Preuves Formelles Calculatoires

Benjamin WERNER

LIX – Ecole polytechnique

Equipe Partout

Journées X-UPS 2026

Chapitre 1

De la question des fondements à l'architecture logicielle

Dans ce document, je décris deux preuves, toutes deux formalisées en théorie des types à travers Rocq mais très différentes, mais qui font toutes deux intervenir le calcul de manière cruciale.

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 ici 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 de manière aussi complète que possible 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, conjointement à 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

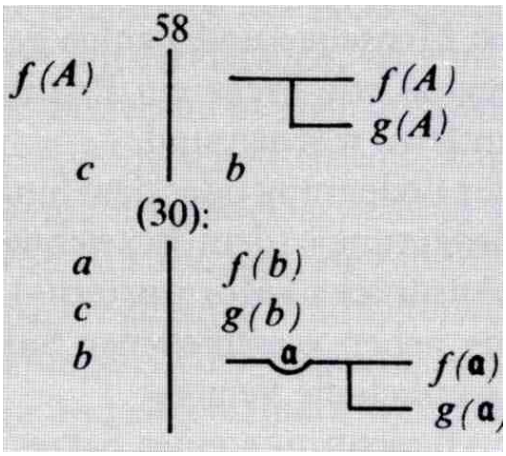


FIGURE 1.1 – Un détail de la *Begrifsschrift* (écriture mathématique) de Frege (1872).

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.

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.

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. Tel que cité par Jean-Yves Girard dans *La mouche dans la bouteille*[18]

FIGURE 1.2 – N. G. de Bruijn

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 [14], dans les années 1960.

Avec Automath arrivent les premières preuves formelles non triviales, en particulier la célèbre formalisation des *Grundlagen der Analysis* de Landau [36], 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.

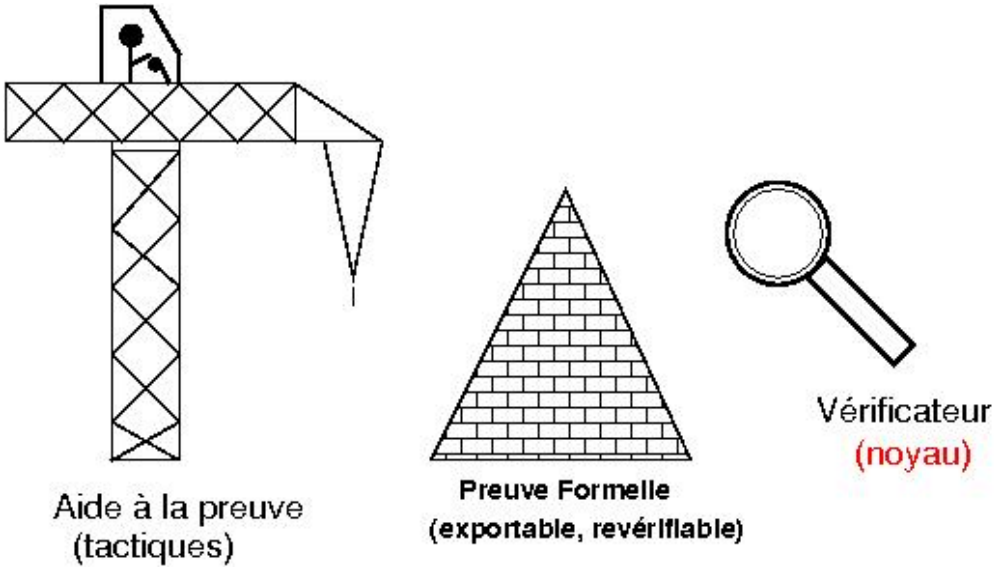
— De Bruijn a donc bien compris l'intérêt de considérer les preuves comme des objets du formalisme. Surtout, il considère cette caractéristique du point de vue de l'architecture logicielle : la syntaxe des preuves étant précisément définie, seule la partie du logiciel qui vérifie la correction des preuves 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*.

Last but not least, si ce qui précède était vrai avant le récent essor de l'intelligence artificielle, celui-ci pourrait rendre encore plus important le rôle de vérificateurs de preuves formels et sûrs. En effet :

— Les IAs font des progrès rapides, y compris dans la production de preuves.

— On peut penser que ces progrès seront plus importants lorsque l'on saura concevoir des IAs spécialisées dans la construction de preuves.

— Ce posera alors la question de la confiance qu'on peut accorder aux preuves construites par des IAs. On connait, par exemple, la possibilité de voir des IAs "halluciner".
— Forcer les IAs à construire des preuves formelles, vérifiées formellement permet évidemment d'éviter cet écueil des vérités alternatives mathématiques. Qui plus est, on peut penser que construire des IAs mathématiques qui interagissent directement avec le vérificateur, y compris pendant la construction de la preuve, peut ouvrir de nouvelles architectures prometteuses.

1.4 Vérités nouvelles

Dès avant ces IAs, l'aide que l'ordinateur pouvait apporter au mathématicien ne se limitait évidemment pas aux preuves formelles. De par sa puissance de calcul et sa capacité de stockage, il a permis 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* [51], 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 2024 on a établit qu'un nombre de 41.024.320 chiffres, $2^{136279841}-1$, était premier ; et on était déjà à plus de 2 millions de chiffres en 1999 et près de mille chiffres dès 1957. La raison première de ces 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

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).

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 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 $2 + 2 \rhd 4$. La relation d'équivalence $=_\beta$ de la règle de conversion étant simplement la clôture congruente, transitive, réflexive et symétrique de $\rhd$. 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).

Chapitre 2

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 [4]. 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.

2.1 Preuves de primalité élémentaires

2.1.1 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.

2.1.2 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 Rocq. 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 Rocq. 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 Rocq. 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.

2.2 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 [3]. 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 [28] ; 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 [10]. Dans notre travail, nous avons pu ré-utiliser en particulier la preuve du théorème de Pocklington proprement dit.

2.3 Les certificats de Pocklington

2.3.1 Le théorème

Le théorème de Pocklington [42] 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 :*

$$p_1 \ldots p_k \quad \textit{sont} \quad \textit{premiers} \tag{2.0}$$

$$(p_1^{\alpha_1} \ldots p_k^{\alpha_k}) \quad | \quad (n-1) \tag{2.1}$$

$$a^{n-1} \quad = \quad 1(\mathsf{mod}\ n) \tag{2.2}$$

$$\forall i \in \{1, \ldots, k\}\ \mathsf{gcd}(a^{\frac{n-1}{p_i}} - 1, n) \quad = \quad 1 \tag{2.3}$$

$$p_1^{\alpha_1} \ldots p_k^{\alpha_k} \quad > \quad \sqrt{n}. \tag{2.4}$$

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 Rocq 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. Mais 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 [10]. 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 [8]. 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 Rocq est le suivant :

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

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

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

$$F_1 \text{ est pair, } R_1 \text{ est impair, et } F_1 R_1 = n-1 \quad (2.5)$$
$$(F_1+1)(2F_1^2 + (r-1)F_1 + 1) > n \quad (2.6)$$
$$a^{n-1} = 1(\mathsf{mod}\ n) \quad (2.7)$$
$$\forall i \in \{1, \ldots, k\}\ \mathsf{gcd}(a^{\frac{n-1}{p_i}} - 1, n) = 1 \quad (2.8)$$
$$r^2 - 8s \text{ n'est pas un carré ou bien } s = 0. \quad (2.9)$$

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 [8] 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.

2.3.2 Les certificats

Un certificat ne se compose pas uniquement des nombres $a, p_1, \alpha_1, \ldots, p_k, \alpha_k$. Pour pouvoir être auto-suffisant, 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 $c = \{n, a, [c_1^{\alpha_1}; \ldots; c_k^{\alpha_k}]\}$ où les $c_1, \ldots, c_k$ sont respectivement des certificats pour les nombres $p_1, \ldots, p_k$. 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 Rocq comme un type inductif. Un certificat est soit :

— un n-uplet de la forme : $c = \{n, a, [c_1^{\alpha_1}; \ldots; c_k^{\alpha_k}]\}$ [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 :

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

Cela se traduit immédiatement par la définition Rocq suivante :

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

```
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.

2.4 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 :

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

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

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

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 Rocq 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.

2.4.1 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 $a^{n-1} \bmod n$ and $\gcd(a^{\frac{n-1}{p_i}} - 1, n)$ 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 $\gcd(b, n) = \gcd(b \bmod n, n)$. De plus, il est possible de ne calculer qu'un seul pgcd en remarquant que $\gcd(b_1 \ldots b_l, n) = 1$ si et seulement si pour tout $i = 1 \ldots l$, on a $\gcd(b_i, n) = 1$.

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, $l = [q_1; \ldots; q_r]$ et n calcule $a^{q_1 \cdots q_r} \bmod n$.

Une autre optimisation est de partager en partie les calculs des

$$a^{\frac{n-1}{p_1}} \bmod n,\ \ldots,\ a^{\frac{n-1}{p_k}} \bmod n,\ a^{n-1} \bmod n.$$

Soit $m = (n-1)/(p_1 \ldots p_k)$; si ces calculs sont effectués séparément, $a^m \bmod n$ est calculé $k+1$ fois, $(a^m \bmod n)^{p_1} \bmod n$ 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 $[q_1; \ldots; q_r]$, la fonction `all_pow_mod` retourne la paire :

$$(P \prod_{1 \le i \le r} A^{\frac{q_1 \cdots q_r}{q_i}} \bmod n, A^{q_1 \cdots q_r} \bmod n).$$

On peut alors remarquer que l'application de cette fonction à $P = 1$, $A = a^m \bmod n$ et $l = [p_1; \ldots; p_k]$ 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.
```

2.4.2 Définitions des opérations arithmétiques en Rocq

La représentation des nombres est évidemment cruciale pour l'efficacité des calculs. Dans le langage purement fonctionnel de Rocq, 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 forte du calcul est décrite par ailleurs. Elle impose parfois certaines contorsions pour arriver à écrire des versions efficaces des algorithmes recherchés.

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

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 Rocq. 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.

2.4.3 Améliorations du traitement numérique

Dès peu après ce travail, plusieurs progrès importants ont été fait pour ce qui est de la représentation des nombres en Rocq. Par exemple :

- D'une part, Benjamin Grégoire et Laurent Théry [25] 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 [46] a implémenté une version de Rocq 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 Rocq et donc des nombres dont on peut prouver la primalité en Rocq. Nous revenons sur ces points à la fin de ce chapitre.

2.5 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 [52] et GMP [48]. Étant donné un nombre n, ce programme engendre un fichier en syntaxe Rocq 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 2.1.2, 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 Rocq : 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 Rocq de calculer $C((n, a, d, \mathsf{sqrt}) :: l)$ est cruciale en particulier pour le temps de vérification.

2.5.1 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.

2.5.2 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 2024) est $2^{136.279.841} - 1$ dont l'écriture comporte 41.024.320 chiffres décimaux...

Le théorème de Pocklington est bien adapté aux nombres de Mersenne. Une première remarque est qu'un nombre de Mersenne s'écrit $1111\ldots1111$ 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.

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 un nombre premier qui est un facteur du prédécesseur. 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 $n = 2281$).

2.5.3 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 Rocq 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 Rocq 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.

Le développement décrit ici a permis de vérifier en Rocq la primalité d'un nombre premier "aléatoire" de 200 chiffres :

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

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 2.1 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).

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

#	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

FIGURE 2.1 – Temps de vérification des nombres de Mersenne

2.6 Développements postérieurs

Le travail décrit dans ce chapitre a depuis été dépassé, mais d'une manière qui valide le principe de son approche : en continuant à importer en Rocq les outils scientifiques et techniques plus récents permettant d'établir la primalité de nombres. En particulier :

— Les outils mathématiques : Laurent Théry a, avec Guillaume Hanrot, formalisé en Rocq les théorèmes sur les courbes elliptiques et la primalité.

— La représentation des grands nombres en Rocq avec la librairie de Benjamin Grégoire et Laurent Théry.

Chapitre 3

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.

3.1 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 3.1. 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 [30].
1880 Variante de la preuve par Tait [47].

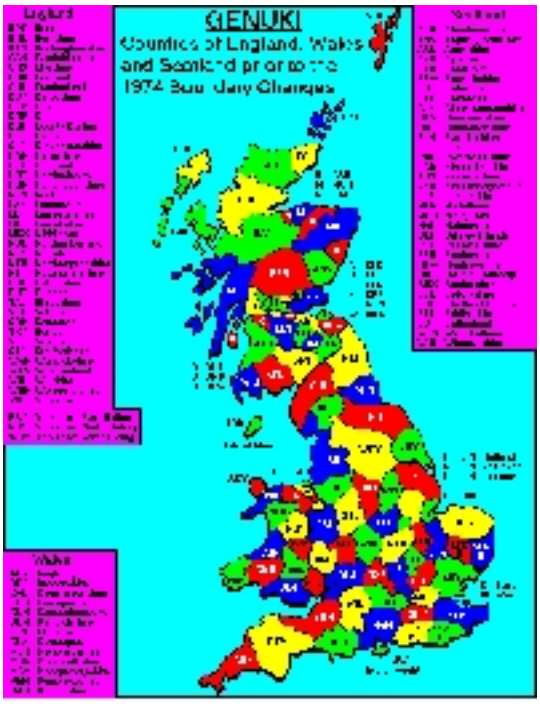


FIGURE 3.1 – La carte des Contés Britanniques avant 1974 (avec l'autorisation de GENUKI).

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* [6]. 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 [29].

1976 Kenneth Appel et Wolfang Haken annoncent avoir vérifié le théorème [31, 32, 33]. 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 [44]. 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.

3.2 La preuve fausse de Kempe

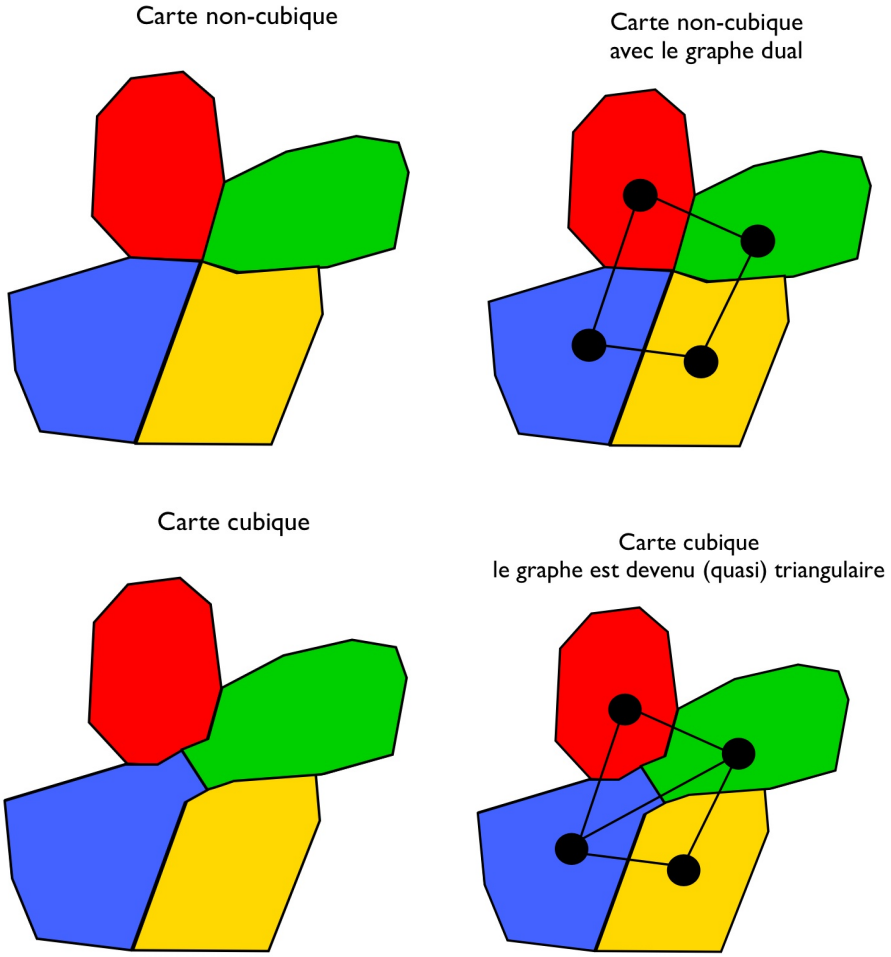
La preuve fausse de Kempe est importante car elle comporte deux idées également essentielles aux preuves "modernes". D'une part qu'il faille trouver le "point faible" du graphe, à savoir un endroit où la densité d'arêtes est faible, et surtout l'idée des permutations de couleurs dans une composante connexe.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

3.2.1 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.



3.2.2 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.

3.2.3 Le sommet de degré quatre

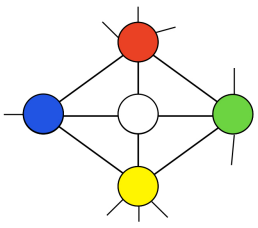
Plus précisément, le seul cas problématique est celui où le coloriage obtenu par hypothèse de récurrence assigne quatre couleurs distinctes aux quatre voisins du sommet considéré :

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 :*

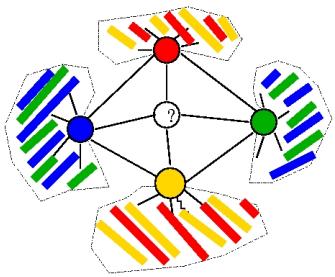
1. *ses sommets sont tous coloriés de deux couleurs, qu'on appellera A et B,*
2. *les sommets du graphes qui sont voisins d'un sommet de la composante, mais ne font pas partie de la composante ne sont pas coloriés par A ou B.*

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



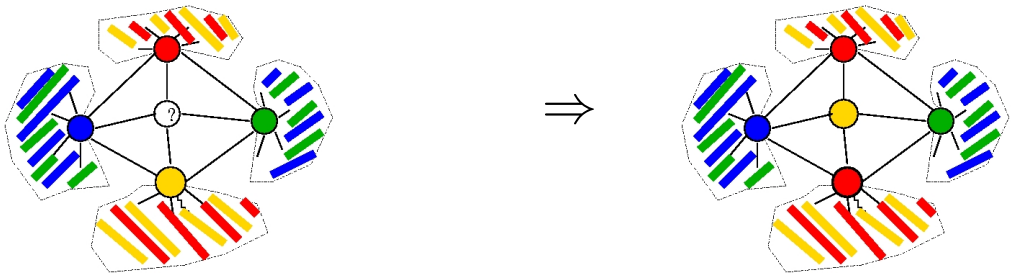
disponibles :

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

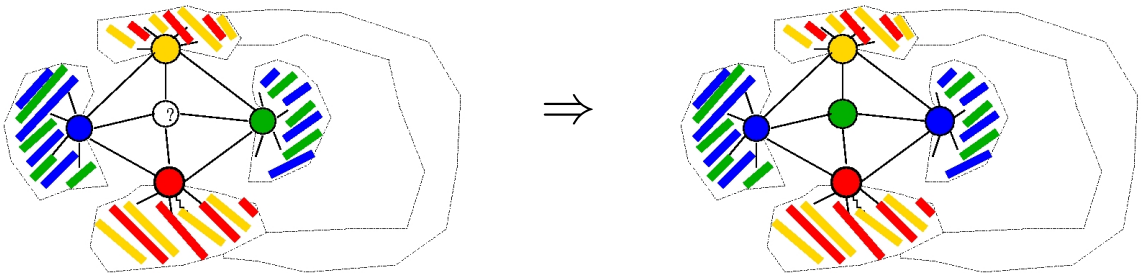


rouge et jaune d'une part, bleu et vert de l'autre.

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 :



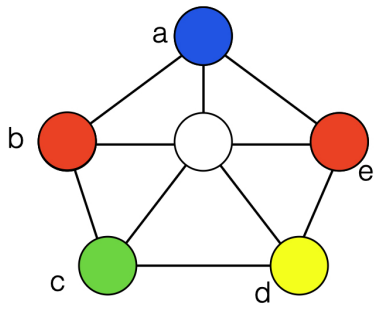
Ceci achève, correctement, le cas du sommet à degré quatre.

3.2.4 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.

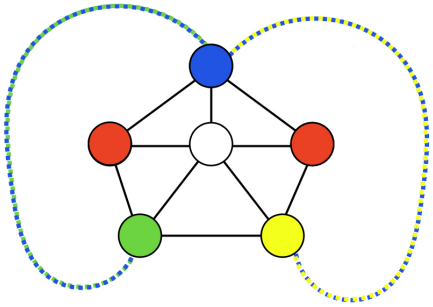
Le seul cas problématique est si les 5 voisins se sont vus affecter les 4 couleurs disponibles. Par symétrie,

il est possible de se ramener au cas de figure suivant.

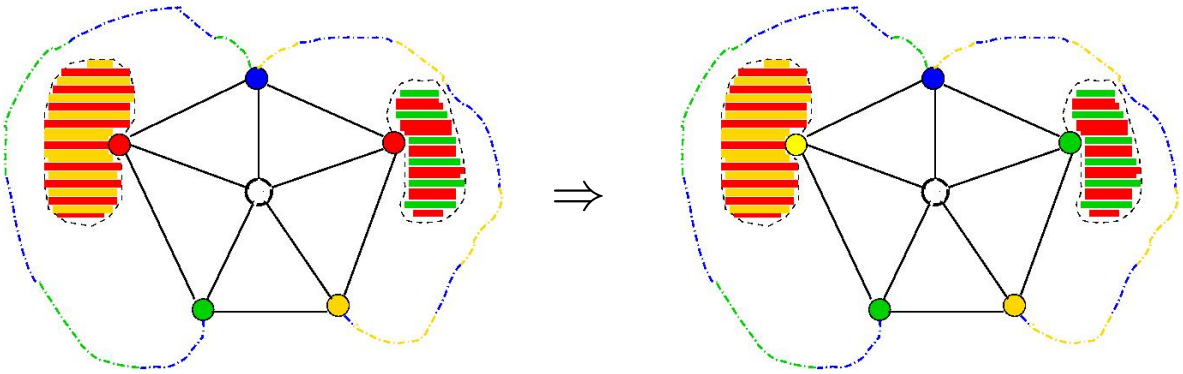


Kempe dit alors :

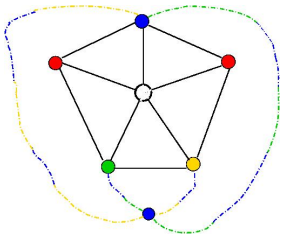
1. Si a et c ne sont pas dans la même composante bleue/verte, on inverse les couleurs dans l'une des deux et on libère la couleur correspondante pour le sommet central.
2. Sinon, si a et d ne sont pas dans la même composante bleue/jaune, on fait de même.
3. reste alors le cas correspondant au dessin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



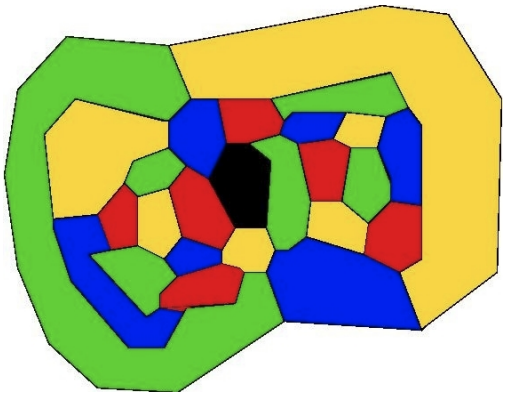
Kempe pense alors que la composante rouge/verte contenant e est disjointe de la composante rouge/jaune contenant b. Si c'est le cas, on peut en effet inverser ces deux composantes et libérer ainsi la couleur bleu pour le sommet central.



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 :



3.3 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 3.3.

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 3.3.

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.

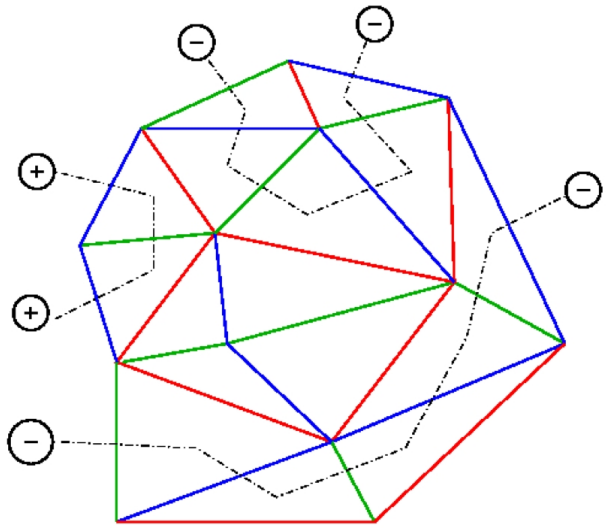


FIGURE 3.2 – Appareillement signé entre arêtes bleues et vertes induit par un trois-coloriage.

— 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.

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 3.3 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 _.

3.4 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.

3.5 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).*

3.6 Les preuves modernes

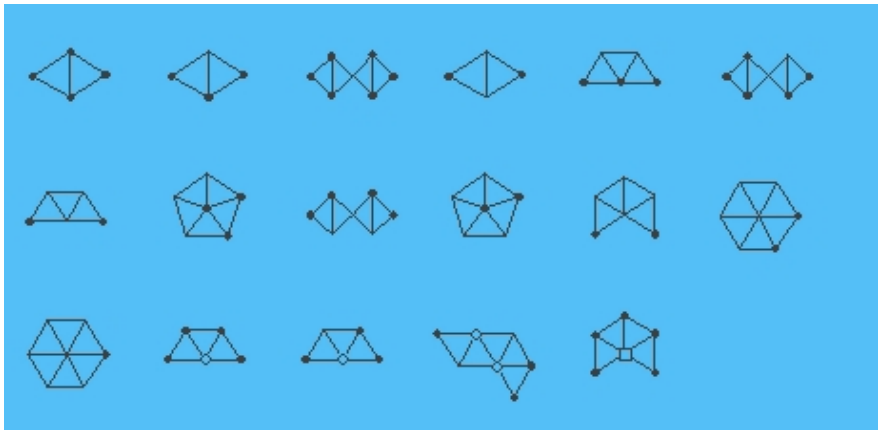
Avec la notion de réductibilité, on a la clé des preuves modernes (et correctes) du théorème des quatre couleurs : on va exhiber un ensemble de quasi-triangulations qu'on appellera *configurations* tel que :

— 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.

3.7 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.

3.7.1 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.

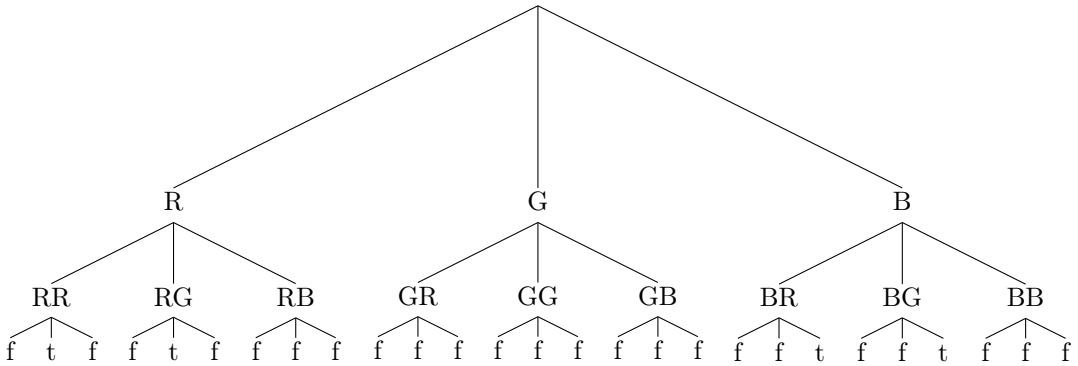
Les coloriages sont naturellement représentés comme des listes de caractères dans l'alphabet à trois lettres. Comme on est dans un cadre purement fonctionnel, on représente les ensembles de telles listes à l'aide d'arbres ternaires :

— 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.

3.7.2 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.

3.7.3 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.

3.7.4 Algorithme optimisé

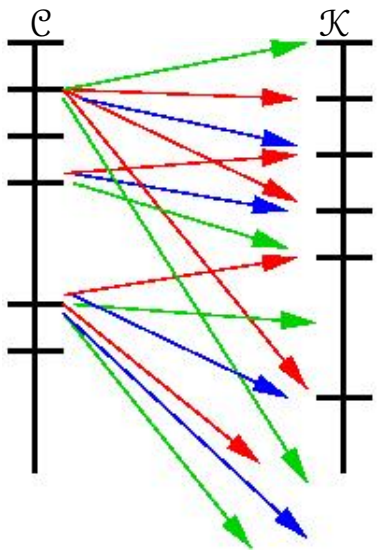
L'algorithme décrit ci-dessus et qui est utilisé, par exemple par Robertson et ses co-auteurs. C'est donc lui qui est (très minutieusement) décrit dans les écrits accompagnant leur article.

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 $w \simeq_p c$:



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.

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

3.8 Hypercartes

Je n'ai pas la possibilité de détailler ici toutes les subtilités et tout l'intérêt de ce point, mais je veux le mentionner, car c'est, à mon sens, un bel exemple de comment l'élégance mathématique peut contribuer à rendre les preuves formelles plus simples et plus tractables.

Dans la preuve des quatre couleurs, étant donné un graphe planaire (souvent triangulé ou quasi-triangulé), on doit souvent considéré le même graphe où l'on aura enlevé (parfois ajouté) une arête ou un sommet.

Le graphe obtenu en enlevant ou ajoutant un élément, partage un certain nombre de propriétés avec le graphe d'origine. Par exemple un coloriage du graphe de départ peut être appliqué au graphe réduit, lorsque l'on enlève un sommet et deux arêtes sur le bord d'une quasi-triangulation.

Ces propriétés sont souvent évidentes intuitivement, mais peuvent être compliquées à traiter formellement. On cherche donc une représentation des graphes où les transformations comme ôter ou ajouter un élément sont définies de la manière la plus simple possible. Au cours du travail de formalisation, Georges Gonthier a remarqué que ces définitions étaient plus simples lorsque l'on voyait les graphes planaires comme des cas particuliers d'une structure plus riche. Cette structure est connue sous le nom d'*hypercartes*, mais il est remarquable que Georges l'a redécouverte par lui-même dans le seul but de simplifier la formalisation.

Définition 4.3 *Une hypercarte est un ensemble B (les brins) muni de trois permutations s, f, a, avec la propriété suivante :*

$$a \circ s \circ f = Id.$$

L'idée est la suivante ; dans le cas des hypercartes correspondant à des graphes planaires :

- Les brins peuvent être vus comme des moitiés d'arêtes (un graphe planaire non-orienté est transformé en hypercarte en partageant chaque arête en deux brins de directions opposées, comme les deux voies d'une route).
- La permutation s passe d'un brin au brin suivant, partant d'un même sommet, en tournant (par exemple) dans le sens horaire. Les sommets sont donc les orbites de s.
- La permutation f passe d'un brin au brin suivant en tournant à droite. Les faces sont donc les orbites de f. Dans le cas d'une triangulation, ces orbites sont donc de cardinal 3/
- La permutation a passe d'un brin au brin de direction opposée. Les arêtes sont donc les orbites de a (qui sont de cardinal 2).

On peut remarquer que l'équation $a \circ s \circ f = \mathrm{Id}$ suffit à garantir que a, s et f sont des permutations. Également, la donnée de deux de ces applications contraint la définition de la troisième.

La formule d'Euler devient, pour les hypercartes :

$$A + S + F = B + 2C$$

où A, S, F sont les nombres respectifs d'arêtes, de sommets et de faces, B le nombre de brins, et C le nombre de composantes connexes pour $a \cup s \cup f$. On peut même utiliser cette formule pour définir ce qu'est une hypercarte planaire. Il est alors possible d'énoncer et de prouver une version purement combinatoire du théorème de Jordan. Ce dernier est crucial pour la preuve, car il permet de partitionner une triangulation en deux quasi-triangulations.

Je laisse le lecteur intéressé regarder les détails, par exemple comment est énoncé ce théorème de Jordan combinatoire, ou d'autres questions comme quelles transformations sur les applications s, f, a correspondent à la suppression d'une arête ou d'un sommet. Voir [20, 19].

Si je mentionne ce point, c'est aussi parce qu'il illustre un point que je trouve important : souvent, pour qu'une preuve compliquée puisse être formalisée, il est utile, voire vital, de trouver les "bonnes" définitions et formulations.

*En d'autres termes, lorsque l'on formalise, l'*élégance mathématique *n'est plus seulement une qualité esthétique gratuite : c'est elle qui aide, voire est nécessaire, à rendre les preuves formelles faisables.*

3.9 Autre aspect

Je passe ici sous silence bien d'autres aspects de la preuve formelle, qui sont décrits dans tous les détails par Georges Gonthier [20]. Une autre point important est celui du langage de preuve. La formalisation du théorème des quatre couleurs a amené le développement d'un *dialecte* du langage de tactiques de Rocq appelé SSR pour *small scale reflection*. Ce dialecte promeut un style de preuves qui 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 évitant à chaque fois le recours à des lemmes explicites.

Bibliographie

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

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

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

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

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

[12] Claudio Sacerdoti Coen. A semi-reflexive tactic for (sub-)equational reasoning. In Filliâtre et al. [16], pages 98–114.

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

[14] 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.

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

[16] 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.

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

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

[19] Georges Gonthier. Formal proof - the four- color theorem. *Notices of the AMS*, 55(11), 2008.

[20] Georges Gonthier. A computer-checked proof of the Four Color Theorem. Technical report, Inria, March 2023.

[21] Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry. A modular formalisation of finite group theory. In Schneider and Brandt [45], pages 86–101.

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

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

[24] 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.

[25] 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.

[26] Thomas Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture. *ArXiv.org*, page 21, January 2015.

[27] Thomas C. Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua, and Roland Zumkeller. A revision of the proof of the Kepler Conjecture. 44(1) :1–34, 2010.

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

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

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

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

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

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

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

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

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

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

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

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

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

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

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

[49] 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`.

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

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

[52] 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`.

[53] 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.