentes. Un effet de bord est la difficulté a prouver le schéma de remplacement[4], ainsi que décrit dans la partie 5.2.5.

L'avantage est que nous gagnons la quantification non-bornée (sur tous les ensembles) grâce à l'imprédicativité. Nous évitons aussi la distinction un peu lourde entre les formules bornées ou non, qui mènent à avoir plusieurs formulations de l'axiome de compréhension.

[2] C'est-à-dire le type inductif qui n'a pas de constructeur.

[3] Plus exactement, si la preuve est sans axiomes, on peut la normaliser et retrouver le témoin, mais c'est une opération *externe* qui ne peut être effectuée dans la théorie des types.

[4] Il faut toutefois noter que, comme les autres constructions sont explicites (ou skolémisées) et que l'on dispose d'un calcul fonctionnel, on a finalement moins souvent besoin du remplacement que dans les versions traditionnelles de Z / ZF.

5.2.4 Les autres constructions explicites

Les autres constructions sous-jacentes à la théorie des ensembles Z (union, compréhension, infini et ensemble des parties) peuvent toutes être définies sans grande difficulté. Voici un choix possible pour ces définitions :

$$\begin{array}{l}
\text{Definition } \mathsf{Power} \;:\; \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set} \;:= \\
\quad \mathsf{fun}\ E \mapsto (\mathsf{sup} \quad (\mathsf{Set} \to \mathsf{Prop}) \\
\qquad\qquad\qquad\quad \lambda P : \mathsf{Set} \to \mathsf{Prop}.(\mathsf{Compr}\ P\ E)). \\
\text{Definition } \mathsf{Union} \;:\; \mathsf{Set} \to \mathsf{Set} \;:= \\
\quad \mathsf{fun}\ (\mathsf{sup}\ A\ f) \mapsto (\mathsf{sup} \quad \Sigma a : A.(\pi_1\ (f\ a)) \\
\qquad\qquad\qquad\qquad\qquad \mathsf{fun}(\sigma\ a\ b) \mapsto (\pi_2\ (f\ a)\ b)). \\
\text{Definition } \mathsf{Comp} \;:\; \mathsf{Set} \to (\mathsf{Set} \to \mathsf{Prop}) \to \mathsf{Set} \;:= \\
\quad \mathsf{fun}\ (\mathsf{sup}\ A\ f) \mapsto (\mathsf{sup} \quad \Sigma a : A.(P\ (f\ a)) \\
\qquad\qquad\qquad\qquad\qquad \mathsf{fun}(\sigma\ a\ p) \mapsto (f\ a)).
\end{array}$$

À partir de là, on peut, par exemple, définir l'intersection de la manière usuelle :

$$\begin{array}{l}
\text{Definition } \mathsf{Inter} \;:\; \mathsf{Set} \to \mathsf{Set} \;:= \\
\quad \mathsf{fun}\ E \mapsto (\mathsf{Comp}\ (\mathsf{Union}\ E)\ \lambda e : \mathsf{Set}.\Pi a : \mathsf{Set}.(\mathsf{In}\ a\ E) \to (\mathsf{In}\ e\ a)).
\end{array}$$

Bien sûr, il faut à chaque fois vérifier les propriétés attendues pour toutes ces constructions, ainsi que leur extentionalité.

Un jolie construction est l'ensemble des entiers naturels, qui correspond à l'axiome de l'infini, et qui est obtenu en utilisant le type des entiers naturels :

$$\begin{array}{l}
\text{Definition } \mathsf{enc} \;:\; \mathsf{nat} \to \mathsf{Set} \;:= \\
\qquad \mathsf{O} \quad \mapsto \mathsf{Empty} \\
\quad (\mathsf{S}\ n) \quad \mapsto (\mathsf{Union}\ (\mathsf{Pair}\ (\mathsf{enc}\ n)\ (\mathsf{Power}\ (\mathsf{enc}\ n)))). \\
\text{Definition } \mathsf{NAT} := (\mathsf{sup}\ \mathsf{nat}\ \mathsf{enc}).
\end{array}$$

Ces dernières définitions sont essentiellement celles d'Aczel, à l'exception de l'ensemble des parties qui utilise fortement l'imprédicativité. Une première conséquence est :

Théorème 6 *La théorie Z peut être encodée en* CCI_2*+EM.*

Où l'on désigne par CCI_i le Calcul des Constructions Inductives restreint aux i premiers univers $\mathsf{Type}(j)$.

5.2.5 Constructions non-calculatoires : remplacement et choix

La situation est plus compliquée pour le schéma de remplacement et l'axiome du choix (ensembliste). Ces deux axiomes reposent tous les deux sur des hypothèses de la forme $\forall \mathcal{A}.\exists \mathcal{B}.\ldots$. Or puisque nous travaillons avec un quantificateur existentiel non calculatoire, nous n'avons aucune chance de construire effectivement l'ensemble attendu au final. Nous pouvons toutefois *prouver* le schéma de remplacement ensembliste en utilisant ACIT. La preuve est sans grande malice même si plus longue que les précédentes.

Formulations du schéma de remplacement

Grâce à l'imprédicativité, les schémas d'axiomes comme le remplacement deviennent des propositions. La version suivante du schéma de remplacement (qui semble être souvent appelée schéma de collection dans la littérature anglo-saxonne) peut être prouvée en Coq sous l'hypothèse ACIT. Elle est paramétrée par un prédicat binaire P :

$$(\forall X\,.\,\exists Y\,.\,P(X,Y)) \Rightarrow \forall E\,.\,\exists A\,.\,(\forall x \in E\,.\,\exists y \in A\,.\,P(x,y)).$$

Supposons de plus que P est une relation fonctionnelle, c'est-à-dire :

$$\forall x, y, y'\,.\,P(x,y) \wedge P(x,y') \Rightarrow y = y'.$$

Alors en supposant ACIT et le tiers-exclu, on peut prouver le schéma de remplacement habituel :

$$\forall X\,.\,\exists Y\,.\,\forall y\,.\,(y \in Y \iff \exists x \in X\,.\,P(x,y)).$$

On peut donc affirmer :

Théorème 7 *La théorie ZF peut être codée dans CCI_2+EM+$ACIT_3$.*

Un codage possible de l'axiome du choix ensembliste

Considérons la version suivante de l'axiome du choix ensembliste :

Soit E un ensemble tel que :
– les éléments de E sont non-vides,
– l'intersection de deux éléments de E est toujours vide.
Alors il existe un ensemble X tel que l'intersection de X avec tout élément de E est un singleton.

Lorsque l'on regarde cette proposition à travers le codage, on peut considérer que E est de la forme $(\mathsf{sup}\ A\ f)$. En utilisant ACIT, on peut prouver l'existence d'une fonction g de type $A \to \mathsf{Set}$, telle que pour tout a de type A, on a $(\mathsf{In}\ (g\ a)\ (f\ a))$. Supposons en revanche que a et b soient deux objets distincts de type A tels que $(\mathsf{Eq}\ (f\ a)\ (f\ b))$; nous ne pouvons conclure que $(\mathsf{Eq}\ (g\ a)\ (g\ b))$. L'ensemble $(\mathsf{sup}\ A\ g)$ n'est donc pas un témoin adéquat pour l'axiome du choix : son intersection avec l'élément $(f\ a)$ de E peut avoir plus d'un élément (ici $(g\ a)$ and $(g\ b)$). Il nous faut supposer de plus ACET pour prouver le lemme attendu. Une conclusion est que :

Théorème 8 *La théorie ZFC peut être encodée dans CCI_3 sous les hypothèses $ACIT_3ACET_3$.*

Le fait que l'axiome du choix ACIT est trop intentionnel (ou pas assez extentionnel) pour permettre de montrer le choix ensembliste à depuis été remarqué indépendamment par Per Martin-Löf ; il discute la question dans [82].

5.2.6 Cardinaux inaccessibles

Jusqu'ici nous n'avons utilisé que deux univers Type_i et Type_{i+1}.En conséquence, CCI_2 (avec les axiomes) est suffisant pour encoder ZFC. On peut maintenant utiliser des codages explicites pour des cardinaux inaccessibles en utilisant d'avantage d'univers.

L'idée est, encore, remarquablement simple ; elle utilise la seule règle de typage que l'on n'a pas encore utilisée ici, à savoir la cumulativité. On duplique tout le codage défini jusqu'ici, mais dans l'univers supérieur. Pour simplifier les notations, supposons que $i < 3$ et redéfinissons les ensembles au niveau inférieur :

$$\text{Inductive } \mathsf{Set}' \ :\ \mathsf{Type}_i \ := $$

$$\mathsf{sup}' : \Pi A : \mathsf{Type}_{i-1}.(A \to \mathsf{Set}') \to \mathsf{Set}'.$$

Le nouveau type Set' correspond alors à $\mathcal{U}(1)$, l'ensemble des "petits ensembles" qui est manifestement clos pour tous les axiomes ensemblistes (c'est-à-dire est un modèle de la théorie des ensembles). De fait, on a une injection immédiate de Set' vers Set :

$$\text{Definition } inj \ : \ \mathsf{Set}' \rightarrow \mathsf{Set} \ := \\ \mathsf{fun}\ (\mathsf{sup}'\ A\ f) \mapsto (\mathsf{sup}\ A\ \lambda a : A.(inj\ (f\ a))).$$

Et nous pouvons bien définir le grands ensemble des petits ensembles :

$$\mathsf{Big} \ := \ (\mathsf{sup}\ \mathsf{Set}'\ inj).$$

Il est, par exemple, étonnamment simple de prouver que Big est clos pour l'axiome des parties :

$$\Pi E : \mathsf{Set}.(\mathsf{In}\ E\ \mathsf{Big}) \rightarrow (\mathsf{In}\ (\mathsf{Power}\ E)\ \mathsf{Big}).$$

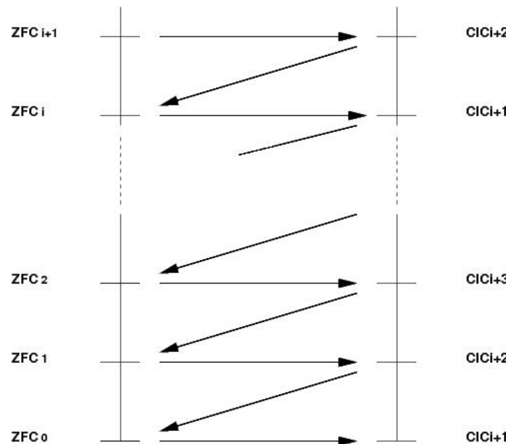
Bien sûr, on peut recommencer cette opération sur plusieurs niveaux et construire en utilisant plusieurs univers.

Même si nous n'avons pas prouvé formellement en Coq que l'existence de Big impliquait l'existence d'un cardinal inaccessible[5], cela semble clair. En admettant donc qu'il n'y a pas de difficulté ici, on peut énoncer le résultat de cohérence relative suivant :

Théorème 9 *La théorie ZFC avec n cardinaux inaccessibles peut être encodée en* $CCI_{n+2} + ACIT_{n+2} + ACET_{n+2}$.

5.3 Conclusion

On donc, à travers des constructions relativement simples, une famille de preuves de cohérence relative entre ZFC d'une part, et CCI muni des deux axiomes du choix d'autre part. On peut résumer cette situation à travers le petit schéma suivant :



Il reste quelques questions ouvertes ; par exemple :

- Peut-on obtenir un schéma similaire avec la théorie des ensembles ZF. Il faudrait pour cela justifier le schéma de remplacement par un axiome plus faible que ACIT donc le codage ensembliste donne directement l'axiome du choix ensembliste.
- Peut-on caractériser ainsi l'expressivité de CCI seul.

[5]La raison principale est que cela demanderait de développer en Coq toute la théorie des ordinaux ensemblistes, ce qui semble assez lourd et peu intéressant. J'ai en revanche formalisé la définition imprédicative mais constructive des ordinaux due à Paul Taylor [99].

Plus généralement, ces constructions peuvent apparaître simples comparées à des sémantiques plus sophistiquées destinées à établir des résultats fins sur tel ou tel système logique ou calculatoire. Mais s'il s'agit de justifier la cohérence d'un système de preuves destiné à être diffusé et utilisé dans des cadres variés, cette simplicité est justement rassurante. La justification qu'il apporte de certains axiomes additionnels comme ceux présentés ici répond à des attentes concrètes d'utilisateurs.

D'une certaine façon, on peut alors voir la théorie des types comme simplement "une bonne implémentation" de la théorie des ensembles : les jugements prouvés peuvent être facilement lus et compris comme leur contrepartie ensemblistes. En revanche, on garde dans le langage logique un vrai langage de programmation, ce qui peut être un avantage décisif pour certaines applications. C'est ce que j'essaye de montrer dans la partie précédente du mémoire.

5.4 Inclure la *proof-irrelevance* dans le formalisme

A l'origine, la distinction entre Prop et les autres sortes avait été introduite pour permettre *l'extraction de programmes* : effacer certaines parties d'une preuve constructive pour ne garder qu'un programme fonctionnel. On choisit alors de placer au niveau Prop exactement ce qui doit être effacé. Pour le fragment PTS on peut facilement définir la fonction d'extraction grâce au marquage que nous avons utilisé dans ce chapitre : on introduit une nouvelle constante (non-typée) ϵ et on définit une forme de réduction qui va précisément effacer les preuves du niveau Prop ; on note $\rhd_\varepsilon$ pour la clôture par congruence des équations suivantes :

$$\begin{array}{rcl} x_* & \rhd_\varepsilon & \epsilon \\ (\epsilon\ t) & \rhd_\varepsilon & \epsilon \\ \lambda x : A.\epsilon & \rhd_\varepsilon & \epsilon \end{array}$$

On voit que cette "réduction" remplace tout objet de sorte Prop par un objet canonique ϵ. Si la formulation de la définition est un peu différente, il s'agit là d'une fonction d'extraction habituelle (voir [90, 23, 78]). On peut évidemment remarquer que tout modèle de réalisabilité qui validera une telle fonction d'extraction sera, a priori, proof-irrelevant.

Inversement, le modèle ensembliste présenté ici valide évidemment la fonction d'extraction. On peut alors aller plus loin et inclure la non-relevance des preuves dans le formalisme. Il suffit pour cela de définir $=_{\beta\varepsilon}$ comme la clôture réflexive, symétrique, et transitive de l'union de $\rhd_\beta$ et $\rhd_\varepsilon$, et de redéfinir la relation de sous-typage en conséquence. On a alors libéralisé la règle de conversion :

$$(\textsc{Conv})\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s}{\Gamma \vdash t : B}\ (\text{si } A \leq B)$$

puisque celle-ci identifie alors toutes les preuves d'une même proposition $P : \mathsf{Prop}$.

Dans l'article [107] j'étudie plus en détail cette version de la théorie des types. A travers quelques applications possibles, j'essaye d'argumenter qu'il s'agit là d'une modification utile pour l'utilisation dans un système de preuves comme Coq. Je montre en particulier de la non-relevance des preuves rend cette théorie plus extentionnelle. On peut aussi voir que cette théorie permet de retrouver, pour l'essentiel, le comportement du formalisme implémenté dans le système de preuves PVS [94]. Autrement dit, on peut voir ce formalisme comme une extension du formalisme de PVS, muni de termes de preuves explicites. Cet aspect à été utilisé par Matthieu Sozeau [96] qui travaille maintenant sur l'intégration de cette caractéristique à Coq.

Notons enfin que ce relâchement de la règle de conversion est strictement plus fort que le simple axiome de non-relevance des preuves :

$$\Pi P : \mathsf{Prop}.\Pi p_1 : P.\Pi p_2 : P.p_1 = p_2.$$

Aussi, tous les paradoxes du chapitre suivant qui utilisent cet axiome peuvent également être construits dans ce système *proof-irrelevant.*

Chapitre 6

Difficultés avec Set imprédicatif

Le système que j'ai étudié dans ma thèse était CCI avec imprédicativité pour la sorte Set et seulement un univers. Le sytème implémenté en Coq était Set imprédicatif avec toute la hiérarchie d'univers. Rappellons que l'imprédicativité de Set signifie que l'on a la régle

$$\frac{\Gamma \vdash A : s \qquad \Gamma(x : A) \vdash B : \mathsf{Set}}{\Gamma \vdash \Pi x : A.B : \mathsf{Set}}$$

La différence essentielle entre Set et Prop étant que dans Set on dispose de types inductifs avec élimination forte (voir par exemple 2.2.3). C'est-à-dire que pour `bool : Set`, il est possible de montrer `true`$\neq$`false`. En revanche, il n'est jamais possible de montrer, sans axiome supplémentaire, que deux preuves d'une proposition `P : Prop` sont distincts.

On peut donc prouver que la proof-irrelevance est *fausse* dans Set.

Dans tout ce chapitre on considère que l'on se trouve dans ce système, avec Set *imprédicatif et muni de l'élimination forte.*

Si l'on peut toujours penser que le système avec Set imprédicatif et toute la hiérarchie d'univers est fortement normalisable et non cohérent, les seules preuves connues sont si complexes dans le détails qu'il est difficile de leur faire vraiment confiance. Surtout on ne semble pas connaitre de modèle raisonnable de ce système. Un tel modèle permettrait d'ailleurs d'espérer une preuve de normalisation plus compréhensible en utilisant des $\|ambda$-sets (voir Altenkirch [6]).

Ce que je veux montrer dans ce chapitre, c'est que ce système est, au mieux, extrêmement délicat, et que nombre d'axiomes relativement intuitifs y sont faux. Une étape importante en ce sens avait été accomplie par Laurent Chicli, Loïc Pottier et Carlos Simpson lorsqu'ils ont montré que la conjonction du tiers-exclu (sur Prop) et de ACIT (l'axiome du choix intentionnel décrit dans le chapitre précédent section 5.1.10) était incohérente.

6.1 Le résultat de Chicli, Pottier et Simpson

Le paradoxe en question repose d'une part sur la remarque précédente : on peut exhiber un $B : \mathsf{Set}$ tel que deux éléments de B sont prouvablement différents. Par ailleurs, Stefano Berardi avait montré que le tiers-exclu dans une sorte imprédicative suffisait à *démontrer* la proof-irrelevance. Tout se passant comme si le tiers-exclu suffisait pour forcer tout modèle à être ensembliste, et donc à se retrouver dans la situation décrite par Reynolds.

Une conséquence immédiate est donc que le tiers-exclu dans Set est prouvablement faux.

On aurait pu croire qu'avoir restreint le tiers-exclu à Prop suffirait à éviter le paradoxe. Ce que Chicli, Pottier et Simpson on remarqué, c'est qu'il n'en était rien, au moins si l'on admettait ACIT (pour Set) :

$$\text{ACIT} \equiv \Pi AB : \mathsf{Set}.\Pi R : A \to B \to \mathsf{Prop}.(\Pi a : A.\exists b : A.(R\ a\ b)) \to \exists f : A \to B.\Pi a : A.(R\ a\ (f\ a)).$$

En effet, cet axiome lie ce qui se passe dans les niveaux Prop et Set. D'abord, on peut "projeter" tout type (donc tout Set) dans Prop :

$$\{A\} \equiv \Pi X : \mathsf{Prop}.(A \to X) \to X.$$

La proposition $\{A\}$ signifie donc quelque chose comme "il existe une preuve de A mais je ne la connais pas" ; on a immédiatement $A \to \{A\}$.

Avec le tiers-exclu dans Prop on montre immédiatement :

$$\forall A : \mathsf{Set}.\{A\} \vee \neg\{A\}$$

et donc aussi :

$$\forall A : \mathsf{Set}.\exists b : \mathsf{bool}.(b = \mathsf{true} \wedge \{A\}) \vee (b = \mathsf{false} \wedge \neg\{A\})$$

et avec ACIT on peut en déduire l'existence d'une fonction de décision pour tout $A : Set$:

$$\exists f : \mathsf{Set} \to \mathsf{bool}.\Pi A : \mathsf{Set}.(f\ A = \mathsf{true} \wedge \{A\}) \vee (f\ A = \mathsf{false} \wedge \neg\{A\}).$$

On peut alors conclure le paradoxe. En effet, il s'agit de démonter une proposition de type Prop (soit la proof-irrelevance dans Set, soit directement $\Pi X : \mathsf{Prop}.X$). On peut pour cela éliminer le quantificateur existentiel ; et une fois que l'on dispose de la fonction de décision f il est aisé de reprendre la construction du paradoxe de Berardi.

6.2 Une variante basée sur l'opérateur J de Girard

6.2.1 Rappel : dans le Système F

Le système F est *paramétrique*, c'est-à-dire que les calculs ne dépendent jamais des types. Jean-Yves Girard avait fait la remarque que casser cette paramétricité en autorisant les tests dynamiques sur les types cassait aussi la propriété de normalisation.

Voici cette construction, telle que l'on légèrement simplifiée Harper et Mitchell. On ajouté au système F un opérateur J :

$$J : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha$$

Cet opérateur effectue donc un test d'égalité sur les types, matérialisé par la règle de réduction non linéaire suivante :

$$J\ A\ A\ a\ b \rhd b$$

autrement dit, lorsque les deux premiers arguments (les types) sont égaux, on sait que le dernier argument est du type attendu, et on peut rendre ce dernier.

On peut alors construire un point-fixe sur les booléens[1] codés de manière imprédicative. Notons Bo le type des booléens :

$$Bo \equiv \forall\alpha.\alpha \to \alpha \to \alpha$$

On définit :

$$R \equiv \Lambda\alpha.J\ \alpha \to \alpha \to \alpha\ Bo\ (\lambda x : \alpha.\lambda y : \alpha.x)\ (\lambda x : Bo.\lambda y : Bo.(x\ Bo\ x\ x)$$

On a bien $R : Bo$. On voit que ce terme se comporte comme le premier booléen de manière générale $(\lambda x.\lambda y.x)$ mais lorsqu'il est appliqué au type des booléens, il exploite cette information pour construire une auto-application.

C'est ce qui se passe lorsque l'on forme le terme $(R\ Bo\ R\ R)$. Il est alors facile de voir que ce terme se réduit vers lui-même : on a perdu la normalisation forte. Mais ce que nous allons utiliser ci-dessous, c'est que la même technique permet la construction d'un point-fixe. Soit $f : Bo \to Bo$ et prenons :

$$R_f \equiv \Lambda\alpha.J\ \alpha \to \alpha \to \alpha\ Bo\ (\lambda x : \alpha.\lambda y : \alpha.x)\ (\lambda x : Bo.\lambda y : Bo.(f\ (x\ Bo\ x\ x))$$

et $Y_f \equiv (R_f\ Bo\ R_f\ R_f)$; on a alors : $Y_f \rhd (f\ Y_f)$.

[1] S'il s'agit simplement de donner un contre-exemple à la normalisation, on peut se contenter du type singleton $\forall X.X \to X$.

6.2.2 En théorie des types

Nous allons effectuer une construction similaire en théorie des types imprédicative. A la place de J nous allons utiliser une construction reposant sur un axiome : la décidabilité de l'égalité. Du coup nous ne pourrons pas nier la normalisation (heureusement), mais simplement prouver l'existence d'un booléen égal à sa propre négation. De là on pourra déduire que tous les types sont dégénérés (réduit à un singleton).

On va localement supposer l'existence d'une fonction de décision pour l'égalité sur les types de type Set. On se donne donc :

$$\begin{array}{rcl} dec & : & \mathsf{Set} \to \mathsf{Set} \to \mathsf{bool} \\ dec_cor_1 & : & \Pi A : \mathsf{Set}.\Pi B : \mathsf{Set}.A = B \to dec\ A\ B = \mathsf{true} \\ dec_cor_1 & : & \Pi A : \mathsf{Set}.\Pi B : \mathsf{Set}.dec\ A\ B = \mathsf{true} \to A = B \end{array}$$

On peut alors construire l'équivalent de l'opérateur J. La syntaxe détaillée est lourde à écrire sur papier, à cause des types dépendants, mais la définition correspond intuitivement à quelques chose comme :

$$J \equiv \lambda A\ B\ a\ b.if(dec\ A\ B)\ \text{then}\ a'\ \text{else}\ b$$

où a' est essentiellement a coercé vers le type B en utilisant la preuve de $A = B$.

On peut avec cela construire les termes R_f décrits ci-dessus, en particulier en choisissant la négation booléenne negb pour f.

Si l'on suppose également l'unicité des preuves d'égalité, on peut également monter que pour tout $A : \mathsf{Type}$ et tous a et b de type A on a bien :

$$J\ A\ A\ a\ b = a.$$

En combinant, on a donc construit un terme :

$$R_{\text{negb}} : Bo$$

avec une preuve que :

$$R_{\text{negb}} = \text{negb}\ R_{\text{negb}}.$$

Et de là on déduit aisément true=false, et à partir de là que tous les types de type Set sont dégénérés. Donc :

$$dec + K \vdash \bot$$

6.3 Une (autre) version de la construction de Diaconescu

J'ai indiqué, dans le chapitre précédant, un encodage possible de la construction de Diaconescu, qui permet de déduire le tiers-exclu à partir du choix. Dans cette construction on avait utilisé ACET. Nous allons maintenant voir une autre variante de cette construction en théorie des types. Les différences sont :

- On utilise ACIT à la place de ACET. Aussi, cette construction ressemble beaucoup plus à la construction ensembliste telle que présentée par Goodman et Myhill [67].
- On ne transpose en fait que la première partie de la construction de Diaconescu. Aussi on va seulement prouver la *décidabilité de l'égalité* et pas le tiers-exclu.
- Comme ACIT est plus intentionnel, il faut rendre la théorie des types plus intentionnelle. On va pour cela supposer la proof-irrelevance.

Les axiomes utilisés dans cette section sont donc ACIT et la proof-irrelevance :

$$PI : \Pi P : \mathsf{Prop}.\Pi p_1 : P.\Pi p_2 : P.p_1 = p_2.$$

6.3.1 Codage de la paire non-ordonnée

Étant donné $A : \mathsf{Type}_i$, $a : A$ et $b : B$ on note :

$$\{a; b\} \equiv \Sigma x : A.x = A \vee x = B$$

où $=$ est l'égalité propositionnelle de Leibniz, et $\vee$ la disjonction dans Prop.

On note $\overline{a}$, respectivement $\overline{b}$, l'élément de $\{a; b\}$ correspondant à a, respectivement b :

$$\begin{array}{rcl} \overline{a} & \equiv & (a, \mathsf{left}\ (\mathsf{refl}\ a)) \\ \overline{b} & \equiv & (b, \mathsf{right}\ (\mathsf{refl}\ b)) \end{array}$$

où left et right sont les termes correspondants aux règles d'introduction de la disjonction et refl celui correspondant à la réflexivité de l'égalité.

Par ailleurs, notons $\underline{x}$ la projection dans A de $x : \{a; b\}$. C'est-à-dire $\underline{(x, p)} = x$.

En utilisant PI, on vérifie facilement que $x = y$ est vérifié si et seulement si $\underline{x} = \underline{y}$.

6.3.2 Construction de Diaconescu

Par ailleurs, il est facile de montrer :

$$\Pi x : \{a; b\}.\exists e : \mathsf{bool}.(e = \mathsf{true} \wedge \underline{x} = a) \vee (e = \mathsf{false} \wedge \underline{x} = b)$$

et donc avec ACIT :

$$\exists f : \{a; b\} \to \mathsf{bool}.\Pi x : \{a; b\}.(f\ x = \mathsf{true} \wedge \underline{x} = a) \vee (f\ x = \mathsf{false} \wedge \underline{x} = b).$$

On peut maintenant prouver $a = b \vee a \neq b$. Comme c'est une proposition, on peut commencer par éliminer l'existentiel ci-dessus. On dispose donc de la fonction f. Les valeurs $f\ \overline{a}$ et $f\ \overline{b}$ étant des booléens, on regarde les 4 cas de figure possibles pour $(f\ \overline{a}, f\ \overline{b})$.

Si $f\ \overline{a} \neq f\ \overline{b}$, il est évident que $\overline{a} \neq \overline{b}$, et donc aussi $a \neq b$ puisque $\underline{\overline{a}} = a$ et $\underline{\overline{b}} = b$.

Si $f\ \overline{a} = f\ \overline{b}$, on a soit $f\ \overline{a} = \mathsf{false}$ soit $f\ \overline{b} = true$. Considérons le premier cas, le second étant parfaitement symmétrique. On sait que :

$$(f\ \overline{a} = \mathsf{true} \wedge \underline{\overline{a}} = a) \vee (f\ \overline{a} = \mathsf{false} \wedge \underline{\overline{a}} = b)$$

c'est-à-dire ici :

$$\mathsf{false} = \mathsf{true} \wedge a = a) \vee (\mathsf{false} = \mathsf{false} \wedge a = b)$$

dont on déduit facilement $a = b$.

Pour résumer, on a donc :

$$ACIT + PI \vdash dec$$

6.4 Conclusion

En combinant les deux constructions précédentes, on conclut que ACIT et PI suffisent à construire un paradoxe.

Cela montre qu'il est plus que délicat de construire un modèle pour cette théorie des types. En particulier :

- A supposer que l'on a un modèle de la théorie sans axiomes, on valide évidemment ACIT en remplaçant simplement Prop par Set.
- On peut espérer construire un modèle de la théorie où Prop est "classique" et donc en particulier vérifiant PI. Il serait de fait intéressant de construire un tel modèle. Mais il est alors crucial de *ne pas* supposer ACIT, c'est-à-dire que les fonctions constructibles sont strictement moins nombreuses que celles dont on peut prouver l'existence dans Prop.

Quatrième partie

Conclusion

Chapitre 7

Conclusion et Perspectives

L'utilisation intensive du calcul dans les preuves formelles a ouvert de nouveaux horizons à la preuve formelle. Elle pose également de nouvelles difficultés. Nous essayons ici d'évoquer les uns et les autres.

7.1 Développements et avancées récentes

Un certain nombre des travaux décrits ici ont déjà eu une certaine descendance, directe ou indirecte.

Le théorème des quatre couleurs

Comme c'était déjà le cas avant sa formalisation en Coq, ce théorème occupe un place un peu à part. D'une part, à cause de sa notoriété, due au coté spectaculaire à la fois de son énoncé simple et de sa preuve très particulière. D'autre part parce que sa preuve, toute complexe qu'elle soit reste relativement élémentaire en ce sens qu'elle n'utilise que peu la littérature mathématique extérieure ; en anglais on dit que les articles la décrivant sont naturellement *self-contained*.

Une conséquence du premier point a été une certaine couverture médiatique de la preuve en Coq, qui a certainement été encourageante et bénéfique pour le domaine. On peut également espérer que cette visibilité accrue des méthodes formelles dans le monde des mathématiques sera une aide pour la diffusion d'outils comme Coq. Cela a au moins été le cas pour un mathématicien important, puisque c'est, semble-t-il, l'effort de formalisation des quatre couleurs qui a poussé Thomas Hales à s'intéresser à la preuve formelle et à initier la tentative de formalisation de sa preuve de la conjecture de Kepler. Cette dernière est calculatoirement plus complexes que les quatre couleurs, mais surtout elle est plus longue et s'appuie plus sur des pré-existants qui n'ont pas encore été formalisés. On est aujourd'hui encore loin de la formalisation complète de cette preuve, mais on observe déjà un certain nombre de résultats partiels intéressants. Parmi ceux-ci les travaux de Roland Zumkeller, qui portent sur le traitement en Coq des problèmes d'optimisation réelle, et qui recoupent le domaine des preuves formelles numériques, que nous évoquons ci-dessous.

Si le traitement de la théorie des graphes que Georges Gonthier a proposé pour la formalisation des quatre couleurs est particulièrement élégant, on ne voit pas forcément d'autres applications immédiates à ce travail. En revanche, la travail qu'il a effectué pour construire un ensemble de tactiques de preuves bien adaptées à la preuve de propriétés de programme à l'intérieur de Coq commence maintenant à être ré-utilisé. L'ensemble de tactiques, baptisé SSR *(small step reflection)* sera bientôt diffusé et est déjà utilisé, entre autres, pour la formalisation du théorème de Feit-Thompson ou dans notre formalisation de la normalisation par évaluation en Coq. Dans un certain nombre de domaine, ces tactiques induisent un style de formalisation nouveau et prometteur que nous ne détaillons pas plus ici mais qui permet des simplifications décisives dans certains cas.

Nombres premiers

Dans le domaine des preuves de primalité les progrès ont été particulièrement rapides. Depuis le travail décrit dans ce mémoire (et qui date pourtant de moins de deux ans) la taille des nombres premiers certifiables en Coq a cru très largement. La dernière raison en date a été l'achèvement par Laurent Théry et Guillaume Hanrot de la formalisation des résultats portant sur les courbes elliptiques. Ceci permet, en conservant un mode de fonctionnement similaire à celui que nous décrivons, d'obtenir des certificats autrement plus performants, en particulier pour les nombres premiers non-remarquables (typiquement qui ne sont pas des nombres de Mersenne ou de Proth).

C'est à la fois un résultat encourageant quand à la capacité de Coq à permettre la formalisation de techniques proches de l'état de l'art d'un domaine actif, et également satisfaisant car une étape importante dans le développement d'une bibliothèque comportant des outils mathématiques et informatiques utiles pour de futurs travaux (typiquement dans le domaine de la sécurité pour cet exemple précis).

Le traitement des nombres

Une autre raison pour les progrès dans le domaine de la primalité a été le traitement plus efficaces des nombres et des opérations numériques en général en Coq. Pour ce qui est du travail à l'intérieur de Coq, Benjamin Grégoire et Laurent Théry on construit une bibliothèque permettant de construire des grands entiers au dessus d'une représentation de nombres bornés : les nombres sont représentés comme des arbres binaires équilibrés ; les feuilles, regardées de gauche à droite correspondent au "chiffres". C'est typiquement une représentation qui tente de tirer le meilleur d'un environnement purement fonctionnel :

- Elle permet d'accéder en temps assez rapide (logarithmique par rapport au nombre de chiffres) à la fois au chiffres de poids faible et de poids fort. On ne favorise donc ni l'addition (poids faible d'abord) ni la division (poids fort d'abord).
- Elle permet naturellement d'utiliser un certain nombre d'algorithmes efficaces de type Karatsuba qui s'appellent récursivement sur les moitiés de poids fort et de poids faible des nombres (pour la multiplication bien sûr, mais aussi pour la division ou la racine carrée).

Il s'agit là encore d'une avancée importante, mais pas inattendue, des bibliothèques Coq allant vers plus d'efficacité et utilisant plus de technologie informatique.

Une autre avancée est due à la représentation de ces entiers bornés. J'ai encadré le travail de stage de M2 d'Arnaud Spiwack qui partait d'une constatation simple : si, au moins dans un domaine particulier, la capacité du système à vérifier des preuves repose sur sa capacité à effectuer des opérations numériques, il est difficilement justifiable de se priver de l'implémantation de ces opérations "en dur" dans le micro-processeur. Après tout, le CPU a été conçu en grande partie pour cela. De plus, les fabriquants font de grand efforts pour garantir la correction de ses opérations et, en particulier pour ce qui est des entiers, elles peuvent être considérées comme fiables.

Nous avons donc proposé le schéma suivant :

1. Un type est construit en Coq, qui correspond précisément aux entiers bornés, ou mots machine, de taille idoine (en fait des vecteurs de n bits).
2. Les opérations "natives" sur ces mots sont construites comme des fonctions Coq (addition, multiplication modulo 2^n, etc...).
3. Il est alors possible de demander au compilateur de Coq de ne plus traiter les objets de type "mots machines" en utilisant les méthodes génériques pour les type inductifs, mais en construisant une forme de valeur particulière reposant sur les "vrais" mots machines. Par "vrai" il faut comprendre le type `int` C de l'environnement d'exécution du byte-code Coq. Ce type `int` qui reposant lui-même sur les entiers du processeur. De même, les opérations mentionnées ci-dessus sont, à *compile-time* déléguées aux routines du processeur.

On voit qu'il ne s'agit ici ni de changer en quoi que ce soit la théorie des types, ni même de nouveau développement *en Coq*. Il s'agit simplement d'optimiser le traitement d'un type de données particulier. On obtient alors un gain important en vitesse d'exécution et en utilisation de la mémoire, au prix d'une plus grande complexité du mécanisme de vérification de preuves.

D'un coté, il s'agit là donc là d'une évolution naturelle : à partir du moment où l'on a décidé de recourir à un mécanisme de compilation pour permettre au noyau du système de vérifier certaines preuves plus rapidement, on est également conduit à adjoindre à ce mécanisme un certain nombre d'optimisation comme ce traitement "natif" des mots machines. D'un autre coté on peut se poser la question si (1) d'autres optimisations de ce type s'avèreront désirables et (2) si la plus grande complexité du code du noyau qui en résulte de risque pas de nuire à la fiabilité de celui-ci en sapant ainsi les fondements même de ce que l'on cherche à faire : garantir la plus grande certitude possible. Je reviens sur cet aspect dans le paragraphe 7.4.

7.2 Développements possibles du système de preuve

Autour du mécanisme d'exécution

Une très grande partie des nouveaux travaux exposés ou mentionnés ici ont donc été rendu possible par l'adjonction d'un mécanisme de calcul performant à l'intérieur de la vérification de preuve : l'utilisation à l'intérieur de Coq des technologies de compilation et d'exécution venues des langages de programmation et de Caml en particulier.

Comme l'indique le traitement particulier des nombres décrit ci-dessus, il est alors légitime de se poser la question de quels nouveaux mécanismes on pourrait ou voudrait ajouter au schéma de compilation. Ce dernier se limite actuellement à un langage fonctionnel pur, avec donc un traitement un peu particulier pour les nombres. La raison de cet état de fait est claire : les programmes fonctionnels sont immédiatement reliés à des objets mathématiques, comme c'est illustré par la sémantique ensembliste simple que nous avons présenté [1]. Dans un certain nombre de cas toutefois, cette restriction à des structures purement fonctionnelles peut apparaître contraignante lors de l'écriture de programmes que l'on veut efficaces. En ne considérons que mes propres travaux, on peut déjà mentionner :

- La représentation des ensembles de coloriages et de chrommogrammes dans la preuve des quatre couleurs. Devoir les représenter par des arbres est beaucoup moins compact que l'utilisation de tableaux de booléens, et ce alors que la question de la mémoire n'est pas loin d'être limitante. De plus, le processus calculatoire revient essentiellement à faire croître[2] ces ensembles. Un tel processus itératif est plus efficacement traité par des mises à jour impératives de ces tableaux, plutôt que la reconstruction systématique de structures de données arborescentes.
- Dans le cas des grands nombres, la représentation arborescente proposée par Benjamin Grégoire et Laurent Théry est sans doute proche de ce que l'on peut faire de mieux dans un cadre purement fonctionnel. De fait, les algorithmes efficaces y sont exprimés et implémentés naturellement. Il n'empêche que dans la pratique, les spécialistes de l'arithmétique des ordinateurs savent bien que les bibliothèques de grands nombres (qu'ils soient entiers comme dans GMP ou flottants comme dans MPFR) sont beaucoup plus efficaces lorsqu'elle reposent sur des tableaux. La raison est que c'est l'allocation de mémoire qui est la tâche la plus coûteuse lorsque l'on manipule de telle données. Aussi, en GMP ou MPFR, l'utilisation et l'allocation de la mémoire ont fait l'objet d'un soin tout particulier, avec une précision qui n'est pas accessible à travers des structures fonctionnelles de plus haut niveau.
- Même s'il s'agit là d'un détail, le traitement de la normalisation par évaluation que nous donnons dans l'article commun avec François Garillot est un exemple de calcul où l'impossibilité de recourir à des effets de bord oblige à recourir à un algorithme beaucoup moins efficace.

Il n'est pas difficile de trouver d'autres exemples de caractéristiques *(features)* de langages de programmation que dont l'on aimerait parfois disposer dans un développement formel. Les exceptions ou certains traits orientés-objets sont sans doute les plus courants.

Or l'on dispose, au moins depuis la thèse de Jean-Christophe Filiâtre, d'une traduction depuis des langages avec de tels traits impératifs vers le fonctionnel pur qui a la particularité d'être bien typée et surtout de bien

[1]Du point de vue de la recherche en langages de programmation, cette sémantique correspond d'ailleurs à une sémantique dénotationelle très simple, voire quasi-triviale. La question du traitement de la proof-irrelevance décrit dans l'article commun avec Alexandre Miquel étant finalement le seul point délicat.

[2]ou décroître, si l'on considère les ensembles complémentaires.

se comporter vis-à-vis des propriétés des programmes : on passe d'une preuve sur le programme fonctionnel à une preuve sur le programme impératif.

On peut donc imaginer des travaux visant à utiliser de telles traductions inversées, pour optimiser la compilation : on prétend raisonner sur les programmes fonctionnels, mais à l'exécution, ceux-ci sont remplacés par leur contre-partie impérative, plus efficace.

Bien sûr, il s'agit là d'une proposition encore très schématique, et je n'ai personnellement pas de plan de travail précis sur ce sujet. Il me semble néanmoins probable qu'à l'avenir, l'on sera soumis à une certaine pression pour étendre le mécanisme de compilation, en allant vers plus d'efficacité mais aussi plus de complexité.

Traitement fin de la réduction

Toute l'idée des diverses preuves calculatoires présentées dans ce mémoire repose sur exploitation intensive de la règle de conversion de la théorie des types pour échanger du calcul contre du raisonnement et raccourcir ainsi les preuves. Ceci est rendu possible en choisissant une représentation des objets qui est entièrement orienté vers la réduction (représentation sous forme de byte-code exécutable).

Il est toutefois des situations où l'on voudra mieux contrôler quelles réductions doivent être considérées ou non. En particulier, la question des *modules* dans un environnement comme celui de Coq pose encore de nombreux problèmes, qui sont souvent liés à la question : dans quelle mesure les règles de réduction font-elles partie de la *signature* du module ?

C'est une question très complexe et qui reste aujourd'hui encore posée de manière trop imprécise. Aussi je ne saurais encore proposer de plan d'attaque précis. Il n'est toutefois pas impossible qu'une partie des réponses soient liées à deux autres de mes travaux, que je n'ai pas évoqués dans ce mémoire :

- La déduction modulo, où justement le formalisme est *paramétré* par un ensemble de règles de réécritures. La plupart des résultats théoriques et méta-théoriques sont génériques et portent sur de larges classes d'ensembles de règles de réécriture. Aussi s'agit-il d'un outil intéressant pour comprendre un cadre où l'ensemble des règles de calcul peut varier (par exemple être différent suivant qu l'on soit en dedans ou en dehors d'un module).
- L'interprétation calculatoire de l'opérateur ε de Hilbert, qui propose une approche logique de la question des modules informatiques. Un point intéressant dans ce travail est qu'il propose également une machine abstraite (donc la possibilité de compilation efficace) qui réussit à faire cohabiter *plusieurs versions* d'un même module. Si on réussissait à transposer ces constructions dans le cadre d'une théorie des types, cela voudrait dire plusieurs versions (ou plusieurs comportements calculatoires) d'un même objet mathématique. Il s'agit là d'une piste encore très imprécise, mais au moins je suis persuadé (et je crois que mes co-auteurs Martin Abadì et Georges Gonthier le sont aussi) que ce travail n'a pas encore eu la descendance qu'il pourrait avoir.

7.3 Directions pour de nouvelles formalisations

Traitement des langages avec lieurs

L'informatique théorique, et en particulier l'étude des langages de programmation est un exemple de discipline où la taille des objets étudiés, des définitions, et donc aussi des preuves a fortement crue au cours des dernières années. Les informaticiens théoriques travaillant dans ce domaine se trouvent donc très naturellement confrontés à une question de fiabilité de leur résultats. En particulier, la vérification par des pairs de nouveaux travaux peut, dans certains cas, demander un temps important ; ce n'est pas sans conséquence dans un domaine dynamique à une époque où de plus la pression est forte sur les chercheurs pour produire des publications nombreuses et rapides.

Dans ce contexte est né le *POPLmark challenge* : une initiative autour des chercheurs de l'Unversité de Pennsylvanie, qui visait à faire le point sur les capacités des systèmes de preuves à traiter de la formalisation des langages de programmation. Or si les preuves en question, très syntaxiques et inductives, ne mettent en

général pas ces systèmes mal à l'aise, il reste un point généralement considéré comme délicat : le traitement formel de la liaison de variables.

En effet, la présence de ces liaisons fait que la syntaxe des programmes sort, en général, du cadre des algèbres libres (ou des arbres simples) qui est lui très facilement traité sous forme de définitions inductives. Lorsque l'on formalise, par exemple, le λ-calcul en théorie des types, on se retrouve immédiatement confronté au choix habituel entre variables nommées et indices de de Bruijn.

- Avec les variables nommées, on soufre très vite des maux habituels de cette représentation : nécessité de raisonner modulo α-conversion, conditions très délicates sur les variables fraîches, etc...Surtout, le fait de travailler dans un cadre formel rend en général ces questions encore plus délicates et coûteuses en temps et en énergie.
- Avec les indices de de Bruijn, il faut définir de nombreuses fonctions de *lifting* des indices et introduire celles-ci aux bons endroits lorsque l'on "traduit" formalise les versions habituelles des lemmes. Il ne s'agit là pas forcément d'un travail aussi dur que le traitement formel de l'α-conversion ; il n'empêche qu'il s'agit là d'un travail qui doit être effectué à nouveau pour chaque nouvelle structure avec lieurs qui doit être introduite.

Bien sûr, je suis conscient que cette dichotomie est une présentation un peu simpliste, surtout aujourd'hui ce domaine de recherche est extrêmement actif est a donné lieu à plusieurs autres approches intéressantes (nominal package, Fresh ML, etc...).

Il n'empêche qu'il serait encore intéressant de disposer en théorie des types, et idéalement dans une théorie des types non modifiée, d'un mécanisme simple de représentation des langages avec lieurs, qui soit à la fois pratique quant à la construction des preuves formelles et qui évite un travail d'implémentation répétitif *(boilerplate)*.

Or il se trouve que dans le cadre d'un formalisme plus faible, LF *(Logical Framework)*, il existe une approche particulièrement concise : la Syntaxe Abstraite d'Ordre Supérieur ou HOAS. L'idée est simplement d'utiliser le mécanisme de lieur du formalisme lui-même pour représenter les lieurs du langage étudié. Dans le cas du λ-calcul pur, ce revient à dire que les objets d'un type `term` sont canoniquement construits à partir de deux variables `App : term→term→term` et `Lam : (term→term)→term`. Par exemple l'identité $\lambda x.x$ est alors représentée par (en syntaxe Coq) : `(Lam (fun x:term => x))`.

Il a tout de suite été remarqué que cette idée ne pouvait être transposée en *définition inductive* en théorie des types, à cause de *l'occurrence négative* de `term` dans le type de `Lam`. Le problème vient du fait qu'en LF, les fonctions de type `term→term` sont simplement typées, c'est-à-dire qu'il s'agit d'objet de type `term` paramétrés par une variable. En théorie des types en revanche, on peut construire de nombreuses fonctions de type `term→term` ; par exemple la fonction qui rendra soit la variable x soit la variable y suivant la taille de l'argument. Appliquer une telle fonction à `Lam` donne alors des objets de type `term` qui ne correspondent à aucun λ-terme.

Il existe divers travaux proposant des extensions modales de la théorie des types, qui permettent de "marquer" les fonctions qui correspondent au fragment simplement typé ; mais ces propositions sont relativement complexes et n'ont pas donné lieu, à ma connaissance, à des implémentations.

Je propose donc une approche qui ne nécessiterait pas de changement dans la théorie des types :

- On garde l'idée de coder les objets du langage avec lieur comme des λ-termes simplement typés ; un langage avec lieurs correspondants en fait à un contexte de typage simplement typé (par exemple le λ-calcul pur correspond au contexte `[Lam ; App]`).
- Par contre, on utilise un *plongement profond* du λ-calcul simplement typé en théorie des types. C'est-à-dire que l'on travaille une fois pour toutes, pour construire une représentation des λ-termes simplement typés en Coq qui soit suffisamment confortable et supporte les bons principes de récurrence structurelle. Les objets d'un langage quelconque avec lieurs codés comme des termes simplement typés à travers un codage "à la HOAS" héritant alors de ces principes.
- Enfin le travail que j'ai effectué avec François Garillot permet, si nécessaire, de passer du plongement profond des types simples en Coq à un plongement *superficiel* ; c'est-à-dire qu'il est possible de retrouver le lien entre le mécanisme de liaison du langage codé et celui de Coq lui-même.

Il s'agit là d'un projet de travail reposant sur quelques idées théoriques mais dont la mise en œuvre est maintenant essentiellement une question technique d'implémentation. Le succès ou non des ces idées

dépendant sans doute de la simplicité et du confort de l'outil résultant.

Calcul Formel

Bien que ce dernier point ne concerne pas vraiment mes projets de recherche personnels, je voudrais mentionner ce que j'espère être les liens croissant entre calcul formel et systèmes de preuve comme Coq.

J'ai, dans ce mémoire, essayer de militer pour l'idée que le calcul pouvait être la bonne manière d'établir certains faits mathématiques et que les implémentations de la théorie des types étaient le lieu où cela pouvait être fait de manière sûre : les calculs sont moins rapides que dans un système dédié, mais les résultats sont aussi crédibles que possibles, et surtout le comportement des programmes est formellement spécifié et prouvé.

Or le Calcul Formel est une discipline, s'il en est, où le calcul informatique est utilisé pour établir des fait mathématiques. Autrement dit, on peut le voir comme un réservoir des techniques qui ne demandent qu'à être importées dans un système comme Coq. Parmi de tels travaux, on peut citer ceux d'Assia Mahboubi [80] sur l'algorithme CAD. L'intérêt des tels travaux et clair :

- ils présentent un intérêt scientifique propre : la certification de routines complexes de calcul formel est un travail intéressant et porteur de difficultés propres,
- ils permettent de disposer de versions spécifiées et certifiées des procédures en question,
- mais aussi chaque nouvelle procédure certifiées augmente immédiatement le pouvoir du système de preuve : elle peut être utilisée pour établir auto-maqiquement des nouveaux faits, et en particulier pour certifier à nouveau d'autres procédures plus complexes...

En d'autres termes, on a là un *cercle vertueux*. Notre capacité collective à entrer dans ce cercle vertueux en développant les bibliothèques de Coq et en incitant plus d'informaticiens à le faire sera décisive pour faire d'un tel système une plate-forme de plus en plus crédible pour le raisonnement mathématique appuyé par le calcul quand c'est désirable ou nécessaire.

7.4 Un système aussi sûr que possible

Une force de la théorie des types est, nous l'avons mentionné dès l'introduction, la décidabilité de la vérification des preuves. C'est cela qui permet la notion de *noyau* dans l'architecture logicielle du système : seule la partie vérifiant les preuves achevées est critique pour la crédibilité du système.

Cette crédibilité est l'argument premier de tout système de preuves. Or il faut noter que la plupart des évolutions que nous avons décrites dans ce mémoire induisent des contraintes fortes sur cette crédibilité :

- Des preuves comme celle des quatre couleurs concernent des vérités mathématiques auxquelles on ne peut accéder qu'à travers des calculs importants. On perd donc toute chance de revenir à une *compréhension intuitive* de la démonstration dans son ensemble. Les étapes *claires et distinctes* dont pouvait parler Descartes sont trop nombreuses pour être parcourues dans le temps d'une vie humaine. On n'a d'autre choix que de faire une confiance "aveugle" à la machine.
- Les preuves calculatoires sont devenues vraiment intéressantes qu'en intégrant un mécanisme de calcul efficace au noyau de Coq. L'étape décisive a été l'utilisation de la compilation par Benjamin Grégoire. Mais ce travail a également signifier un élargissement significatif du noyau qui intègre à partir de là un compilateur, un environnement d'exécution de byte-code moderne, bref un petit monde informatique à lui tout seul. Qui plus est, j'ai évoqué plus haut comment il est naturel à partir de là d'au moins considérer des extensions du mécanisme de compilation/exécution : les nombres en tout cas, des traits de programmation impérative un jour peut-être. Il y a donc une question importante quant à la bonne taille et la bonne complexité que doit avoir le noyau.
- Enfin le développement incrémental de nouvelles bibliothèques calculatoires, typiquement de routines de décision automatique et de calcul formel conduit de plus en plus à des preuves où l'intuition mathématique est ponctuellement, mais de plus en plus fréquemment, remplacée par l'appel au calcul. Là encore, cela ne fait du sens que si la confiance accordée au mécanismes de calcul et ses différents niveaux (mécanisme d'exécution, correction des procédures certifiées...) est suffisante.

Gérard Huet a donc fait récemment la remarque qu'il faudrait un jour valider le noyau de Coq en appliquant les critères de validation de logiciels critiques qui sont justement en train de s'imposer dans l'informatique industrielle. Il s'agit là d'une tâche certes difficile mais, les points ci-dessus le montrent, au moins utile sinon nécessaire. Si un tel travail peut être effectué, ce ne pourra être qu'à travers un effort collectif important. Je voudrais juste mentionner quelques pistes à ce sujet :

- Le noyau doit être aussi petit que possible, aussi grand que nécessaire. C'est particulièrement vrai des mécanismes d'exécution. Or, par exemple, des calculs numériques performants ne sont nécessaires que pour certaines preuves et certains domaines. Je pense donc que va se généraliser la possibilité de brancher ou débrancher certains *features* du noyau, en particulier ceux ayant trait au calcul (compilation, nombres machines...).
- C'est également vrai pour le formalisme lui-même. L'un des points les plus délicats à décrire dans le formalisme de Coq[3] et le critère définissant ce qu'est un appel récursif légal dans la théorie des types. Par ailleurs, proposer des nouveaux critères plus confortables est l'un des sujets de recherche les plus actifs du domaine actuellement et donne lieu à de très nombreuses publications. Il me semble là aussi que l'on sera conduit à proposer un mécanisme d'options permettant d'appeler Coq avec différentes variantes plus ou moins libérales de tels critères. Dans l'idéal, on devrait pouvoir proposer un mode "le plus sûr" qui corresponde à une version de la théorie qui soit suffisamment comprise pour avoir été elle-même formellement vérifiée (avec bien sûr le "petit axiome" nécessaire de part le théorème d'incomplétude de Gödel).
- Enfin bien sûr, il y a la question de la validation formelle aussi poussée que possible du code du noyau lui-même. Les travaux de Bruno Barras ouvrent là voie à de telles possibilités ; surtout si ils sont combinés avec les progrès faits dans le domaine de la certification de code impératif (on pense à ce qu'a produit l'équipe-projet ProVal ces dernières années).

On le voit, le travail est important. Il est difficile en le considérant de ne pas avoir la nostalgie d'un certaine simplicité perdue ; celle du formalisme si concis du Calcul des Constructions originels, ou celle des implémentations *small in beautiful* des débuts. Mais justement, s'il y a une thèse qui me semble s'être imposée à moi ces dernières années ces que si jusqu'à maintenant, c'était l'ingénieur qui avait besoin des mathématiques, dorénavant, ce seront peut-être de plus en plus de mathématiciens qui auront besoin de faire confiance à l'ingénieur.

[3]Et d'ailleurs il n'est pas vraiment décrit dans ce mémoire.

Bibliographie

[1] Martín Abadi, Georges Gonthier, and Benjamin Werner. Choice in dynamic linking. In Igor Walukiewicz, editor, *FoSSaCS*, volume 2987 of *Lecture Notes in Computer Science*, pages 12–26. Springer, 2004.

[2] Peter Aczel. The type theoretic interpretation of constructive set theory. In *Logic Colloquium 77*. Springer, 1977.

[3] Peter Aczel. The type theoretic interpretation of constructive set theory : Choice principles. In *The L. E. J. Brouwer Centenary Symposium*. North-Holland, 1982.

[4] Peter Aczel. The type theoretic interpretation of constructive set theory : Inductive definitions. In *Proceedings of Methodology and Philosophy of Sciences*, 1985.

[5] Peter Aczel. On relating type theories and set theories. In Thorsten Altenkirch, Wolfgang Naraschewski, and Bernhard Reus, editors, *TYPES*, volume 1657 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1998.

[6] Thorsten Altenkirch. *Constructions, Inductive Types and Strong Normalization*. PhD thesis, University of Edinburgh, 1993.

[7] Thorsten Altenkirch and Conor McBride, editors. *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*. Springer, 2007.

[8] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses : The poplmark challenge. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.

[9] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2. Elsevier, 1992.

[10] Henk Barendregt and Erik Barendsen. Autarkic computations in formal proofs. *J. Autom. Reasoning*, 28(3) :321–336, 2002.

[11] Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In *Workshop on Type theory, Proof theory, and Rewriting*, 2007.

[12] Bruno Barras and Benjamin Grégoire. On the role of type decorations in the calculus of inductive constructions. In *CSL'05*. LNCS, Springer-Verlag, August 22-25,2005, Oxford, UK.

[13] Laurent Théry Benjamin Grégoire and Benjamin Werner. A computational approach to pocklington certificates in type theory. In M. Hagiya and P. Wadler, editors, *FLOPS 2006*, volume 3945 of *LNCS*. Springer, 2006.

[14] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[15] G. D. Birkhoff. The reducibilit of maps. *American Journal of Mathematics*, 35 :115–128, 1913.

[16] Frédéric Blanqui, Jean-Pierre Jouannaud, and Albert Rubio. Higher-order termination : From kruskal to computability. In Miki Hermann and Andrei Voronkov, editors, *LPAR*, volume 4246 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2006.

[17] Frédéric Blanqui, Jean-Pierre Jouannaud, and Pierre-Yves Strub. Building decision procedures in the calculus of inductive constructions. In Jacques Duparc and Thomas A. Henzinger, editors, *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2007.

[18] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *TACS*, pages 515–529, 1997.

[19] J. Brillhart, D. H. Lehmer, and J. L. Selfridge. New primality criteria and factorizations of $2^m \pm 1$. *Mathematics of Computation*, 29 :620–647, 1975.

[20] John Brillhart, D. H. Lehmer, J. L. Selfridge, Bryant Tuckerman, and S. S. Wagstaff, Jr. *Factorizations of* $b^n \pm 1$, volume 22 of *Contemporary Mathematics*. American Mathematical Society, Providence, R.I., 1983. $b = 2$, 3, 5, 6, 7, 10, 11, 12 up to high powers.

[21] G. Dowek C. Muñoz and V. Carreño. Modeling and verification of an air traffic concept of operations. In *International Symposium on software testing and analysis*, 2004.

[22] Olga Caprotti and Martijn Oostdijk. Formal and efficient primality proofs by use of computer algebra oracles. *Journal of Symbolic Computation*, 32(1/2) :55–70, July 2001.

[23] Benjamin Werner Christine Paulin-Mohring. Synthesis of ml programs in the system coq. *Journal of Symbolic Computation*, 15 :607–640, 1993.

[24] Jacek Chrząszcz. Implementation of modules in the coq system. In *Theorem Proving in Higher Order Logic, TPHOLs 2003*, volume 2758 of *LNCS*, pages 270–286. Springer, 2003.

[25] Jacek Chrzaszcz and Jean-Pierre Jouannaud. From obj to ml to coq. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Essays Dedicated to Joseph A. Goguen*, volume 4060 of *Lecture Notes in Computer Science*, pages 216–234. Springer, 2006.

[26] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5 (1) :56–68, 1940.

[27] Claudio Sacerdoti Coen. A semi-reflexive tactic for (sub-)equational reasoning. In Filliâtre et al. [48], pages 98–114.

[28] Thierry Coquand. An analysis of girard's paradox. In *Proceeding of LICS*. IEEE press, 1985.

[29] Thierry Coquand. Metamathematical investigations of a Calculus of Constructions. In P. Oddifredi, editor, *Logic and Computer Science*. Academic Press, 1990.

[30] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3) :95–120, 1988.

[31] Thierry Coquand and Arnaud Spiwack. Towards constructive homological algebra in type theory. In *Proceedings of 14th Symposium, Calculemus 2007, 6th International Conference, MKM 2007*. Springer, 2007.

[32] Pierre Corbineau. First-order reasoning in the Calculus of Inductive Constructions. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES 2003 : Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 162–177. Springer-Verlag, 2004.

[33] Judicaël Courant. *Un calcul de modules pour les systèmes de types purs*. Thèse de doctorat, Ecole Normale Supérieure de Lyon, 1998.

[34] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2007.

[35] N. G. de Bruijn. A survey of the project automath. In *To H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[36] Radu Diaconescu. Axiom of choice and complementation. *Proceedings of A.M.S.*, 51, 1975.

[37] G. Dowek. La théorie des types et les systèmes informatiques de traitement des démonstrations mathématiques. *Mathématiques et Sciences Humaines*, 165 :13–29, 2004.

[38] G. Dowek. *Les Métamorphoses du Calcul.* Le Pommier, 2007.

[39] G. Dowek and B. Werner. Proof normalization modulo. *Journal of Symbolic Logic*, 68-4 :1289–1316, 2003.

[40] G. Dowek and B. Werner. Arithmetic as a theory modulo. In J. Giesel, editor, *Term rewriting and applications*, pages 423–437. Lecture Notes in Computer Science 3467, Springer-Verlag, 2005.

[41] Gilles Dowek. Confluence as a cut elimination property. In Robert Nieuwenhuis, editor, *RTA*, volume 2706 of *LNCS*, pages 2–13. Springer, 2003.

[42] Gilles Dowek. What do we know when we know that a theory is consistent ?. In Robert Nieuwenhuis, editor, *CADE*, volume 3632 of *Lecture Notes in Computer Science*, pages 1–6. Springer, 2005.

[43] Gilles Dowek. Truth values algebras and proof normalization. In Altenkirch and McBride [7], pages 110–124.

[44] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Binding logic : Proofs and models. In Matthias Baaz and Andrei Voronkov, editors, *LPAR*, volume 2514 of *Lecture Notes in Computer Science*, pages 130–144. Springer, 2002.

[45] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *J. Autom. Reasoning*, 31(1) :33–72, 2003.

[46] Gilles Dowek and Olivier Hermant. A simple proof that super-consistency implies cut elimination. In Franz Baader, editor, *RTA*, volume 4533 of *Lecture Notes in Computer Science*, pages 93–106. Springer, 2007.

[47] Peter Dybjer. Inductive sets and families in martin-löf's type theory and their set-theoretic semantics. In *Logical Frameworks.* Cambridge University Press, 1991.

[48] Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors. *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, volume 3839 of *Lecture Notes in Computer Science.* Springer, 2006.

[49] François Garillot and Benjamin Werner. Simple types in type theory : Deep and shallow encodings. In Schneider and Brandt [95], pages 368–382.

[50] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur, Thèse d'Etat.* PhD thesis, Université Paris 7, 1972.

[51] Jean-Yves Girard. La mouche dans la bouteille (en mémoire de jean van heijenoort). In *Logic Colloquium '85.* North-Holland, 1987.

[52] G. Gonthier. A computer checked proof of the four-color theorem. available on the web, 2005.

[53] Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry. A modular formalisation of finite group theory. In Schneider and Brandt [95], pages 86–101.

[54] Georges Gonthier and Benjamin Werner. Le théorème des quatre couleurs : ingénierie d'une preuve formelle. *La lettre de l'Académie des sciences*, 21, 2007.

[55] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science.* Springer, 1979.

[56] B. Grégoire. *Compilation des termes de preuves : un (nouveau) mariage entre Coq et Ocaml.* Thése de doctorat, spécialité informatique, Université Paris 7, école Polytechnique, France, December 2003.

[57] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pages 235–246. ACM Press, 2002.

[58] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *ICFP*, pages 235–246, 2002.

[59] Benjamin Grégoire and Laurent Théry. A purely functional library for modular arithmetic and its application to certifying large prime numbers. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 423–437. Springer, 2006.

[60] John Harrison and Laurent Théry. A skeptic's approach to combining HOL and Maple. *J. Autom. Reasoning*, 21(3) :279–294, 1998.

[61] Heinrich Heesch. Untersuchungen zum vierfarbenproblem. 80/a/b, 1969.

[62] H. Herbelin. *C'est maintenant qu'on calcule, au cœur de la dualité*. PhD thesis, Université Paris-Sud, 2005. Habilitation à diriger des Recherches.

[63] Hugo Herbelin. On the degeneracy of sigma-types in presence of computational classical logic. In Pawel Urzyczyn, editor, *Seventh International Conference, TLCA '05, Nara, Japan. April 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 209–220. Springer, 2005.

[64] Olivier Hermant. *Méthodes Sémantiques en Déduction Modulo*. PhD thesis, Université Paris 7 - Denis Diderot, 2005.

[65] Arend Heyting. *Intuitionism : an introduction*. North-Holland, 1956.

[66] Douglas Howe. On computational open-endedness in martin-löf's type theory. In *LICS*. IEEE, 1991.

[67] Nicholas D. Goodman John Myhill. Choice implies excluded middle. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 24, 1978.

[68] Alfred Kempe. On the geographical problem of the four colours. *American Journal of Mathematics*, 2 (part 3) :271–283, 1879.

[69] Wolfgang Haken Kenneth Appel. Every planar map is four colorable. *Bull. Amer. Math. Soc.*, 82 :711–712, 1976.

[70] Wolfgang Haken Kenneth Appel. Every planar map is 4-colorable – 1 : Discharging. *Illinois Journal of Mathematics*, 21 :421–490, 1977.

[71] Wolfgang Haken Kenneth Appel. Every planar map is 4-colorable – 2 : Reducibility. *Illinois Journal of Mathematics*, 21 :491–567, 1977.

[72] Florent Kirchner. *Interoperable proof systems*. PhD thesis, École Polytechnique, 2007.

[73] Jean-Louis Krivine. *Théorie Axiomatique des Ensembles*. Presses Universitaires de France, 1969.

[74] Kenneth Kunen. *Set Theory, An Introduction – Independence Proofs*. North-Holland, 1980.

[75] Edmund Landau. *Grunlagen der Analysis*. Akademische Verlagsgesellschaft, Leipzig, 1930.

[76] Serge Lang. *Algebra*, volume 211 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, third edition, 2002.

[77] Pierre Letouzey. A new extraction for coq. In Freek Wiedijk Herman Geuvers, editor, *Proceedings of TYPES 2002*, volume 2646 of *LNCS*, 2003.

[78] Pierre Letouzey. *Programmation fonctionnelle certifiée : l'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-sud, 2004.

[79] Zaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.

[80] Assia Mahboubi. *Contributions à la certification des calculs dans R : théorie, preuves, programmation*. PhD thesis, Université de Nice-Sophia Antipolis, 2005.

[81] Per Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, 1984.

[82] Per Martin-Löf. 100 years of zermelo's axiom of choice : what was the problem with it ? *Comput. J.*, 49(3) :345–350, 2006.

[83] Paul-André Melliès and Benjamin Werner. A generic normalisation proof for pure type systems. In Eduardo Giménez and Christine Paulin-Mohring, editors, *TYPES*, volume 1512 of *Lecture Notes in Computer Science*, pages 254–276. Springer, 1996.

[84] Alexandre Miquel and Benjamin Werner. The not so simple proof-irrelevant model of cc. In Herman Geuvers and Freek Wiedijk, editors, *TYPES*, volume 2646 of *LNCS*, pages 240–258. Springer, 2002.

[85] César Muñoz, Victor Carreño, Gilles Dowek, and Ricky W. Butler. Formal verification of conflict detection algorithms. *STTT*, 4(3) :371–380, 2003.

[86] J. Narboux. *Formalisation et automatisation du raisonnement géométrique en Coq*. Thése de doctorat, spécialité informatique, Université Paris-Sud, September 2006.

[87] Julien Narboux. A graphical user interface for formal proofs in geometry. *the Journal of Automated Reasoning special issue on User Interface for Theorem Proving*, 2006. to appear.

[88] Julien Narboux. Mechanical theorem proving in Tarski's geometry. In *Proceedings of Automatic Deduction in Geometry 06*, 2006.

[89] C. Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[90] Christine Paulin-Mohring. *Extraction de Programmes dans le Calcul des Constructions*. PhD thesis, Université Paris 7, 1989. Thèse de Doctorat.

[91] Henry C. Pocklington. The determination of the prime or composite nature of large numbers by Fermat's theorem. *Proceedings of the Cambridge Philosophical Society*, 18 :29–30, 1914.

[92] John C. Reynolds. Polymorphism is not set-theoretic. In *Proceedings Int. Symp. on Semantics of Data Types, Sophia-Antipolis*, volume 173 of *LNCS*. Springer, 1984.

[93] Neil Robertson, Daniel P. Sanders, Paul D. Seymour, and Robin Thomas. The four-colour theorem. *J. Comb. Theory, Ser. B*, 70(1) :2–44, 1997.

[94] N. Shankar S. Owre. The formal semantics of pvs. Technical report, SRI, Revised March 1999. Technical Report CSL-97-2R.

[95] Klaus Schneider and Jens Brandt, editors. *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*. Springer, 2007.

[96] Mathieu Sozeau. Subset coercions in coq. In Altenkirch and McBride [7].

[97] Arnaud Spiwack. Ajouter des entiers machine à coq. 2006.

[98] P. G. Tait. Note on a theorem in the geometry of position. *Transactions of the Royal Society of Edinburgh*, 29 :657–660, 1880.

[99] Paul Taylor. Intuitionistic sets and ordinals. *Journal of symbolic Logic*, 61(3) :705–744, 1996.

[100] The GMP Team. *GNU Multiple Precision Arithmetic Library*. `http ://www.swox.com/gmp/`.

[101] The Coq development team. The coq proof assistant reference manual v7.2. Technical Report 255, INRIA, France, mars 2002. `http ://coq.inria.fr/doc8/main.html`.

[102] Christine Paulin-Mohring Thierry Coquand. Inductively defined types. In *Proceedings of Colog'88*, volume 417 of *LNCS*. Springer-Verlag, 1990.

[103] Gérard Huet Thierry Coquand. The calculus of constructions. *Information and Computation*, 76(2/3), 1988.

[104] Daria Walukiewicz-Chrząszcz. *Termination of Rewriting in the Calculus of Constructions*. PhD thesis, Warsaw University and Université de Paris-Sud, 2003.

[105] B. Werner. La vérité et la machine. In Jacques Istas Etienne Ghys, editor, *Images des Mathématiques – 2006*. Société Mathématique de France, 2006.

[106] Benjamin Werner. Sets in types, types in sets. In Martín Abadi and Takayasu Ito, editors, *TACS*, volume 1281 of *Lecture Notes in Computer Science*, pages 530–346. Springer, 1997.

[107] Benjamin Werner. On the strength of proof-irrelevant type theories. In U. Furbach and N. Shankar, editors, *Int. Joint Conf. Automated Reasoning — IJCAR 2006*, volume 4130 of *LNAI*. Springer, 2006.

[108] Herbert Wilf. Mathematics : An experimental science. to appear, 2005.

[109] Paul Zimmermann, Pierrick Gaudry, Jim Fougeron, Laurent Fousse, Alexander Kuppa, and Dave Newman. *Elliptic Curve Method Library*. `http ://www.loria.fr/~zimmerma/records/ecmnet.html`.

[110] Roland Zumkeller. Formal global optimisation with taylor models. In U. Furbach and N. Shankar, editors, *Int. Joint Conf. Automated Reasoning — IJCAR 2006*, volume 4130 of *LNAI*. Springer, 2006.

Table des matières

I Introduction 3

1 De la question des fondements à l'architecture logicielle 5
1.1 Les règles du jeu mathématiques . . . 5
1.2 De vraies machines . . . 6
1.3 Les deux rôles de l'ordinateur . . . 7
1.4 Vérités nouvelles . . . 8
1.5 De nouvelles preuves . . . 9
1.6 Les règles du jeu informatique . . . 10
1.7 Un choix d'Architecture . . . 11
1.8 Des Mathématiques comme une science expérimentale . . . 11
1.9 Organisation du mémoire . . . 11
1.10 Le reste du dossier HDR . . . 12

2 Rappels sur la Théorie des Types 13
2.1 Les types dépendants : une implémentation des règles logiques . . . 13
2.1.1 Sortes . . . 13
2.1.2 Imprédicativité . . . 14
2.1.3 Les sortes de Coq . . . 14
2.2 Description sommaire du système . . . 14
2.2.1 le PTS dans CCI . . . 14
2.2.2 Les règles . . . 15
2.2.3 Types inductifs . . . 15
2.3 Élimination forte et imprédicativité . . . 18

II Preuves calculatoires formelles 19

3 Traitement des certificats de Pocklington en Théorie des Types 21
3.1 Preuves de primalité élémentaires . . . 21
3.1.1 La méthode de l'instituteur . . . 21
3.1.2 Le programmeur BASIC . . . 22
3.2 Calculer en autarcie ou accepter des certificats . . . 22
3.3 Les certificats de Pocklington . . . 23
3.3.1 Le théorème . . . 23
3.3.2 Les certificats . . . 24
3.4 La vérification des certificats . . . 25
3.4.1 Vérification des conditions calculatoires . . . 25
3.4.2 Définitions des opérations arithmétiques en Coq . . . 26
3.4.3 Amélioration récentes . . . 27
3.5 La construction de certificats . . . 27

3.5.1 Construction de certificats pour des nombres premiers arbitraires 28
3.5.2 Construction de certificats pour les nombres de Mersenne 28
3.5.3 Aller à la limite 29
3.6 Performances 29
3.7 Développements récents 32

4 Le théorème des quatre couleurs 33
4.1 Historique 33
4.2 La preuve fausse de Kempe 34
4.2.1 Triangulation 35
4.2.2 La formule d'Euler 35
4.2.3 Le sommet de degré quatre 36
4.2.4 Le sommet de degré cinq 37
4.3 La contribution de Tait 39
4.4 Recollement 41
4.5 Réductibilité 41
4.6 Les preuves modernes 42
4.7 Réductibilité en Coq 43
4.7.1 Représentation des ensembles $\mathcal{C}_i$ 43
4.7.2 Algorithme naïf 44
4.7.3 Algorithme habituel 44
4.7.4 Algorithme optimisé 45
4.8 Quelques remarques 46

III À propos du Calcul des Constructions Inductives 47

5 Consistences relatives entre Théorie des Types et Théorie des Ensembles 49
5.1 Sémantique ensembliste de Coq 49
5.1.1 Le résultat de Reynolds 49
5.1.2 Cardinaux inaccessibles 50
5.1.3 Caractérisation syntaxique des preuves 50
5.1.4 Notations 51
5.1.5 Interprétation du fragment PTS 51
5.1.6 Interprétation des types inductifs 52
5.1.7 Correction du modèle 52
5.1.8 Axiomes justifiés par le modèle 53
5.1.9 Le tiers-exclu 53
5.1.10 Axiomes du choix 54
5.2 Coder les ensembles comme des types 55
5.2.1 Les ensembles 55
5.2.2 Les propositions 56
5.2.3 Comparaison avec l'approche d'Aczel 56
5.2.4 Les autres constructions explicites 57
5.2.5 Constructions non-calculatoires : remplacement et choix 57
5.2.6 Cardinaux inaccessibles 58
5.3 Conclusion 59
5.4 Inclure la *proof-irrelevance* dans le formalisme 60

6 Difficultés avec `Set` imprédicatif **61**
6.1 Le résultat de Chicli, Pottier et Simpson . . . 61
6.2 Une variante basée sur l'opérateur J de Girard . . . 62
6.2.1 Rappel : dans le Système F . . . 62
6.2.2 En théorie des types . . . 63
6.3 Une (autre) version de la construction de Diaconescu . . . 63
6.3.1 Codage de la paire non-ordonnée . . . 64
6.3.2 Construction de Diaconescu . . . 64
6.4 Conclusion . . . 64

IV Conclusion **65**

7 Conclusion et Perspectives **67**
7.1 Développements et avancées récentes . . . 67
7.2 Développements possibles du système de preuve . . . 69
7.3 Directions pour de nouvelles formalisations . . . 70
7.4 Un système aussi sûr que possible . . . 72