Logique, complexité, démonstration automatique et thèmes connexes

— Mémoire d'habilitation à diriger les recherches —

Jean Goubault-Larrecq

23 juin 1997

Chapitre 1

Introduction

Mon travail de recherche a commencé lorsque je suis entré au centre de recherches de Bull S.A. en septembre 1989, dans le projet de spécification et de validation formelle du logiciel, sous la direction de Patrick BEHM et de Gérard MEMMI, puis de Dominique BOLIGNANO et François ANCEAU. Le but du projet était de mettre au point une méthodologie de développement du logiciel, à usage des équipes de développement en interne. Ce projet impliquait une activité de formation, non seulement des chercheurs, mais en général des développeurs, aux méthodes de spécification (la méthode VDM [Jon90] avait été choisie, notamment pour la relative abondance de documentations disponibles à l'époque). Ce projet incluait surtout de nombreuses activités de recherche, notamment en démonstration automatique ou semi-automatique de théorèmes (nécessaire pour prouver la correction de spécifications, ou de raffinements entre spécifications), et en prototypage de spécifications (utile pour tester rapidement si une spécification décrit grosso modo ce qui était voulu).

1.1 Démonstration automatique

En ce qui concerne la démonstration de théorèmes, l'axe de recherche sur lequel je m'étais lancé, et qui devait par la suite constituer l'essentiel de mon travail de thèse sous la direction de Patrick COUSOT [Gou93b], était la démonstration automatique en logique classique du premier ordre. Ce choix était dû aux considérations suivantes. Premièrement, les langages de la famille de VDM/SL ou Z sont essentiellement formés d'un cœur – une théorie des ensembles typée, essentiellement – et de sucre syntaxique autour. La logique employée est classique (en Z), ou recodable en logique classique relativement aisément (la logique de VDM est la logique à trois valeurs de KLEENE). De plus, même si le cœur logique est d'ordre supérieur – une théorie des ensembles typée est essentiellement la même chose qu'une logique d'ordre supérieur – la plupart des raisonnements à effectuer en pratique sont des raisonnements d'ordre 1, ou des récurrences simples. L'importance de la logique d'ordre 1 s'est trouvée réaffirmée lorsque Patrick BEHM a cédé la direction de l'équipe à Dominique BOLIGNANO en 1990. L'intérêt de l'équipe se portait alors vers diverses méthodes de vérification de logiciels séquentiels ou parallèles, un bon nombre pouvant bénéficier de progrès en matière de techniques de preuve, notamment en logique classique du premier ordre.

À cause de la difficulté inhérente à la découverte automatique de preuves en logique classique du premier ordre, ou dans des logiques plus expressives, il était plus raisonnable d'adopter une approche de recherche de preuves assistée par un être humain. Mais l'écriture de preuves pleinement formelles est un travail relativement pénible et exigeant pour un être humain, de sorte que la question suivante est d'une grande importance pour l'utilisabilité des méthodes formelles : est-il possible de demander à l'utilisateur humain de décrire la preuve qu'il a en tête uniquement dans ses grandes lignes, et de laisser la machine vérifier l'argumentation, en complétant les maillons manquant de la preuve formelle ?

Précisons cette idée. Comme il est déjà suggéré au début de [Bou90] (p. E I.15), une preuve, telle que présentée dans un livre de mathématiques par exemple, n'est pas une preuve formelle, mais une description concise qu'un esprit éduqué doit pouvoir expanser en une véritable preuve formelle, dans un système d'axiomes convenu. Vérifier une telle "preuve" demande de vérifier chaque passage d'une ligne à l'autre, ce qui doit normalement être suffisamment évident. Mon approche a donc consisté à tenter de répondre à la double question : qu'est-ce que l'évidence ? peut-on trouver rapidement, par algorithme, une preuve d'un théorème évident, ou conclure à sa non-évidence ? (Un énoncé non prouvable étant assimilé à un énoncé infiniment non évident.)

Cette double question, dont l'étude a déjà été commencée dans ma thèse, a fait l'objet des articles que je présente en chapitre 2, où la question est abordée sous l'angle de la théorie de la complexité. L'étude de méthodes de preuve automatique a fait l'objet de publications dont la présentation est regroupée dans le chapitre 3. J'y parle aussi des diverses directions de recherche que j'ai explorées par la suite, notamment dans l'étude de la preuve en logique du temps linéaire – un autre langage logique bien adapté à la preuve de spécifications, notamment de protocoles distribués – ou de l'étude de restrictions praticables de la logique d'ordre supérieur, notamment par l'utilisation de la notion de ramification due à Bertrand RUSSELL au début du siècle, recherches que j'ai menées lors de ma demi-année sabbatique à l'université de Karlsruhe en 1996, aupès du professeur Peter H. SCHMITT, à l'institut de logique, complexité et systèmes de déduction.

1.2 Langages fonctionnels

Sur la suggestion de Gérard MEMMI, j'avais examiné en 1989 la possibilité d'écrire un programme permettant d'exécuter des spécifications VDM, dans un but de prototypage. Lorsque l'on écrit une spécification, au même titre que lors de l'écriture d'un programme, il se peut que l'on écrive tout autre chose que ce que l'on avait l'intention d'écrire. Il est alors nécessaire de déboguer la spécification. Bien que les langages de spécification comme VDM/SL permettent d'écrire des spécifications non exécutables, il est intéressant de pouvoir tester celles qui seraient suffisamment simple pour être interprétables. Un sous-langage naturellement exécutable de VDM/SL est alors un langage fonctionnel sans effet de bord, avec des types de données et des opérations ensemblistes (sur des ensembles finis).

Pour réaliser un interprète de ce langage, il a fallu examiner en détail la sémantique de VDM/SL, ce qui a mené l'équipe à réaliser que la version de l'époque de cette sémantique laissait à désirer, tant du point de vue de la simplicité que de celui de la cohérence. Ceci a abouti à un effort de ma part pour définir une sémantique renouant avec la tradition considérant VDM/SL comme un habillage de la théorie des ensembles de ZERMELO-FRÆNKEL [Gou91], dans le but de fixer la définition du langage, au moins à usage interne chez Bull. L'expertise que j'avais acquise m'a ensuite permis de participer à la mise au point de la sémantique officielle de VDM/SL, telle que définie par Peter Gorm LARSEN pour la normalisation par le BSI et par l'ISO du langage [Lar92]. Mon entrée au comité de normalisation est due à Cliff B. JONES, sur recommandation d'A. Jeremy J. DICK.

Néanmoins, la retombée la plus intéressante de cet effort initial est ma réalisation d'un langage dérivé de Standard ML, étendu par des types et des primitives ensemblistes, ainsi que des constructions syntaxiques adéquates. (J'ai appelé ce langage HimML – le ciel, ou le paradis, en allemand.) L'intérêt d'une telle réalisation est l'efficacité des opérations ensemblistes ainsi codées, qui dépassent d'assez loin ce que l'on pourrait obtenir par leur codage comme fonctions de bibliothèques. La réalisation repose sur une utilisation générale du hash-consing, et requiert une gestion spéciale de la mémoire à l'exécution.

L'efficacité de l'interprète HimML, dont la réalisation a commencé en décembre 1991, et dont une première version fonctionnait en mars 1992, a dépassé mes espérances, au point qu'il devenait utilisable non seulement comme langage de prototypage, mais en fait comme langage de programmation à part entière. C'est ainsi qu'il me fut possible d'utiliser tout le confort d'une programmation ensembliste pour tester des idées d'algorithmes difficiles à réaliser autrement. Le démonstrateur automatique de la section 3.2, par exemple, demanderait des efforts colossaux à programmer en C, et poserait même de nombreuses difficultés à programmer en ML (certaines opérations sur des ensembles d'ensembles, notamment). Ce langage m'a aussi permis de disposer d'une liberté de pensée suffisante pour oser des techniques algorithmiques auxquelles on renonce usuellement, par crainte d'inefficacité; l'interprète m'a finalement permis de coder et de mettre au point rapidement des algorithmes de preuve relativement compliqués.

L'interaction entre une activité dans le domaine des langages fonctionnels (à la ML, ici) et celui de la logique et de la preuve ne s'arrête pas à une amélioration du confort algorithmique. Chacun des deux domaines profite en effet à l'autre. D'un côté, le λ-calcul, prototype des langages fonctionnels, fournit l'un des codages les plus simples de la théorie des types, ou logique d'ordre supérieur. D'un autre côté, la compréhension de mécanismes tels que le typage ou l'optimisation de programmes en termes de dérivation et d'exploitation de propriétés logiques desdits programmes, apporte une vue claire et rigoureuse du typage, ou de la transformation de programmes.

Le chapitre 4 regroupe les présentations de mes publications dans le domaine des langages fonctionnels. Ces publications sont moins centrées autour d'un thème comme en preuve de théorèmes, mais sont davantage une suite d'applications des idées que j'avais développées pour attaquer les problèmes précédents. En section 3.3, on trouvera a contrario la description d'une idée en démonstration automatique, dont les intuitions majeures proviennent de l'étude des langages fonctionnels.

1.3 Logique modale S4

Finalement, j'expliquerai dans le chapitre 5 les tenants et les aboutissants d'un travail que j'ai commencé en 1994 indépendamment de mon travail chez Bull, que j'ai pu poursuivre lors de mon séjour à Karlsruhe en 1996, et qui porte sur l'interprétation calculatoire des preuves dans la logique modale S4, à la Curry-Howard. Ce travail est de nature plus mathématique qu'informatique, mais a de nombreuses implications tant en logique qu'en informatique.

D'un point de vue logique, il s'agit d'une étude fine du processus d'élimination des coupures en S4; d'un point de vue informatique, il s'agit d'une définition propre de mécanismes à la eval et quote de Lisp, ou plus précisément de mécanismes permettant de, en un certain sens, geler un morceau de programme (quote) pour le dégeler ensuite (eval). Un autre intérêt de l'étude est que S4 est formellement similaire à la logique linéaire, qui a suscité un engouement certain en informatique ces dernières années, mais que S4 est plus simple à étudier, dans la mesure où l'on n'a pas à se préoccuper de la gestion des ressources propre à la logique linéaire – cette gestion des ressources peut ensuite être ajoutée par-dessus un calcul obtenu pour S4, sous forme de contraintes.

Mais le point à mon avis le plus intéressant, et aussi le plus troublant, est que dans mon effort pour obtenir un langage qui ait un réel contenu calculatoire, et qui pour des raisons que j'expliquerai dans le chapitre 5 utilise des calculs de substitutions explicites, j'ai obtenu un langage qui, modulo la notion d'égalité induite par la réduction définie dans le calcul, est un complexe simplicial – une structure fondamentale de la topologie algébrique. J'ai de nombreuses questions en suspens, et assez peu de réponses pour le moment, sur les propriétés de ce complexe simplicial, et surtout sur la signification informatique de l'apparition même de cette structure dans un langage informatique.

Chapitre 2

Complexité de la recherche de preuve

2.1 Complexité de l'évidence

Il s'agit de ma première publication dans une conférence internationale [Gou94b]. Dans cet article, je suggère que l'on peut quantifier la complexité des problèmes de déduction automatique (un problème indécidable) d'une façon plus fine, en définissant d'abord des notions acceptables de *difficulté* ou de *non-évidence* d'un énoncé logique; et ensuite, en examinant la complexité non de la question « la formule donnée en entrée est-elle prouvable ? » mais de la question « est-elle évidente ? »

Plus précisément, la non-évidence d'un énoncé logique est donnée par une *mesure* m, c'est-à-dire par une fonction prenant une formule de la logique considérée, et à valeurs dans $\mathbb{N} \cup \{+\infty\}$. Intuitivement, si Φ est une formule, Φ est trivialement prouvable si $m(\Phi) = 0$, facilement prouvable, ou raisonnablement évidente si $m(\Phi)$ est un petit entier; Φ est un résultat difficile si $m(\Phi)$ est grand; et par extension, un résultat improuvable est un résultat infiniment difficile à prouver, dont on conviendra que la mesure doit être infinie. La question « la formule d'entrée est-elle évidente ? » prend alors un sens sous la forme du problème de décision suivant, pour chaque entier k :

ENTRÉE : une formule Φ de la logique considérée, de taille n.

QUESTION : $m(\Phi)$ est-elle inférieure ou égale à k ?

Ce problème, lorsqu'il est décidable, peut alors être analysé en cherchant la classe de complexité naturelle à laquelle il appartient. J'appelle désormais ce problème le *problème de l'évidence*.

L'article [Gou94b] s'intéresse au cas des formules de la logique classique du premier ordre, sans symboles interprétés, ou avec des symboles interprétés dans une théorie équationnelle, cas déjà considéré dans ma thèse [Gou93b], soutenue un mois après la soumission de l'article. Il s'agissait d'une première étape, avant d'attaquer des logiques plus expressives; des difficultés se présentent en fait dès que l'on considère le cas de l'égalité, ce dont je parlerai en section 2.2.

2.1.1 Discussion de l'approche

Quelques remarques sur cette approche sont nécessaires. Premièrement, toute mesure m correspondant à nos critères informels a la propriété que Φ est prouvable si et seulement si $m(\Phi)$ est finie. En particulier, dès que la logique qui nous intéresse est indécidable, la fonction m est incalculable; c'est certainement le cas pour la logique du premier ordre. En conséquence, contrairement à une mesure de la formule en entrée comme la taille ou le nombre de symboles, il ne s'agit pas d'une mesure de Blum, ce qui rend le problème de l'analyse de la complexité du problème ci-dessus a priori plus délicat, les résultats classiques de la théorie de la complexité n'étant plus nécessairement valides. L'astuce consistant à étudier le problème modifié suivant permet de circonvenir cet inconvénient :

ENTRÉE : une formule Φ de la logique considérée, de taille n, et un entier k, codé en unaire.

QUESTION : $m(\Phi)$ est-il inférieur ou égal à k ?

La taille de l'entrée est alors en effet celle de Φ, soit n, plus celle de k en unaire, soit k, ce qui est maintenant une mesure de BLUM, et change relativement peu de choses au problème.

Une seconde remarque est une réponse à l'objection suivante : comment se fait-il qu'un énoncé trivialement faux se trouve qualifié d'infiniment non évident par une mesure m comme ci-dessus ? Le paradoxe repose sur le fait que nous confondons ici deux notions d'évidence, celle de savoir si le fait que Φ soit prouvable est évident; et celle de savoir si le fait de savoir si Φ est vrai (prouvable) ou faux (incohérent) est évident. En logique classique, et en prenant comme entrée une formule close Φ, la première notion d'évidence correspond à $m(\Phi)$, tandis que la seconde est plus fidèlement représentée par une mesure telle que $\min(m(\Phi), m(\neg\Phi))$. Ainsi, une notion d'évidence n'exclut pas l'autre. L'inconvénient de la seconde est qu'elle symétrise arbitrairement le problème ci-dessus, et fournit finalement moins d'information. Par exemple, si le problème de la prouvabilité à $m(\Phi)$ borné est NP-complet dans une logique – donc un problème d'*existence* d'un objet de taille polynomial – le problème du « vrai ou faux » à $\min(m(\Phi), m(\neg\Phi))$ borné n'est ni particulièrement existentiel ou universel, mais les deux. (En l'occurrence, il sera en général NP-équivalent, et en particulier à la fois NP-difficile et coNP-difficile.)

Une autre critique que l'on peut faire de l'idée de construire une mesure telle que m est que le concept, tel que nous l'avons présenté, est quasiment vide. L'existence de telles mesures est effectivement triviale. Par exemple, on peut définir m comme la fonction qui à toute formule prouvable associe 0, et aux autres associe $+\infty$; ceci ne peut alors rien nous apprendre de plus que ce que nous savions déjà : que la prouvabilité des formules classiques du premier ordre est indécidable et semi-décidable. Le critère qui m'a guidé pour décider de l'intérêt d'une mesure m a été double : d'abord, la mesure m devait être *naturelle*, et correspondre à une notion intuitive de difficulté d'un théorème; il s'agit d'un critère de nature philosophique, et dont l'appréciation ne peut être que subjective. Ensuite, la mesure doit avoir une valeur *opératoire*, elle doit permettre de tirer des conclusions sur la structure du problème de preuve, et ces conclusions doivent pouvoir être confrontées à l'expérience tirée de l'utilisation des différentes méthodes de preuve automatique. En résumé, ce dernier critère est la valeur *empirique* de m.

Ainsi, une meilleure mesure m, parce que plus naturelle, est la suivante. Dans toute logique définie par un formalisme de preuve (ou plusieurs, comme la logique du premier ordre), une mesure possible de Φ est la taille minimale d'une preuve de Φ dans le formalisme considéré, ou $+\infty$ s'il n'y en a pas. Le problème est la valeur opératoire de cette mesure, qui reste assez faible. En effet, par définition, le problème de l'évidence sera dans NP – en supposant que la vérification qu'une dérivation est bien une preuve est faisable en temps polynomial, ce qui est le cas pour la plupart des formalismes connus, en tous cas au premier ordre. Dans la plupart des cas, en fait, le problème sera NP-complet, ce qui ne permettra pas de détecter de différences de complexité entre différents problèmes de preuve. En logique classique du premier ordre, j'ai identifié quatre mesures qui me semblent satisfaisantes, et dont je parlerai en section 2.1.2.

La quatrième et dernière remarque que je ferai sur cette approche a trait à sa nouveauté. L'approche consistant à prendre une logique, et à limiter ou à contrôler l'usage que l'on fait de certaines ressources n'est en effet pas neuve. L'un des exemples les plus connus est certainement la logique linéaire de Jean-Yves GIRARD, dans laquelle les ressources de nombre d'utilisation des hypothèses logiques sont cruciales [Gir87a], et qui connaît de nombreuses applications informatiques.

L'idée d'analyser la complexité d'un problème de preuve en bornant une taille de preuve a aussi été exploitée par Rohit PARIKH [Par73] dans le cadre de l'arithmétique de PEANO du second ordre. La question, dans son cas, n'était pas de caractériser la complexité d'un problème à ressources bornées décidable, mais de savoir si ce problème à ressources bornées était décidable, problème qui à ma connaissance est toujours ouvert.

L'approche que je propose pour analyser la complexité de la recherche de preuves n'est pas non plus la seule. Les travaux d'Elmar Eder [Ede92] notamment attaquent le problème en essayant de comparer l'efficacité de diverses méthodes de preuve (résolution, connections, consolution). Ceci est une approche plus concrète, dans la mesure où elle permet de répondre à la question « tel algorithme est-il plus efficace que tel autre ? » mais ne s'adresse pas au problème de la recherche de preuve indépendamment d'un algorithme.

2.1.2 Résultats et conséquences

Les résultats que je présente dans [Gou94b] sont de deux natures : premièrement, j'identifie quelques mesures de difficultés d'énoncés en logique classique du premier ordre qui me semblent naturelles et intuitives; secondement, je détermine la complexité du problème de savoir si une formule en entrée est de difficulté inférieure ou égale à k.

Les mesures de difficulté que je choisis proviennent du théorème de Herbrand. En supposant que la formule en entrée soit existentielle, c'est-à-dire de la forme $\exists x_1, \ldots, x_p \cdot \Psi$, où Ψ est sans quantificateur, ce théorème s'exprime ainsi :

Théorème 2.1.1 (Herbrand) *Une formule $\exists x_1, \ldots, x_p \cdot \Psi$ est prouvable au premier ordre si et seulement s'il existe un entier k, et k substitutions $\sigma_1, \ldots, \sigma_k$, telles que $\Psi\sigma_1 \vee \ldots \vee \Psi\sigma_k$ soit propositionnellement prouvable.*

Dans cette formulation, $\Psi\sigma$ dénote l'application de la substitution σ à la formule sans quantificateur Ψ, et prouvable signifie prouvable dans n'importe quel système définissant la logique du premier ordre, resp. propositionnelle. (Je pourrais ici m'abstraire de ces systèmes en parlant de validité à la place; c'est en fait ce que j'ai fait dans l'article.)

Contraindre la formule en entrée à être existentielle ne restreint pas la généralité de l'étude : pour toute formule de la logique classique du premier ordre, il existe une formule existentielle dont la prouvabilité – et la validité – est équivalente à celle de départ. Il s'agit de son *herbrandisation*, c'est-à-dire de la formule de départ où toute variable universelle est remplacée par un terme de Skolem; l'herbrandisation est le dual de la *skolémisation*, qui conserve la satisfiabilité et où ce sont les variables existentielles qui sont remplacées par des termes de Skolem. Je reviendrai sur ce point en section 2.1.3.

On en déduit immédiatement quatre mesures de difficulté de théorèmes. La première est la *multiplicité* κ, qui n'est rien d'autre que le k minimum garantissant l'existence des k substitutions mentionnées, ou $+\infty$ s'il n'en existe pas. Il est bien vrai que k est fini si et seulement si la formule est prouvable. De surcroît, la multiplicité admet une description intuitive : il s'agit du nombre minimal de cas à considérer pour prouver la formule. Par exemple, pour prouver $\exists x \cdot (P(a) \vee P(b) \Rightarrow P(x))$, la multiplicité est de 2, les substitutions correspondantes étant $[a/x]$ et $[b/x]$ (envoyant x vers a et vers b respectivement). On peut reformuler cet exemple en disant que la *valeur* pour x qui permet de prouver ce théorème (le *témoin* de la validité de la formule existentielle) est descriptible sous forme d'un programme « `if` $P(a)$ `then` $x := a$ `else` $x := b$ ». Le théorème de Herbrand nous garantit que toute formule existentielle est soit improuvable, soit prouvable avec des témoins pris dans un langage de programmation simpliste permettant uniquement des expressions E de la forme « `if` $\Psi\sigma_1$ `then` σ_1 `else if` $\Psi\sigma_2$ `then` σ_2 `else` ... `else` σ_k », où une substitution $[t_1/x_1, \ldots, t_q/x_q]$ est vue comme une notation pour une définition, ou affectation $x_1 := t_1, \ldots, x_q := t_q$.

Comme la plupart des méthodes de preuve au premier ordre sont fondées sur la notion d'unification, la deuxième mesure que je propose est le nombre minimum v d'opérations d'unification entre deux formules atomiques qu'il s'agit d'effectuer pour trouver les substitutions σ_i si elles existent ($+\infty$ sinon).

La troisième mesure est très proche de la précédente, mais est moins liée au côté opérationnel : la *dimension* δ de la formule existentielle en entrée est le nombre minimum d'affectations ($:=$) dans un programme témoin tel que présenté ci-dessus.

La quatrième mesure, qui est finalement l'une des plus intuitives avec la première, est le *volume* β, ou taille minimale d'un programme témoin, la taille étant mesurée par la longueur du texte décrivant le programme, ou par la taille d'une représentation arborescente (avec ou sans partage des sous-expressions communes) du programme. (Selon que l'on partage ou pas, la taille peut varier non polynomialement, mais les résultats de complexité ne changent pas.)

Sans doute à cause du fait que toutes ces mesures sont définies à partir d'un concept unique, celui de programme témoin, ces différentes mesures définissent une notion robuste de difficulté. (Je remercie Étienne GRANDJEAN de m'avoir signalé l'importance de la notion de robustesse.) Par robuste, nous voulons dire plus précisément le résultat suivant :

Lemme 2.1.2 *Nous disons que deux mesures x et y sont* polynomialement équivalentes, *ou* P-équivalentes, *si et seulement s'il existe deux polynômes P et Q tels que $x \leq P(y)$ et $y \leq Q(x)$ pour toutes les entrées du problème de preuve.*

Alors, si n est le nombre de sous-formules atomiques de Ψ, s la taille maximale d'une sous-formule atomique de Ψ, et S la taille de Ψ, les mesures κns, κS, νs, δ et β sont P-équivalentes.

En particulier, elles sont aussi P-équivalentes à δs, βs, ou diverses autres mesures. L'intérêt de la P-équivalence est que si notre problème de preuve à mesure m bornée est complet pour une classe de complexité C stable par transformation polynomiale, tout problème de preuve à mesure P-équivalente m' bornée est aussi C-complète. C'est ainsi que l'on peut dire que les différentes mesures définissent un concept robuste de difficulté des énoncés classiques du premier ordre.

Le résultat principal de l'article est alors:

Théorème 2.1.3 *Quelle que soit la mesure m considérée (κns, κS, νs, δ ou β), le problème de savoir si une formule existentielle en entrée est de non-évidence inférieure ou égale à m est Σ_2^p-complet.*

Rappelons que Σ_2^p est la classe $\mathrm{NP}^{\mathrm{NP}}$, autrement dit aussi la classe des problèmes de la forme $\exists x \in X \cdot \forall y \in Y \cdot P(x,y)$, où X et Y sont deux classes d'objets de tailles polynomiales, et P est un prédicat testable en temps polynomial [GJ79].

Le résultat est relativement étonnant, d'une part parce que Σ_2^p est une classe de complexité pour laquelle peu de problèmes naturels complets sont connus. D'autre part, parce qu'il contredit l'intuition couramment répandue que la recherche de preuve est un pur problème combinatoire (qui serait alors plutôt dans NP), ou est assimilable à un jeu à deux joueurs comme les échecs [Sti88] (qui serait alors plutôt PSPACE-complet ou même DEXPTIME-complet). En fait, la structure du problème est à la fois plus compliquée que celle des problèmes de NP et moins compliquée que celle de recherche de stratégies gagnantes dans des jeux, et est de la forme:

$$\exists \sigma \cdot \forall \rho \cdot (\rho \models \Psi^k \sigma) \tag{2.1}$$

où Ψ^k est la formule $\Psi_1 \vee \Psi_2 \vee \ldots \vee \Psi_k$, $k \leq \kappa$, chaque Ψ_i, $1 \leq i \leq k$, est une version de Ψ avec toutes ses variables renommées en de nouvelles variables, ρ est une interprétation booléenne des formules atomiques de Ψ^k, et σ est une substitution que l'on peut décrire en une forme triangulaire qui, à toute variable libre dans Ψ^k, associe un sous-terme de Ψ^k.

Le problème $\forall \rho \cdot (\rho \models \Psi^k \sigma)$ est celui de la validité (ou de façon équivalente de la prouvabilité) propositionnelle de la formule sans quantificateur Ψ^k, et est donc soluble en temps polynomial dans le cas où Ψ est un ensemble de 2-clauses, ou de clauses de HORN, notamment. L'étude des clauses de HORN est en particulier intéressante, le cas se présentant souvent en preuve, et toujours en programmation logique. (Vu que nous nous intéressons à la prouvabilité plutôt qu'à l'insatisfiabilité comme il est de coutume, nous appellerons clause de HORN toute conjonction de littéraux dont au plus un est négatif, un ensemble de clauses de HORN étant alors vu comme une disjonction.) Le théorème devient dans ce cas :

Théorème 2.1.4 *Lorsque Ψ est un ensemble de clauses de* HORN, *le problème de l'évidence est NP-complet.*

Cette approche permet donc de comprendre au moins une raison pour laquelle la recherche de preuve est plus simple sur des ensembles de clauses de HORN que sur des formules générales. Toute méthode de recherche de preuve est naturellement obligée de parcourir l'espace de recherche des preuves dans un certain ordre. La méthode des connexions, par exemple, est naturellement contrôlée par la multiplicité; la résolution est naturellement contrôlée par le nombre de clauses engendrées et leur taille, ce qui est relié à la mesure κns. La différence entre NP et Σ_2^p, si elle est exploitée (et si elle est réelle), permet ainsi d'aboutir plus rapidement à une preuve dans la plupart des méthodes connues dans le cas de clauses de HORN.

On peut encore raffiner l'analyse de la nature du problème de recherche de preuve (à difficulté bornée) en remarquant que la substitution σ à chercher peut être trouvée comme solution d'un problème d'unification. Un problème d'unification est le problème suivant:

ENTRÉE : un ensemble de couples (A_i, A'_i), $1 \leq i \leq m$, de formules atomiques.

QUESTION : existe-t-il une substitution σ telle que $A_i\sigma = A'_i\sigma$ pour tout i, $1 \leq i \leq m$?

Autrement dit, on peut représenter l'ensemble de couples E comme une formule encore notée E, de la forme $\bigwedge_{i=1}^m A_i \Leftrightarrow A'_i$, et le problème de l'évidence peut se réécrire :

$$\exists E \cdot E \text{ unifiable} \wedge \forall \rho \cdot (\rho \models E \Rightarrow \Psi^k) \qquad (2.2)$$

où E parcourt l'ensemble des ensembles de couples de sous-formules atomiques de Ψ^k, représentable en taille polynomiale en fonction de kns, $k \leq \kappa$. On retrouve l'appartenance à Σ_2^p si l'on remarque que le test d'unifiabilité de E est polynomial en la taille de E, donc en kns.

L'avantage de cette reformulation est qu'elle permet de généraliser l'analyse du problème de l'évidence aux formules du premier ordre interprétées dans une théorie équationnelle T. Dans ce cas, le problème de l'évidence reste de la forme (2.2), mais où le test d'unifiabilité de E est désormais à effectuer modulo la théorie T [Plo72]. Les notions de multiplicité κ, de nombre minimum v de couples d'atomes (A_i, A'_i) dans E à unifier modulo T, de dimension δ (nombre minimum de variables dans le domaine d'un unificateur de E), de volume β (taille minimale d'un unificateur de E) s'adaptent immédiatement, et mènent au résultat suivant :

Théorème 2.1.5 *Soit T une théorie équationnelle.*

Si l'unifiabilité modulo T est un problème Σ_j^p, alors le problème de l'évidence est dans $\Sigma_{\max(2,j)}^p$ et est Σ_2^p-difficile.

En particulier, si l'unifiabilité modulo T est dans Σ_2^p, alors le problème de l'évidence est Σ_2^p-complet.

Et si l'unifiabilité modulo T est dans une classe C contenant coNP et stable par conjonction, alors le problème de l'évidence est dans NP^C.

Le deuxième point est particulièrement intéressant : pour peu que l'unifiabilité modulo T ne soit pas plus complexe qu'un problème Σ_2^p, la preuve modulo T n'est pas plus fondamentalement plus complexe que dans le cas sans symboles interprétés, selon cette approche. Un cas d'école est la preuve modulo la théorie d'un symbole de fonction associatif et commutatif (modulo AC). Si l'on utilise une méthode fondée sur l'unification AC, c'est-à-dire qui calcule des ensembles complets d'unificateurs AC, il existe des exemples de formules Ψ pour lesquelles tester l'évidence demande un temps doublement exponentiel au moins [KN92]. Mais l'AC-*unifiabilité* étant seulement NP-complète, une procédure qui, par exemple, accumulerait dans un ensemble E les couples de formules atomiques (A_i, A'_i) sans les unifier – c'est-à-dire sous formes d'ensembles de *contraintes* – et testerait de temps en temps l'AC-unifiabilité de E, serait seulement simplement exponentielle. Ceci correspond précisément à suivre la structure du problème de l'évidence telle que décrite dans l'équation (2.2).

Les remarques concernant les clauses de HORN et l'AC-unifiabilité correspondent relativement bien aux intuitions que l'on peut retirer de la pratique des méthodes de démonstration automatique. En ce qui concerne les clauses de HORN, il est vrai que Prolog et les techniques de preuve utilisant la technologie de Prolog sur les clauses de HORN permettent d'obtenir des niveaux d'efficacité supérieurs aux techniques de preuve pour des ensembles de clauses quelconques, ou des formules générales du premier ordre. L'utilisation de contraintes en preuve, que l'approche permet ici de justifier dans le cadre des théories équationnelles, est déjà un des piliers de la programmation logique avec contraintes, et promet d'être l'une des améliorations importantes des techniques classiques de preuve. Je considère ces deux points comme une première confirmation de la valeur opératoire de l'étude du problème de l'évidence. Bien que la théorie nous apprenne ici finalement relativement peu de choses, elle nous indique au moins qu'une structure naturelle de la recherche de preuve consiste en une recherche existentielle (de substitution, ou d'ensemble de couples de formules atomiques), avec une recherche universelle imbriquée (une vérification de validité propositionnelle), ce que j'exploite en section 3.2.

L'approche s'adapte au cas de la logique du premier ordre avec égalité, ce dont je parle en section 2.2, mais elle y est déjà moins robuste.

2.1.3 Quelques remarques

Une première remarque concernant l'approche du problème de la recherche de preuve par le problème de l'évidence, est que c'est l'une des rares méthodes existantes permettant de quantifier la complexité des problèmes de preuve qui soit raisonnablement indépendante du système de preuve que l'on adopte pour la logique étudiée (déduction naturelle, séquents de GENTZEN, tableaux, résolution, etc.) La seule hypothèse est que les mesures de non-évidence que l'on adopte sont suffisamment intuitives et d'une valeur empirique suffisante. Le prix à payer est qu'il est pour l'heure impossible d'affirmer que certains problèmes sont plus complexes que d'autres (par exemple, la preuve au premier ordre par rapport à la preuve d'ensembles de clauses de HORN), tant que l'on n'a pas répondu à diverses questions fondamentales de la théorie de la complexité, comme $\mathrm{NP} \neq \Sigma_2^p$.

Un défaut de la méthode, telle que je l'ai appliquée en utilisant le théorème de HERBRAND, est qu'elle suppose que l'on a d'abord transformé la formule à prouver en forme existentielle par herbrandisation. Matthias BAAZ et Alexander LEITSCH [BL94] ont montré que la multiplicité κ pouvait varier dans des rapports non élémentaires, c'est-à-dire dépassant toute tour finie d'exponentielles en fonction de la taille de la formule en entrée, selon qu'elle était effectuée naïvement ou intelligemment. Ce dernier effet dépasse largement une complexité dans Σ_2^p, c'est-à-dire simplement exponentielle : les résultats de la section précédente ne consistent-ils pas alors à chercher la paille dans l'œil du voisin (la méthode de preuve), alors que j'ai une poutre dans l'œil (que j'ignore l'effet de la skolémisation) ? La réponse est à mon avis négative, et voici pourquoi.

Constatant que toutes les méthodes de skolémisation – et d'herbrandisation – classiques, incluant celles étudiées par BAAZ et LEITSCH, tournent en temps polynomial en la taille de la formule en entrée, il est raisonnable d'étendre la définition des différentes mesures $m(\Phi)$ de la section 2.1.2 à toutes les formules du premier ordre, non seulement les formules existentielles, par :

Définition 2.1.6 *Soit m une mesure de non-évidence définie sur les formules existentielles. Pour toute procédure d'herbrandisation h en temps polynomial, on définit la mesure m_h sur toutes les formules du premier ordre par $m_h(\Phi) = m(h(\Phi))$.*

Il est par contre totalement déraisonnable de définir une unique mesure $\overline{m}$ sur toutes les formules du premier ordre par :

$$\overline{m}(\Phi) = \min(m(h(\Phi)) \mid h \text{ algorithme d'herbrandisation en temps polynomial})$$

En effet, parmi les procédures d'herbrandisation h en temps polynomial, nous trouvons toutes les fonctions $h_{\Phi'}$, pour chaque formule Φ', définies par $h_{\Phi'}(\Phi) = h_0(\Phi)$ si $\Phi \neq \Phi'$ – h_0 étant une procédure d'herbrandisation polynomiale choisie à l'avance – et $h_{\Phi'}(\Phi') = \mathbf{T}$ (une tautologie propositionnelle quelconque) si Φ' est prouvable,

$h_{\Phi'}(\Phi') = \mathbf{F}$ (une antilogie propositionnelle quelconque) si Φ' n'est pas prouvable. Cet argument prouve que $\overline{m}$ serait la fonction associant 0 aux théorèmes et $+\infty$ aux non-théorèmes, quelle que soit la mesure m de départ prise parmi les quatre de la section 2.1.2, et n'aurait donc aucun sens. Restreignons-nous donc à des mesures paramétrées m_h.

Il est remarquable que, quelle que soit la mesure m_h, tous les résultats de la section 2.1.2 restent inchangés. Par exemple, la Σ_2^p-complétude du problème de l'évidence sans symbole interprété est une conséquence immédiate du fait que h est supposée tourner en temps polynomial. En fait, le résultat de BAAZ et LEITSCH est orthogonal à ceux de la section 2.1.2, et établit que l'on peut *gagner* jusqu'à au moins un nombre non borné d'exponentielles en efficacité avec un algorithme d'herbrandisation ou de skolémisation adéquat; je m'intéresse à la réalisation d'un tel algorithme en section 3.1.

La dernière remarque que je ferai sur cette approche est liée à la présentation que j'ai faite ci-dessus du théorème de HERBRAND : une formule existentielle est prouvable si et seulement si on peut trouver des témoins pour les variables existentielles sous la forme de programmes dans un certain langage. Ceci diffère de la présentation des mesures de difficulté de l'article [Gou94b]. Si ces programmes sont de simples programmes conditionnels en logique du premier ordre, des formalismes logiques plus expressifs nécessitent de trouver les témoins de formules existentielles sous forme de programmes écrits dans des langages plus riches; par exemple, en arithmétique de *Peano* du premier ordre, le langage de programmation correspondant doit exprimer exactement les fonctions récursives primitives d'ordre supérieur, d'après l'interprétation « no-counterexample » de Georg KREISEL ([Gir87b], annexe 7.D). La remarque que différentes mesures de taille des programmes du langage de programmation associé à une logique peuvent donner lieu à des mesures pertinentes de la difficulté de théorèmes ouvre à mon avis une voie d'étude fine de la complexité de la recherche de preuves dans différents formalismes logiques, incluant l'arithmétique du premier ordre. (Dans ce dernier cas, noter que l'approche est différente de celle de PARIKH.)

2.2 E-unification rigide et matings équationnels

2.2.1 Définitions et problèmes

La logique du premier ordre telle qu'elle a été approchée en section 2.1 est une logique sans aucun symbole interprété. L'étude que j'ai faite de la complexité de l'évidence s'est étendue à la logique du premier ordre modulo une théorie équationnelle, mais quels sont les résultats analogues sur des logiques dans lesquels davantage de symboles sont interprétés? En particulier, qu'en est-il de la logique du premier ordre avec égalité, c'est-à-dire de la logique dans laquelle le symbole de prédicat binaire $\doteq$ est interprété comme la relation d'égalité? Deux de mes articles [Gou93c, Gou94h] apportent un début de réponse dans le cas le plus difficile; je décris ici le cadre de cette recherche, les résultats, et les conclusions que l'on peut en tirer.

Le problème de l'évidence en logique du premier ordre avec égalité est important, d'une part parce que l'égalité est l'une des relations les plus souvent utilisées, et d'autre part parce qu'il est irréaliste en pratique de vouloir utiliser des méthodes de preuve pour le premier ordre en codant l'égalité par les axiomes de réflexivité, de symétrie, de transitivité et de congruence (si $x_i \doteq y_i$ pour tout i, $1 \leq i \leq n$, alors $f(x_1, \ldots, x_n) \doteq f(y_1, \ldots, y_n)$ pour chaque symbole de fonction n-aire, et $P(x_1, \ldots, x_n) \Rightarrow P(y_1, \ldots, y_n)$ pour chaque symbole de prédicat n-aire).

En logique du premier ordre avec égalité, le théorème de HERBRAND devient :

Théorème 2.2.1 (HERBRAND-GALLIER) *Une formule $\exists x_1, \ldots, x_p \cdot \Psi$ est prouvable au premier ordre avec égalité $\doteq$ si et seulement s'il existe un entier k, et k substitutions σ_1, ..., σ_k, telles que $\Psi\sigma_1 \vee \ldots \vee \Psi\sigma_k$ soit prouvable en théorie non quantifiée de l'égalité.*

comme il est présenté dans [GNRS92]. La

théorie non quantifiée de l'égalité est un système formel défini sur les formules du premier ordre sans quantificateurs, et muni d'une relation de déduction $\vdash_{0,=}$, définie comme suit.

- si la formule Φ est déductible de l'ensemble d'hypothèses Γ en logique propositionnelle classique, alors $\Gamma \vdash_{0,=} \Phi$.
- (réflexivité) $\Gamma \vdash_{0,=} t \doteq t$ pour tout terme t;
- (symétrie) $\Gamma, s \doteq t \vdash_{0,=} t \doteq s$ pour tous termes s et t;
- (transitivité) $\Gamma, s \doteq t, t \doteq u \vdash_{0,=} s \doteq u$ pour tous termes s, t et u;
- (congruence) $\Gamma, s_1 \doteq t_1, \ldots, s_n \doteq t_n \vdash_{0,=} f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n)$ pour tout symbole de fonction f d'arité n, et tous termes $s_1, \ldots, s_n, t_1, \ldots, t_n$;
- (congruence, bis) $\Gamma, s_1 \doteq t_1, \ldots, s_n \doteq t_n, P(s_1, \ldots, s_n) \vdash_{0,=} P(t_1, \ldots, t_n)$ pour tout symbole de prédicat P d'arité n, et tous termes $s_1, \ldots, s_n, t_1, \ldots, t_n$.

La théorie non quantifiée de l'égalité est donc une extension de la logique propositionnelle, et la remplace dans l'énoncé du théorème de HERBRAND.

Il est alors possible de modifier les arguments de la section 2.1.2 pour obtenir des mesures de difficulté pertinentes à la logique du premier ordre avec égalité. Les définitions sont les mêmes : la *multiplicité* κ est le k minimum garantissant l'existence des k substitutions mentionnées, ou $+\infty$ s'il n'en existe pas. La *dimension* δ est le cardinal minimum du domaine d'une substitution σ telle que $\vdash_{0,=} \Psi^k\sigma$ pour un certain k. (On rappelle que $\Psi^k = \Psi_1 \vee \ldots \vee \Psi_k$, où les Ψ_i sont des renommages de Ψ, d'ensembles de variables libres deux à deux disjoints.) Le *volume* β est la taille minimum d'une telle substitution – c'est-à-dire d'un programme conditionnel témoin. Cependant, il n'y a pas d'équivalent naturel au nombre minimal v d'opérations d'unifications à réaliser pour trouver σ : la théorie de l'égalité n'est pas une théorie équationnelle, et l'on ne peut a priori pas chercher σ en unifiant successivement des paires d'atomes dans une théorie.

Dans le cas où l'on mesure la non-évidence par le volume β, on a le résultat facile suivant [Gou93b] :

Théorème 2.2.2 *En logique du premier ordre avec égalité, et pour la mesure de non-évidence β (volume), le problème de savoir si une formule existentielle en entrée est de non-évidence inférieure ou égale à m est Σ_2^p-complet.*

Il est en effet Σ_2^p-difficile, parce que le sous-problème de la non-évidence de formules sans le symbole $\doteq$ est exactement celui le problème de la non-évidence en logique du premier ordre sans symboles interprétés, qui est Σ_2^p-complet (théorème 2.1.3). Il est dans Σ_2^p, parce que de la forme suivante, une fois traduit sémantiquement :

$$\exists\sigma \cdot \forall\rho \cdot \text{si } \rho \text{ équationnelle, alors } \rho \models \Psi^m$$

où σ varie parmi toutes les substitutions de taille inférieure ou égale à m sur le langage de la formule Ψ^m, (et est donc devinable en temps polynomial), ρ varie parmi toutes les interprétations de formules atomiques de Ψ^m (c'est-à-dire parmi tous les sous-ensembles de ces formules atomiques, qui sont devinables en temps polynomial également), et ρ est dite *équationnelle* si et seulement si elle valide tous les axiomes et toutes les règles de la théorie non quantifiée de l'égalité sur le langage de Ψ^m. On remarquera que si la taille de σ est bornée par m, en particulier si le domaine de σ ne contient pas plus de m variables, alors on n'a besoin que d'au plus m copies de Ψ, d'où l'utilisation de Ψ^m. Vérifier si $\rho \models \Psi^m$ est faisable en temps polynomial, et vérifier que ρ est équationnelle est faisable en temps polynomial par un algorithme dérivé de la *clôture de congruence* [DST80, Gal87, NO80].

Dans le cas où l'on mesure la non-évidence par la dimension δ, on peut se ramener au cas de la mesure κS, où S est la taille de Ψ (ou κns, où n est le nombre de sous-formules atomiques de Ψ et s est la taille maximale de ces sous-formules) comme en section 2.1.2, parce que comme dans le cas de la logique sans égalité, ces mesures sont P-équivalentes (cf. l'argument de [Gou94b]). D'autre part, le problème de l'évidence à mesure

κS se ramène comme en section 2.1.2 à l'étude du problème :

ENTRÉE : une formule Φ sans quantificateurs, de taille S

QUESTION : existe-t-il une substitution σ telle que $\vdash_{0,=} \Phi\sigma$?

En effet, ceci est le problème de l'évidence avec $\kappa = 1$; et réciproquement, le problème de l'évidence avec κ quelconque est exactement ce problème, où Φ est Ψ^κ.

Ce problème a été appelé par Jean Gallier *et al.* le problème de l'existence de *matings équationnels* à travers Φ [GRS87, GNRS92]. Ils ont montré qu'on pouvait ramener ce problème, sans perte de généralité et en temps polynomial, au sous-cas où le seul symbole de prédicat utilisé dans Φ était l'égalité $\doteq$. (On recode un atome non égalitaire $P(t_1, \ldots, t_n)$ sous forme d'une équation $P(t_1, \ldots, t_n) \doteq \top$, où P est désormais un symbole de fonction et $\top$ est une nouvelle constante, représentant le vrai. Ceci est correct même sans utiliser de sortes [Gou93b].)

Si l'on écrit maintenant Φ en forme normale conjonctive $C_1 \wedge C_2 \wedge \ldots \wedge C_n$, où les C_i, $1 \leq i \leq n$, sont des implications $s_i^1 \doteq t_i^1 \wedge \ldots \wedge s_i^{m_i} \doteq t_i^{m_i} \Rightarrow s_i^{m_i+1} \doteq t_i^{m_i+1} \vee \ldots \vee s_i^{n_i} \doteq t_i^{n_i}$, un sous-problème à explorer est celui de savoir s'il existe une substitution σ_i telle que $\vdash_{0,=} C_i\sigma_i$ pour chaque i séparément. Le problème de l'existence d'un mating équationnel est alors celui de l'existence d'une *même* substitution σ telle que $\vdash_{0,=} C_i\sigma$ pour tous les i, $1 \leq i \leq n$. On peut espérer a priori qu'un tel σ pourra être construit comme une combinaison à trouver des σ_i, $1 \leq i \leq n$, comme cela se passe au premier ordre sans égalité.

Dans la méthode des matings, C_i est ce qu'on appelle un chemin vertical complètement expansé. Faisant abstraction des détails algorithmiques, si C est une implication de la forme $s_1 \doteq t_1 \wedge \ldots \wedge s_m \doteq t_m \Rightarrow s_{m+1} \doteq t_{m+1} \vee \ldots \vee s_n \doteq t_n$, le problème de la recherche de σ telle que $\vdash_{0,=} C\sigma$ est donc celui de la *fermeture d'un chemin équationnel* C.

Alors que le problème de la fermeture des chemins C se ramène en logique sans égalité à trouver une formule atomique à gauche du signe $\Rightarrow$, et une formule atomique à droite du signe $\Rightarrow$ qui soient unifiables, dans le cas égalitaire il se ramène à trouver une équation $s_j \doteq t_j$, $m+1 \leq j \leq n$, à droite de l'implication, telle qu'il existe σ telle que $\vdash_{0,=} s_1\sigma \doteq t_1\sigma \wedge \ldots \wedge s_m\sigma \doteq t_m\sigma \Rightarrow s_j\sigma \doteq t_j\sigma$. L'opération d'unification du cas non égalitaire se trouve alors remplacée par le problème suivant, appelé *E-unifiabilité rigide* de s et t par Jean Gallier *et al.*, et que j'appelle aussi E-*unifiabilité rigide* du problème $E \vdash^?_{0,=} s \doteq t$:

ENTRÉE : un ensemble fini E d'équations, deux termes s et t.

QUESTION : existe-t-il une substitution σ telle que $E\sigma \vdash_{0,=} s\sigma \doteq t\sigma$?

où $E\sigma$ est l'ensemble des équations $u\sigma \doteq v\sigma$, lorsque $u \doteq v$ parcourt E. Une telle solution σ est alors appelée E-*unificateur rigide* du problème $E \vdash^?_{0,=} s \doteq t$, ou *E-unificateur rigide* de s et t.

On peut alors remonter au cas des matings équationnels en demandant à fermer plusieurs chemins simultanément, ce que Jean Gallier *et al.* appellent *$\vec{E}$-unification rigide*, et que j'appelle aussi E-*unifiabilité rigide simultanée* :

ENTRÉE : une famille finie d'ensembles finis d'équations E_i, de termes s_i, et de termes t_i, pour $1 \leq i \leq n$.

QUESTION : existe-t-il une substitution σ telle que $E_i\sigma \vdash_{0,=} s_i\sigma \doteq t_i\sigma$ pour tout i, $1 \leq i \leq n$?

Ceci résout presque le problème des matings équationnels sur une formule Φ sans quantificateurs : pour utiliser le problème ci-dessus, nous devons d'abord extraire les chemins C_i de Φ, qui sont en nombre exponentiel, et nous devons ensuite choisir une équation $s_i \doteq t_i$ non niée sur chaque chemin, et construire le problème $E_i \vdash^?_{0,=} s_i \doteq t_i$, où E_i est l'ensemble des équations niées de C_i.

2.2.2 Résultats en E-unification rigide simple

Les problèmes liés à l'E-unification rigide – simple, simultanée, matings équationnels – semblent simples à résoudre à première vue, mais sont en réalité assez retors. La simplicité apparente du problème tient peut-être au fait que vérifier si $E\sigma \vdash_{0,=} s\sigma \doteq t\sigma$ est valide (ou vérifier que σ est un E-unificateur rigide, c'est-à-dire une solution de $E \vdash^?_{0,=} s \doteq t$) est faisable en temps polynomial – même quasi-linéaire – et ce par un algorithme simple [NO80], l'algorithme de *clôture de congruence*.

Si l'on pouvait affirmer d'emblée qu'il existait des E-unificateurs rigides les plus généraux, et qu'un tel E-unificateur rigide de s et de t le plus général est descriptible comme une substitution en forme triangulaire envoyant des variables libres dans E, s ou t vers des sous-termes de E, s ou t, on en déduirait immédiatement que le problème de l'E-unifiabilité rigide est décidable et dans NP. (Donc NP-complet, la NP-difficulté étant facile à prouver [GRS87].) C'est effectivement le cas, et Jean Gallier *et al.* sont les premiers à l'avoir prouvé [GSNP88], en utilisant une méthode fondée sur la complétion sans échec, et dans laquelle les variables ne sont pas renommées lors du calcul des paires critiques [GNRS92]. La méthode est compliquée, et visiblement difficile à coder. La question que je me suis posée dans [Gou93c] était de savoir si l'on pouvait inventer un autre algorithme, si possible plus efficace, et fondé sur la clôture de congruence, elle-même une procédure de vérification très efficace.

Le résultat a été positif. D'un point de vue théorique, cet article propose une preuve un peu plus simple que celle de [GNRS92] de la NP-complétude du problème de l'E-unifiabilité rigide; il propose aussi une notion d'ordre "est plus général que" entre unificateurs plus naturelle et légèrement plus générale que celle de Jean GALLIER *et al.*, et propose un algorithme calculant un ensemble complet d'E-unificateurs rigides. Je propose des arguments tendant à prouver qu'il est plus efficace que celui de Jean Gallier *et al.*, bien qu'il soit difficile de comparer les deux sans les coder.

Depuis la publication de cet article, j'ai considérablement simplifié la preuve d'existence d'ensembles complets d'E-unificateurs rigides, et de l'appartenance à NP du problème de décision associé. La technique de base est l'utilisation de transformations de preuves dans un format adapté à la théorie non quantifiée de l'égalité (les *preuves graphiques*), ces transformations s'apparentant à des techniques d'élimination des coupures. J'en fais mention dans le rapport technique qui est la version longue de [Gou94h]; j'ai d'autre part développé la notion dans un exposé fait en présence de Jean GALLIER à l'université de Pennsylvanie fin juin 1994, et je l'ai rédigée dans un manuscrit non publié de 1995. La raison principale de non-publication a été la preuve d'Eric DE KOGEL [dK95] d'appartenance à NP, qui est nettement plus simple, mais moins constructive. Ces notions mènent naturellement à un algorithme de recherche exhaustive à la DAVIS-PUTNAM-LOGEMANN-LOVELAND, qui n'est peut-être pas la méthode la plus bête en réalité.

2.2.3 Résultats en E-unification rigide simultanée

Le problème de l'E-unification rigide simultanée est beaucoup plus compliqué, et ce à plusieurs titres. D'abord, il est difficile de combiner deux substitutions σ_1 et σ_2, respectivement E-unificateurs rigides de $E_1 \vdash^?_{0,=} s_1 \doteq t_1$ et de $E_2 \vdash^?_{0,=} s_2 \doteq t_2$, pour en former un ensemble complet d'E-unificateurs rigides simultanés pour les deux problèmes, et moins généraux que σ_1 et σ_2. Ceci est dû à la nature de la notion de relation "est plus général que" dont nous avions besoin pour le problème non simultané : le critère fait en effet intervenir, pour chaque σ_i, l'ensemble E_i et la notion de conséquence équationnelle de façon non triviale. En fait, il est impossible de combiner deux substitutions, parce que l'E-unifiabilité rigide simultanée est indécidable, comme l'ont montré Anatoli DEGTYAREV et Andrei VORONKOV [DV96b]. Ce résultat a été confirmé et amélioré ensuite par David PLAISTED [Pla95], puis par Margus VEANES [Vea97], et infirme les résultats de décidabilité précédents, parmi lesquels ceux de Jean GALLIER *et al.* [GNRS92] (ou le problème était identifié comme NP-complet) et, à mon grand dam, le mien [Gou94h] (DEXPTIME-complet).

Ensuite, le problème de l'E-unification simultanée est compliqué parce qu'il était aussi relativement difficile de prouver qu'il est difficile. La plupart des tentatives pour montrer qu'il était C-difficile, pour une classe de complexité C, ne montent pas plus haut que NP-difficile. C'est assez paradoxal, lorsque l'on sait que plusieurs personnes (Gérard Bécher, de Caen, Andrei Voronkov, d'Uppsala, et moi-même) avaient effectivement pensé à un moment ou à un autre que le problème était probablement indécidable. En particulier, essayer de recoder le problème de la validité d'une formule propositionnelle classique quantifiée – QBF, le prototype des problèmes PSPACE-complets – en E-unification rigide n'avait jamais abouti. L'un des résultats de mon article [Gou94h] (dont seule l'affirmation d'appartenance à DEXPTIME était erronée) était que le problème était DEXPTIME-difficile, ce qui améliorait de beaucoup la borne inférieure de complexité du problème. L'outil utilisé était la DEXPTIME-difficulté du problème de la vacuité de l'intersection de langages d'arbres définis par des automates d'arbres bottom-up déterministes (que je prouvais en annexe). Les preuves d'indécidabilité ultérieures ont été trouvées d'abord par des codages du problème de la semi-unification monadique, puis de l'unifiabilité du second ordre et directement du dixième problème de Hilbert (Voronkov *et al.*), puis ont été simplifiées en des codages du problème de correspondance de Post (Plaisted), puis les résultats ont été améliorés grâce à de nouveaux codages de problèmes d'automates d'arbres bottom-up (Veanes).

L'article [Gou94h], malgré son erreur et le fait que la DEXPTIME-difficulté du problème soit trivialement impliquée par son indécidabilité, propose cependant un résultat qui n'a pas été amélioré, à savoir la PSPACE-difficulté de l'E-unifiabilité rigide simultanée dans le cas monadique, où tous les symboles de fonction sont d'arité au plus 1. Le sous-problème monadique est décidable [DMV96] et est lié aux problèmes de décision de formules construites sur l'addition, l'égalité et le prédicat de divisibilité en arithmétique. PSPACE est la meilleure borne inférieure connue pour ce problème à l'heure actuelle, alors que les bornes supérieures sont de l'ordre de plusieurs tours d'exponentielles.

Ensuite, cette erreur m'a été personnellement profitable, dans la mesure où, en cherchant à comprendre comment l'unification du second ordre se recodait en problème d'E-unification rigide simultanée [DV96b] et ce que mon algorithme erroné ne prenait pas en compte en unification du second ordre à travers le codage, il m'est venu à l'idée d'examiner le problème de l'unification dans des logiques d'ordre supérieur présentant des restrictions de *ramification*, notion introduite au début du siècle par Bertrand Russell. J'en parlerai en section 3.4.

2.2.4 Matings équationnels et complexité de l'évidence équationnelle

Finalement, le résultat d'indécidabilité de l'E-unifiabilité rigide simultanée montre que le problème de l'évidence pour la mesure κS est infiniment plus complexe (indécidable) que pour le volume β (Σ_2^p, par le théorème 2.2.2). Ceci est un résultat embarrassant en ce qui concerne la robustesse de la notion de complexité de l'évidence : cette notion est en effet relativement fragile. En particulier, vouloir ajouter des contraintes sur la multiplicité de Herbrand d'une formule avec égalité, loin de simplifier le problème, peut rendre des formules évidentes (en termes de volume) arbitrairement difficiles (en termes de multiplicité). (Une conséquence surprenante, due à Voronkov *et al.*, est que la prouvabilité dans le fragment prénexe $\exists\exists\forall^*$ de la logique intuitionniste avec égalité est indécidable.)

Ce résultat de fragilité peut cependant être retourné en un conseil pratique quant au choix de méthodes de preuve au premier ordre classique avec égalité. En supposant que les deux mesures sont toutes deux raisonnables, il sera préférable a priori de coder une méthode de preuve dans laquelle la mesure naturelle de non-évidence soit le volume; plutôt que d'explorer l'espace de recherches des preuves par multiplicités κ croissantes, il est nettement moins complexe de les explorer par tailles de substitutions croissantes (volumes croissants). C'est en gros ce qui se passe dans les méthodes par résolution et paramodulation, où le facteur limitatif est la taille de l'ensemble des clauses générées, à l'intérieur desquelles la substitution en question est codée.

Dans les méthodes de tableaux ou de matings au contraire, il est naturel, comme dans la méthode des matings équationnels de Jean GALLIER *et al.* de chercher les σ instanciant Ψ^k à une tautologie équationnelle, pour $k \in \mathbb{N}$ croissants – autrement dit de garder k copies *rigides* de chaque variable. Ceci est en fait infaisable, à cause de l'indécidabilité du problème. Par contre, laisser les variables flexibles (instantiables plusieurs fois), et contrôler la direction de la recherche de preuve via – entre autres – la taille des substitutions partielles trouvées, mène à un contrôle de la recherche via la notion de volume, pour lequel la classe de complexité naturelle du problème n'est que Σ_2^p. Ceci me mène à penser que des techniques ressemblant à la paramodulation sont intrinsèquement les plus efficaces que l'on puisse utiliser – on ne pourra pas en effet descendre en dessous de Σ_2^p –, et notamment la technique d'élimination de l'égalité [DV96a].

Je dois cependant tempérer ce pessimisme vis-à-vis sur l'avenir de l'E-unification rigide simultanée par deux observations. Premièrement, ce problème est incontournable en logique intuitionniste (c'est la cas des formules prénexes intuitionnistes, où la multiplicité de HERBRAND est nécessairement 1). Ensuite, même en logique classique, David PLAISTED [Pla95] a fait la remarque suivante (dans notre cas, rappelons qu'une clause est vue comme une conjonction, et que l'on cherche à décider de la validité d'une formule) :

Théorème 2.2.3 (PLAISTED) *Soit Φ un ensemble de clauses. Si chaque clause de Φ dans laquelle apparaît une sous-formule atomique de la forme $s \doteq t$ est une clause de* HORN, *alors l'existence d'un mating équationnel à travers Φ est Σ_2^p-complet.*

Si de plus Φ est un ensemble de clauses de HORN, *alors ce même problème est NP-complet.*

David PLAISTED a présenté ce résultat à l'atelier sur l'unification organisé lors de la douzième conférence sur la déduction automatique (CADE-12, Nancy, France, juin 1994). Son résultat était en fait que le problème était toujours NP-complet, mais il est apparu dans un échange de courrier entre David PLAISTED et moi en juillet 1994 que c'était la version ci-dessus qui était valide. Ainsi, non seulement les clauses de HORN simplifiaient-elles déjà la recherche de preuve dans le cas sans égalité, mais ici la simplification est drastique : on passe ainsi d'indécidable à Σ_2^p. Il reste à voir si ce cas est aussi fréquent qu'il y paraît.

Chapitre 3

Démonstration automatique

Le but de l'étude du chapitre 2 était de mieux comprendre la nature de la difficulté de la recherche de preuves dans diverses logiques au premier ordre. Que peut-on en tirer comme conséquences dans le domaine de la démonstration automatique ? En particulier, que peut-on faire pour améliorer les méthodes de preuve actuelles, ou quelles meilleures méthodes peut-on inventer ?

3.1 Simplification et skolémisation

J'ai déjà mentionné les résultats de Baaz et Leitsch [BL94], qui montraient que soigner la procédure de skolémisation – qu'elle soit d'ailleurs utilisée comme une étape préliminaire à la preuve, ou en cours de preuve comme dans la méthode des tableaux à variables libres – était crucial. Une skolémisation de mauvaise qualité peut bloquer n'importe quelle méthode de preuve utilisée ensuite sur la formule skolémisée. Ce fait était déjà plus ou moins ressenti auparavant, et Peter Andrews au moins avait fait ce constat en pratique. C'est peut-être pour cela que, contrairement à la plupart des autres livres de logique, [And86] présente une procédure de skolémisation qui ne passe pas d'abord par la mise en forme prénexe de la formule à prouver.

La notion cruciale est celle de *fausse dépendance.* Pour prouver $(\exists x \cdot \forall z \cdot \Phi) \wedge (\exists y \cdot \forall z \cdot \Phi')$, par exemple, on doit trouver une valeur pour x (et une pour y) telle que pour tout z dépendant de x (resp. y), Φ (resp. Φ') soit valide. Mais sa forme prénexe est $\exists x, y \cdot \forall z \cdot \Phi \vee \Phi'$, où maintenant z dépend à la fois de x et de y. (Il existe d'autres cas de figure.) Cette fausse dépendance peut ensuite empêcher des unifications lors du processus de recherche de preuve, retardant ainsi la découverte de la preuve. Baaz et Leitsch ont montré que ce phénomène seul pouvait augmenter la multiplicité (le nombre de copies de la formule skolémisée) d'un facteur non élémentaire en la taille de la formule (c'est-à-dire qui dépasse toute tour d'exponentielles de hauteur bornée).

L'idée de la procédure de skolémisation que j'ai présentée au Tableaux workshop de 1994 [Gou94a] était non seulement de ne pas ajouter de fausses dépendances lors du processus de skolémisation, mais encore de tenter de simplifier la formule à prouver pour *supprimer* des fausses dépendances. L'article complet traitant de cette technique, avec les algorithmes complets, une discussion de leur complexité, et des résultats expérimentaux, est [Gou95b]. Comme dans ces articles, je me placerai dans le cas où l'on cherche établir l'insatisfiabilité d'une formule, et parlerai de skolémisation plutôt que d'herbrandisation ; ceci est dû à des considérations de clarté de l'exposé, et ne change en rien la portée des techniques.

Le point de départ de ces deux articles n'est pas neuf, il s'agit d'une idée de David Hilbert et de Paul Bernays [HB39]. Considérons par exemple que $\forall x \cdot \Phi$ est une abréviation de $\neg \exists x \cdot \neg \Phi$. Alors Hilbert et Bernays fabriquent une extension du langage des termes du premier ordre en autorisant des constructions de la forme $\epsilon x \cdot \Phi$, et ajoutent les axiomes :

$$\exists x \cdot \Phi \vdash \Phi[(\epsilon x \cdot \Phi)/x] \tag{3.1}$$

exprimant que $\epsilon x \cdot \Phi$ est un terme t tel que Φ est vrai lorsque $x = t$, et :

$$\forall x \cdot \Phi \Leftrightarrow \Phi' \vdash (\epsilon x \cdot \Phi) = (\epsilon x \cdot \Phi') \qquad (3.2)$$

exprimant que ϵ est une fonction de choix. En effet, si les variables libres de $\exists x \cdot \Phi$ sont y_1, …, y_m, alors la fonction $\lambda y_1, \ldots, y_m \cdot \epsilon x \cdot \Phi$ est une fonction qui à chaque y_1, …, y_m associe un unique x parmi ceux qui rendent Φ vrai s'il en existe, ou alors un élément quelconque. L'existence sémantique d'un tel objet est garanti par l'axiome du choix.

La logique du premier ordre avec ϵ est une extension stricte, bien que conservative, de la logique du premier ordre. On peut alors utiliser l'intuition fournie par l'axiomatisation de ϵ pour mieux comprendre la skolémisation. Pour skolémiser $\exists x \cdot \Phi$, la méthode classique consiste à construire un nouveau symbole de fonction f, et à produire $\Phi[f(y_1, \ldots, y_m, y_{m+1}, \ldots, y_n)/x]$, où y_1, …, y_m, y_{m+1}, …, y_n sont les variables quantifiées dans le contexte où l'on skolémise. Peter ANDREWS [And86] avait déjà remarqué que l'on pouvait se contenter de produire $\Phi[f(y_1, \ldots, y_m)/x]$, où y_1, …, y_m sont les variables libres dans $\exists x \cdot \Phi$ (qui sont nécessairement quantifiées plus haut). Ceci évite de prendre en compte les fausses dépendances de x par rapport à y_{m+1}, …, y_n.

On peut faire encore mieux. Bernhard BECKERT, Reiner HÄHNLE et Peter H. SCHMITT ont proposé une amélioration de la δ-règle (ce qui tient lieu de skolémisation) des méthodes de tableaux à variables libres [BHlS93] dans laquelle, si l'on a à skolémiser la même formule deux fois, on peut réutiliser le même symbole de fonction f. En fait, comme ils le remarquent à la fin de leur article, et comme il est justifié par l'équation (3.2), on peut même attribuer le même symbole de fonction f à deux formules à skolémiser *équivalentes* en logique du premier ordre. Ceci est intéressant, parce que plus on identifie de symboles de SKOLEM, plus l'unification a des chances de réussir, et plus les preuves à trouver seront courtes et donc faciles à trouver.

L'idée de [Gou94a, Gou95b] est alors, pour skolémiser $\exists x \cdot \Phi$, de choisir une formule Φ' équivalente à Φ et qui ait le minimum possible de variables libres. Ceci constitue une première phase de simplification de la formule à prouver, qui est ensuite skolémisée par une méthode à la ANDREWS, améliorée de sorte que deux formules équivalentes soit associées au même symbole de SKOLEM.

Cependant, si par « équivalent », nous entendons équivalent en logique du premier ordre, alors l'idée est inapplicable, le problème de simplification ci-dessus étant indécidable. Mais il peut être intéressant de se restreindre à des sous-systèmes logiques pour lesquels on dispose d'algorithmes efficaces en pratique.

Le système choisi comprend les règles de la logique propositionnelle classique, plus les règles suivantes, où $\mathbf{F}$ et $\mathbf{T}$ sont deux constantes atomiques représentant respectivement le faux et le vrai :

- $(\exists x \cdot \mathbf{F}) = \mathbf{F}$, $(\exists x \cdot \mathbf{T}) = \mathbf{T}$;
- $(\exists x \cdot \Phi) = (A \Rightarrow \exists x \cdot \Phi[\mathbf{T}/A]) \wedge (\neg A \Rightarrow \exists x \cdot \Phi[\mathbf{F}/A])$.

où $\Phi[\mathbf{T}/A]$ et $\Phi[\mathbf{F}/A]$ sont Φ avec la sous-formule atomique A remplacée par vrai et par faux respectivement, et toutes les quantifications sont comprises à α-renommage près. La définition précise est dans [Gou94a, Gou95b].

Si l'on dispose de formes canoniques pour les formules propositionnelles (forme de Reed-Müller, c'est-à-dire formes normales pour un système de réécriture dans les anneaux Booléens; ou diagrammes de décision binaire, comme dans l'article), et si l'on représente les quantifications à l'aide d'indices de de Bruijn, le système logique ci-dessus s'interprète naturellement comme un système de réécriture convergent. Il suffit en effet d'interpréter les égalités ci-dessus, opérationnellement, comme la réécriture du côté gauche en le côté droit. L'intérêt de ces règles est qu'elles ont pour but d'éliminer les quantifications portant sur des variables non libres dans la formule quantifiée (première équation), et de normaliser le plus possible les corps des quantifications, en sortant toutes les formules atomiques qui ne dépendent pas de la variable quantifiée (seconde équation).

L'article [Gou95b] décrit comment il est possible de coder efficacement ce système de réécriture, ainsi que la skolémisation qui s'en suit. Les résultats pratiques sont encourageants : sur des jeux de tests standard, la procédure de simplification et de skolémisation tourne toujours en temps quasi-négligeable, et simplifie suffisamment certaines formules simples pour les prouver complètement.

Il reste cependant un point mystérieux et un point noir dans cette méthode.

Le mystère est le suivant : le système de réécriture ci-dessus est capable de déterminer la validité (ou l'insatisfiabilité) de certaines formules du première ordre; y a-t-il une classe naturelle de formules du premier ordre pour laquelle la procédure de simplification ci-dessus apporte une procédure de décision ? De façon surprenante, parmi les sous-classes décidables classiques du problème de la décision [DG79], il semble qu'aucune ne contienne ou ne soit incluse dans la classe des formules décidées par la procédure de simplification. Il serait intéressant de tenter de caractériser cette dernière classe d'une autre façon, qui reste à trouver.

Le point noir est la complexité théorique de l'algorithme. Je croyais au départ qu'elle serait exponentielle, à cause du comportement exponentiel du sous-système décidant l'équivalence propositionnelle, et qu'en pratique cette borne serait rarement atteinte. L'expérience sur des problèmes usuels confirme que les temps de simplification et de skolémisation restent raisonnables, mais la surprise est que l'algorithme est en fait non élémentaire. Bien que l'on puisse toujours borner le temps d'exécution et l'espace utilisé par l'algorithme par une tour de d exponentielles, où d est le degré d'imbrication des quantificateurs, il existe une famille d'exemples simples qui nécessite cette quantité de temps et d'espace. Cet exemple, fourni à la procédure de simplification, ne retourne pas en un temps humainement acceptable sauf dans les cas triviaux. Bizarrement, ces cas pathologiques semblent très rares en pratique. Il semble qu'il y ait là une parenté avec le problème de la β-équivalence de deux λ-expressions simplement typées, qui exhibe les mêmes comportements tant en théorie qu'en pratique. (Je soupçonne en particulier que le système logique utilisé par le simplificateur détermine un problème d'équivalence non élémentaire.)

Par contre, l'idée de simplifier à l'avance – ou au cours de la preuve – les formules à prouver pour en accélérer la découverte de preuves me semble une idée extrêmement intéressante à explorer. Ceci est d'autant plus vrai si l'on cherche à incorporer des théories traitant de l'égalité, de l'arithmétique, des corps réels clos (les algébriques, c'est-à-dire les systèmes d'équations polynomiales à coefficients entiers, pour lesquelles les techniques de calcul formel, comme le calcul de bases de Gröbner, pourraient être utilisés par exemple), etc.

3.2 Preuve au premier ordre

Une fois la formule d'entrée skolémisée (resp. herbrandisée), il ne reste plus qu'à en chercher une réfutation (resp. une preuve). Les résultats présentés en section 2.1 indiquent que, au premier ordre, et si les différentes mesures de non-évidence que j'ai identifiées sont pertinentes et robustes, toute procédure de recherche de preuve raisonnable doit explorer un espace de recherche en général infini, mais dont la restriction à toute profondeur non nulle doit être typique des arbres de recherche pour les problèmes Σ_2^p.

Σ_2^p n'est pas NP (jusqu'à preuve du contraire), et les opérations à effectuer aux nœuds de l'arbre de recherche sont des opérations coNP (en général Δ_2^p) et non des opérations en temps polynomial. La structure de l'arbre de recherche ressemble donc à la Figure 3.1, où la mesure de non-évidence varie polynomialement en fonction de la profondeur dans l'arbre où la preuve est trouvée. Par exemple, si l'on reprend l'interprétation de l'équation (2.1), l'arbre lui-même – que j'appellerai l'arbre *externe* dans la suite – correspond à la recherche d'une substitution σ, et chaque arbre à l'intérieur des boîtes que sont les nœuds de l'arbre externe sont des vérifications de validité $\forall\rho\cdot(\rho \models \Phi\sigma)$ de formules propositionnelles. C'est ainsi que fonctionnent

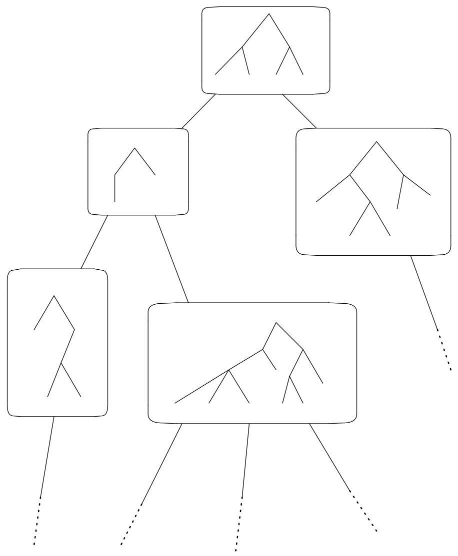

Figure 3.1: Structure de recherche de preuve

les méthodes des tableaux (ou de connexions, ou de matings). La résolution est plus compliquée à interpréter dans ce cadre, mais on retrouve la structure ci-dessus lorsque l'on sépare le principe de résolution en résolution propositionnelle (la vérification de validité) et unification (la recherche de substitution), et à condition de découpler les deux activités, comme par exemple Shie-Jue Lee et David Plaisted l'ont proposé [LP92], ou Jean-Paul Billon et Henri Fraïssé chez Bull. (En résolution, étant données deux clauses $C \vee A$ et $\neg A' \vee C'$, avec σ unificateur le plus général de A et de A', on fabrique le résolvant $C\sigma \vee C'\sigma$; dans ces dernières méthodes, on produit les deux clauses $C\sigma \vee A\sigma$ et $\neg A'\sigma \vee C'\sigma$, et on applique un démonstrateur propositionnel sur l'ensemble des clauses, plus efficace que la résolution propositionnelle.)

Cette analyse étant faite, il restait à trouver une procédure de recherche de preuve automatique qui cherche à explorer au mieux cet arbre. C'est ce qui a été fait dans [Gou94g], qui est la version publiée de la méthode de preuve déjà exposée au chapitre 6 de ma thèse [Gou93b]. Le procédé algorithmique correspondant a fait l'objet d'un brevet (numéro PCT/FR94/01089, dépôt international du 19 septembre 1994, date de priorité du 17 septembre 1993, catégories CIB6, G06F17/50, G06F9/44).

L'ensemble des techniques utilisées dans cette méthode de recherche automatique de preuve au premier ordre serait trop longue à expliquer ici. Disons seulement qu'elle repose sur deux piliers :

- l'utilisation de diagrammes de décision binaires (BDD) pour traiter le problème coNP de validité propositionnelle $\forall\rho \cdot (\rho \models \Phi\sigma)$ à chaque nœud de l'arbre de recherche;
- et le contrôle de la taille de l'arbre de recherche, et de son exploration par une heuristique fondée sur des principes de théorie de la communication [SW49] – ce que j'appelle le contrôle de l'information.

D'autre part, plusieurs techniques adaptées de techniques classiques sont utilisées :

- on peut voir, en regardant bien, la technique comme une généralisation aux BDD de la méthode de résolution linéaire avec ensemble de support;
- on utilise des graphes de connexions – dûs à Robert Kowalski – pour faciliter la recherche d'unificateurs;
- pour éviter d'explorer des branches de l'arbre de recherche redondantes, on utilise une technique adaptée de la *soustraction d'instances* due à Jean-Paul Billon (non publié), qui généralise à la fois les tests de subsomption de la résolution et l'effacement de liens dans la méthode de résolution par graphes de connexions de Kowalski.

Parlons d'abord des BDD, ensuite du contrôle de l'information.

3.2.1 BDD

Les BDD, que j'ai aussi utilisés dans [Gou94a, Gou95b] (cf. section 3.1), sont une structure très simple fournissant des représentations canoniques de formules propositionnelles. Il s'agit, en gros, de formes « si A alors formule sinon formule », représentées comme des graphes orientés acycliques : toutes les sous-formules identiques sont partagées. De plus, et assez miraculeusement, ces représentations canoniques sont étonnamment compactes, ce qui a permis en particulier de les utiliser avec un grand succès en vérification de matériel [Bry86, Bil87]. Bull S.A. a été dans les années 1980 et 1990 un des participants essentiels dans le développement des techniques à base de BDD pour vérification de matériel (l'équipe de Christian Berthet, Olivier Coudert et Jean-Christophe Madre, sous la direction de François Anceau), et c'est de là que je tiens mon initiation aux BDD. Dans la méthode de [Gou94g], les BDD servent principalement à décider de la validité propositionnelle de $\Phi\sigma$: ce sont des codages en espace des arbres à l'intérieur des boîtes de la figure 3.1.

Dans [GP94], en plus de rappeler ce que les BDD sont et comment on les utilise en calcul propositionnel, Joachim Posegga, de Karlsruhe, et moi explorons les différentes façons que l'on connaît actuellement d'adapter les BDD au premier ordre. Les BDD sont une structure très riche, et l'on peut, selon la façon qu'on a de les voir, trouver différentes méthodes de preuve. La méthode de Joachim Posegga est dans l'esprit des méthodes de tableaux [Pos93], ma première tentative d'utilisation des BDD [Gou93d] était elle aussi dans le style de la méthode des connexions, et ma seconde [Gou94g] est beaucoup plus proche de la résolution. Quand je dis qu'une méthode à BDD est proche d'une méthode plus connue, c'est pour fixer les idées, et en fait les méthodes en question sont en général très éloignées : dans le cas de [Pos93, Gou93d], la parenté avec les méthodes de tableaux se limite au fait qu'on cherche à aboutir à une preuve en fermant, ou en éliminant, des chemins aboutissant à faux (chemins, au sens des tableaux; chemins verticaux, dans la méthode des matings), alors que dans [Gou94g], je cherche à éliminer des sous-formules atomiques A, ou de façon équivalente, à combiner des ensembles de chemins aboutissant à vrai (clauses, en résolution; chemins horizontaux, dans la méthode des matings) sur une formule A.

Jan Friso Groote, d'Utrecht, nous a fait part en 1995, à Joachim Posegga et à moi, de sa découverte d'une autre technique [Gro95]; si l'on adopte le moyen de comparaison employé ci-dessus, il s'agit essentiellement d'une adaptation et d'une généralisation de la stratégie d'hyperrésolution positive aux BDD. Nous avions prévu d'écrire un article récapitulant l'état de l'art sur les BDD au premier ordre à nous trois en 1996, mais pour le moment le manque de temps de chacun, et le fait que chacun d'entre nous travaille maintenant sur des domaines différents (vérification formelle pour Jan Friso Groote, Java et télécommunications pour Joachim Posegga et protocoles cryptographiques pour moi), a retardé la concrétisation de ce projet.

3.2.2 Contrôle de l'information

Le contrôle de l'information est à mon avis l'idée la plus intéressante de l'article [Gou94g]. Je n'ai pas encore réussi à en trouver de justification mathématique formelle, et son élaboration s'est effectuée par une série d'intuitions.

La première idée est la suivante : l'arbre de recherche n'a pas une forme fixée, et selon que l'on décide de se focaliser sur tel ou tel atome à éliminer (ou tel chemin à fermer, par exemple), il y aura plus ou moins de branches à explorer dans l'espace de recherche. Une première heuristique est alors de chercher à toujours minimiser le facteur de branchement dans l'arbre, ce qui détermine sa forme. C'est en particulier la façon dont la procédure de satisfiabilité de formule propositionnelle de Davis-Putnam-Logemann-Loveland atteint son efficacité : le facteur de branchement est d'au plus deux (une variable peut être vraie ou fausse), mais on choisit toujours la variable qui minimise ce facteur de branchement (variable dans une clause unitaire, dans un littéral pur). Cette stratégie était déjà présentée, mais en cherchant à éliminer des chemins, non des atomes, dans [Gou93d]. Malheureusement, trouver un chemin qui minimise le nombre de choix possibles était difficile, et n'était probablement pas faisable en temps polynomial en fonction de la taille du BDD courant; c'est pourquoi mon attention s'est ensuite portée sur les sous-formules atomiques dans [Gou94g], ce qui du coup en a fait une méthode généralisant la résolution linéaire.

Pourquoi cette idée mérite-t-elle déjà le qualificatif de « contrôle de l'information » ? C'est que, si l'on voit un choix comme symbolisé par une lettre, un arbre de recherche est juste un ensemble de mots, les mots étant les chemins dans l'arbre. Il y a alors une relation directe entre la largeur w de l'arbre à une profondeur donnée, sa taille lorsqu'élagué à cette profondeur, et la richesse du langage ainsi défini. L'*entropie* de ce langage est $\log w$ [SW49], ce qui tend à montrer une relation entre temps de recherche et concepts de théorie de l'information. La quantité globale w est alors réduite par une heuristique minimisant localement le nombre de choix possible – ce qui n'est pas optimal est général, mais est mieux que rien.

C'est l'intuition qu'il devait y avoir un rapport entre des notions de la théorie de l'information et la complexité de la recherche dans un arbre qui m'a ensuite guidé pour trouver le reste de l'heuristique. La notion d'information ci-dessus est probablement bonne pour des problèmes NP (comme dans DAVIS-PUTNAM), mais dans le cas d'un problème Σ_2^p, il manque encore ce qu'on peut concevoir comme l'évolution de la quantité d'information stockée dans les nœuds de l'arbre de recherche (les BDD à chaque nœud, donc). Une réflexion de nature intuitive, et fondée sur quelques propriétés que l'on attend de cette quantité d'information stockée m'ont amené à la formule :

$$I(\Phi) = n_f \ \log \frac{n_f}{n_f + n_v}$$

pour la quantité d'information stockée dans le BDD Φ, où n_f est le nombre de branches menant à faux dans le BDD, et n_v est le nombre de branches menant à vrai dans ce BDD. (Ceci exprime l'information que l'on a pour qu'une branche, ou une interprétation propositionnelle, prise au hasard rende Φ faux. Thermodynamiquement parlant, il s'agit de l'entropie partielle du gaz des branches fausses dans le mélange que constitue l'ensemble des branches du BDD.)

On peut alors estimer le gain d'information lorsque l'on passe d'un nœud Φ à un nœud successeur Φ' dans l'arbre de recherche à $I(\Phi') - I(\Phi) - \log p$, où p est le nombre de successeurs de Φ, de sorte que $-\log p$ représente le gain d'information lorsqu'on particularise Φ' parmi les p successeurs de Φ, et $I(\Phi') - I(\Phi)$ est le gain d'information dû à la variation d'information stockée entre Φ et Φ'. La stratégie de parcours de l'arbre de recherche est alors de toujours choisir d'explorer, parmi les nœuds en attente d'être traités, celui dont le gain d'information depuis la racine est le plus grand.

Cette stratégie d'exploration de l'arbre de recherche est complète, parce que la combinaison des deux stratégies de minimisation du facteur de branchement et de choix du nœud avec gain maximal d'information force le gain d'information à tendre vers $-\infty$ sur toute branche de l'arbre qui ne mène à aucune preuve, alors que ce gain est fini en tout nœud de BDD réduit à vrai (qui prouve le théorème demandé). De plus, les résultats pratiques sur des jeux de test standard montrent que la procédure fait montre d'une certaine intelligence pour trouver un chemin à explorer dans l'arbre de recherche qui aboutit à une preuve, et ne rebrousse chemin qu'assez rarement. (La méthode a été codée en HimML, une extension de ML dont je parlerai en section 4.1.)

3.2.3 Et après ?

Il y a plusieurs points qu'il serait intéressant d'explorer.

Le premier point concerne les BDD. Bien qu'ils soient très efficaces sur de petites formules, ils deviennent rapidement très gros. La procédure de contrôle d'information arrive en général à limiter la taille des BDD sur le chemin choisi dans l'arbre de recherche, parce que statistiquement $I(\Phi)$ est d'autant plus grand que Φ a peu de branches fausses, et est donc en général plus petit. Mais certains exemples classiques, pourtant encore petits, demandent de fabriquer des BDD déjà très gros, que ma procédure met longtemps à construire et à analyser. Il est donc nécessaire d'explorer d'autres méthodes de preuve propositionnelle, si possible moins gourmandes en mémoire, par exemple la méthode de DAVIS-PUTNAM ou l'algorithme de STÅLMARCK [Har96]. Le point délicat est que, si l'on souhaite conserver l'heuristique du gain d'information maximal, il est nécessaire de calculer une quantité d'information dépendant de deux quantités n_f et n_v. Ces deux quantités sont, intuitivement, les nombres de branches de calcul menant à un échec, respectivement à l'acceptation, d'une machine de Turing non-déterministe. (Que l'on a représentée par un BDD jusqu'à présent.) Évaluer l'une ou l'autre est par définition un problème $\#P$-complet, qui est donc bien plus dur a priori que la résolution du problème de preuve lui-même, qui est dans Σ_2^p. (Rappelons que $P^{\#P}$ contient toute la hiérarchie polynomiale, dont Σ_2^p ne constitue que le deuxième niveau.) Comme il ne s'agit que d'une heuristique, le calcul d'une valeur approchée de la quantité d'information, par des méthodes stochastiques, devrait suffire.

Le second point concerne la justification de l'heuristique de contrôle d'information. Lorsque j'ai inventé cette heuristique, elle me semblait naturelle, et je pensais pouvoir la justifier par quelques calculs statistiques. Il semble pourtant qu'elle soit beaucoup plus difficile à justifier que prévu. Déjà, essayer de comprendre comment ce genre d'heuristique peut fonctionner sur un problème Σ_2^p-complet semble trop compliqué, tant que l'on n'aura pas réussi à comprendre pourquoi minimiser le facteur de branchement localement (ce qui suffit à faire de DAVIS-PUTNAM une procédure efficace) est bon en général. Mais alors que seuls des arguments probabilistes semblent pouvoir expliquer le phénomène, les seules lois de probabilité que l'on connaisse sur la répartition des formules Φ sont visiblement très éloignées des répartitions statistiques des problèmes courants. Pour prendre un exemple, la taille moyenne théorique d'un BDD sur n variables propositionnelles est $O(2^n/n)$, soit sa taille maximale, ce qui est en total désaccord avec la pratique. Je suis de plus en plus convaincu que l'approche la plus rationnelle et la plus convaincante est l'approche bayésienne, dont le but n'est pas d'estimer la fréquence avec laquelle l'heuristique est efficace, mais d'estimer un niveau de *plausibilité* de réussite de l'exploration de telle ou telle branche, connaissant un certain nombre de faits; je n'ai pas ici la place d'expliquer en quoi cette approche est réellement plus générale et plus intelligente que l'approche fréquentiste, et je renvoie le lecteur intéressé à l'excellent livre d'E. T. JAYNES [Jay94].

Plus important encore, si l'on arrive à comprendre pourquoi certaines heuristiques fonctionnent bien, il sera possible de les adapter à d'autres problèmes de recherche, à commencer par les problèmes de preuve au premier ordre avec égalité, ou modulo diverses autres théories.

3.3 E-unification rigide d'ordre supérieur

Je ne mentionne pas mes travaux sur l'E-unification rigide et la relation d'égalité, dont j'ai déjà parlé en section 2.2. Le pouvoir expressif de la relation d'égalité incite cependant à examiner comment on peut tenter de recoder d'autres formalismes expressifs en théories du premier ordre avec égalité. Pour la logique d'ordre supérieur, il s'agit de recoder le λ-calcul simplement typé avec β ou $\beta\eta$-conversion en une théorie du premier ordre. Une logique combinatoire est précisément l'outil qu'il faut, dans ce cas, et Daniel DOUGHERTY et Patricia JOHANN [DJ92] ont ouvert la voie.

L'utilisation de la logique combinatoire, fondée sur l'application et les deux combinateurs S et K (avec $Suvw \to uw(vw)$, $Kuv \to u$), permet de résoudre non seulement le problème de l'unification d'ordre supérieur, mais encore celui de l'E-unification d'ordre supérieur, c'est-à-dire de l'unification modulo λ et une théorie équationnelle E. Le principal apport d'un système combinatoire est d'éliminer les difficultés de traitement dues à la présence de variables liées en λ-calcul.

Les travaux de DOUGHERTY et JOHANN présentaient quelques restrictions gênantes. Notamment, le système d'équations E modulo lequel l'unification d'ordre supérieur était effectuée était contraint à être une théorie du premier ordre présenté comme un système de réécriture convergent, et seule la $\beta\eta$-équivalence était traitée. D'autre part, le système SK de logique combinatoire présente certaines défauts, dont le fait que la traduction d'un λ-terme en forme normale vers un SK-terme en forme normale peut prendre un espace exponentiel. Plus grave d'un point de vue algorithmique était le fait que les auteurs ne savaient pas reconnaître les termes flexibles des termes rigides [Hue75], ce qui est crucial pour rendre l'unification d'ordre supérieur utilisable.

J'ai tenté de résoudre ces problèmes dans [Gou94d], où l'on retrouve une partie du chapitre 10 de [Gou93b].

D'abord, à cause du fait que les traductions de λ-termes en SK-termes pouvaient nécessiter un espace exponentiel, je me suis fixé sur un autre système de logique combinatoire, dû à CURRY :

$$\begin{array}{ll} Cxyz \doteq xzy & C : (\alpha \to \beta \to \gamma) \to \beta \to \alpha \to \gamma \\ Bxyz \doteq x(yz) & B : (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma \\ Wxy \doteq xyy & W : (\alpha \to \alpha \to \beta) \to \alpha \to \beta \\ Kxy \doteq x & K : \alpha \to \beta \to \alpha \end{array}$$

Les traductions de λ-termes en CBWK-termes sont alors faisables en temps et en espace polynomial. On pourrait aussi utiliser n'importe quel autre système de combinateurs à traduction polynomiale, dont ceux utilisés par la communauté des langages fonctionnels purs [Dil88].

Ceci étant fait, le gros problème d'un système de combinateurs comme celui-ci est que la relation d'égalité $\doteq$ induite par les équations ci-dessus est plus faible que la β, resp. $\beta\eta$, équivalence. On peut corriger ceci en utilisant les *équations de* CURRY, qui sont un ensemble fini d'équations entre termes combinatoires clos. Si l'on ajoute les équations de CURRY aux équations combinatoires ci-dessus, on obtient une théorie équationnelle qui code exactement l'égalité en λ-calcul (pour la β, ou la $\beta\eta$-équivalence, c'est selon; cf. section 4 de l'article).

Les équations de CURRY posent un double problème. Premièrement, elles ne sont pas orientables facilement, ce qui ne facilite pas l'utilisation de techniques à base de complétion et de réécriture. Deuxièmement, pour pouvoir prouver une équation du λ-calcul, utiliser ces axiomes demande d'instancier un nombre assez élevé des axiomes combinatoires ($Cxyz \to xzy$, etc.)

Pour corriger le premier problème, l'idée de [Gou94d] était d'utiliser un algorithme d'E-unification rigide fondé sur la clôture de congruence, comme celui de [Gou93c]. On n'a alors plus aucun besoin d'orienter les équations, la clôture de congruence s'en passant. Par contre, il est nécessaire de créer de temps en temps de nouvelles instances des axiomes combinatoires pour éviter à la recherche d'unificateurs d'être bloquée par manque de variables libres à instancier. (Il s'agit de la procédure d'amplification des matings, dite aussi extension dans la méthode des connexions.)

La structure Union-Find servant à la clôture de congruence agit alors comme une base de données compacte d'axiomes (axiomes combinatoires, équations de CURRY, équations dans la théorie E modulo laquelle on cherche à effectuer l'unification, hypothèses $x \doteq t$ correspondant à la liaison d'une variable x à un terme t) et de lemmes démontrés à partir de ces axiomes. De plus, les équations de CURRY sont des équations sans variables : elles se codent alors dans une structure Union-Find comme une sous-structure fixe. Rappelons qu'une structure Union-Find est un automate d'arbres déterministe bottom-up; les équations de CURRY sont donc aussi un sous-automate fixé à l'avance de l'automate de clôture de congruence représentant les lemmes et axiomes équationnels courants.

Dans cet automate, les états, ou les classes d'équivalence de termes combinatoires, ne sont pas nécessairement des classes de termes en forme λ-normale. Ceci est dû au fait que l'algèbre combinatoire ne permet pas d'identifier facilement des traductions combinatoires de λ-termes en forme normale. C'est, en l'état de la technique, un inconvénient majeur, parce que l'existence et l'unicité des formes normales en λ-calcul simplement typé est un guide très fort dans la recherche d'unificateurs.

Mais, en présence d'un système d'équations E, c'est-à-dire en *E-unification* d'ordre supérieur, on ne peut pas garantir en général l'existence ou l'unicité de telles formes normales. De plus, des preuves d'existence d'unificateurs d'ordre supérieur sur des termes non nécessairement normaux peuvent être beaucoup plus courtes que des preuves restreintes à n'opérer que sur des termes λ-normaux. L'absence de propriété de normalisation ou de confluence n'était donc pas forcément un mal dans le cas général. D'un autre côté, il est stupide de vouloir ignorer délibérément des propriétés comme la normalisation, qui permettent de simplifier le problème d'unification. Dans l'automate de clôture de congruence, les états sont des classes d'équivalence de termes combinatoires u, et si l'on sait que u se réduit en v (par les axiomes combinatoires, mais aussi toutes les règles de réduction du λ-calcul), on peut ajouter à la base d'axiomes et de lemmes que code l'automate l'équation $u \doteq v$. Comme toutes les réductions du λ-calcul simplement typé terminent, on ne rajoutera ainsi qu'un nombre fini d'équations à la base d'axiomes. J'esquisse rapidement comment opérer la reconnaissance de rédex et produire l'équation correspondant à la règle de réduction en section 6.2 de [Gou94d]. Il s'agit essentiellement d'étiqueter les états de l'automate de clôture de congruence par des bits exprimant qu'à ces états sont reconnus des rédex de telle ou telle sorte.

Dans cet automate, on peut aussi étiqueter les états auxquels sont reconnus des termes flexibles, c'est-à-dire les classes d'équivalence de termes flexibles. Comme la notion de terme flexible est usuellement définie sur des formes λ-normales, il m'a fallu généraliser la notion aux termes non nécessairement en forme normale. (Il s'agit de la section 6.1 de [Gou94d].) J'appelle un terme t flexible en la variable x si et seulement s'il existe m et n tels que $t[K^m y/x]$ et $K^n y$ sont λ-convertibles (i.e., égaux modulo CBWK plus les équations de CURRY), où y est une nouvelle variable. Je définis ensuite une classe syntaxiquement reconnaissable de termes flexibles, les termes *fortement flexibles*, qui généralise la notion classique de terme flexible due à Gérard HUET, et conserve la propriété fondamentale que deux termes (fortement) flexibles sont toujours λ-unifiables. Ainsi, le processus d'unification peut-il être mis en suspens quand il vient à devoir unifier deux termes fortement flexibles. De plus, il est encore possible d'étiqueter les états de l'automate de clôture de congruence auxquels sont reconnus des termes fortement flexibles.

La technique de l'article n'est pas à mon avis utilisable du moins dans l'état. Le premier problème est que l'on ne dispose pas encore d'algorithme d'E-unification rigide (simple) efficace, et ceci conditionne l'utilisabilité des techniques de [Gou94d]. Le second problème est que les propriétés de normalisation et de flexibilité utilisées ci-dessus sont encore assez faibles, et d'autres optimisations doivent exister qui permettraient d'améliorer la méthode générale d'E-unification rigide au cas particulier du λ-calcul. Le troisième problème est que CBWK est une notation trop éloignée du λ-calcul pour pouvoir relire facilement les unificateurs trouvés, ou pour déboguer le programme d'unification, ou pour inventer des optimisations efficaces. Les systèmes de substitutions explicites, ou de combinateurs catégoriques semblent meilleurs de ce point de vue, et Gilles DOWEK *et al.* [DHK95] ont utilisé le premier système pour retrouver et généraliser l'algorithme d'unification d'ordre supérieur de Gérard HUET; en l'absence de système d'équations E modulo lequel on souhaiterait raisonner, ces derniers résolvent tous les problèmes mentionnés ci-dessus.

L'approche par traduction combinatoire est à mon avis prometteuse principalement au titre de la conception de procédures d'unification dans des systèmes de λ-calcul typés plus riches que la logique d'ordre supérieur. L'extension de [Gou94d] à la logique d'ordre supérieur avec variables de types et types de données définis comme en ML, à la logique d'ordre supérieur équipée d'un récurseur à la GÖDEL, à la logique LCF (où la normalisation échoue), ou aux systèmes TRC et TRCU de Randall HOLMES (représentations combinatoires de la théorie des ensembles NF de Willard Van Orman QUINE), est triviale. L'extension à des théories de types dépendants est nettement moins triviale, mais doit pouvoir se réduire non à une théorie équationnelle, mais à un système de clauses de Horn équationnelles (équations conditionnelles), pour lequel la théorie de la réécriture est bien comprise, et pour lequel d'un autre côté l'E-unification rigide fait place à un sous-problème de l'E-unification rigide simultanée de même complexité que l'E-unification rigide simple (cf. théorème 2.2.3).

L'avantage des techniques à base de clôture de congruence et d'E-unification rigide sur les techniques à base de réécriture est alors que les structures de données utilisées permettent un partage des termes et des lemmes déjà trouvés dans une structure d'automate déterministe. C'est un avantage similaire à celui des BDD – qui sont aussi des automates déterministes – sur les méthodes de tableaux dans le cas du problème de satisfiabilité propositionnelle. Ce n'est pas, cependant, toujours un avantage : la méthode de Davis-Putnam, qui ne partage aucun résultat non plus entre deux branches de l'exploration qu'elle mène, est parfois moins, parfois plus efficace que les méthodes à base de BDD.

3.4 Unification d'ordre supérieur ramifiée

Alors que le travail précédent portait sur une extension de l'unification d'ordre supérieur, mon travail sur l'unification d'ordre supérieur ramifiée [GL96e, GL97e] porte au contraire sur une *restriction* du λ-calcul simplement typé – le langage le plus naturel de la logique d'ordre supérieur – dans laquelle l'unification est décidable. Il s'agit de la partie *ramifiée* ou *prédicative* du langage, au sens de Bertrand Russell et Alfred N. Whitehead.

Les résultats de [GL97e] sont les suivants. Tout d'abord, et bien que la notion de ramification soit très vieille, les publications sur le sujet sont à la fois peu nombreuses et très éparses, mais surtout aucune ne *formalise* la notion de logique ramifiée (l'exception est [Laa95], qui formalise précisément les idées de Russell, et dont les préoccupations sont davantage la fidélité historique que l'élégance mathématique). Ceci explique aussi que les notions de ramification proposées par les divers auteurs – cités dans [GL97e] – soient toutes différentes. Le premier résultat de [GL97e] est donc une formalisation élégante d'une vision moderne des logiques ramifiées [Haz83], sous forme d'un λ-calcul typé avec une notion simple de sous-typage. Le deuxième résultat de cet article est l'établissement des propriétés standard de ce λ-calcul, notamment l'*auto-réduction*, et d'une propriété non standard, le *principe du cercle vicieux* de Russell. Le troisième résultat est la décidabilité de l'unification dans ce langage modulo la théorie équationnelle engendrée par les règles β et η (on peut aussi se passer de η [GL96e]), obtenue en adaptant la procédure de Gérard Huet [Hue75, SG89]. Ce langage est cependant trop faible pour construire une logique digne de ce nom, et je propose donc une extension avec des opérateurs réintroduisant une forme limitée d'imprédicativité (le langage *ramifié impur*), dans lequel malheureusement le problème de l'unification redevient indécidable : la preuve de Warren Goldfarb [Gol81] s'applique directement; l'algorithme précédent s'adapte, mais ne termine plus nécessairement. Je pense cependant que les restrictions imposées par la partie prédicative du langage forment un moyen intéressant, et mathématiquement bien fondé, pour justifier de couper des branches dans l'espace de recherche des unificateurs dans certains systèmes logiques prédicatifs, comme ceux étudiés par Solomon Feferman (FS_0 notamment [Fef91]), Harvey Friedman ou Steven G. Simpson (ACA_0 par exemple [Sim85]). Ceci est cependant encore de l'ordre du putatif.

L'idée intuitive de la ramification est que toute valeur est définie à un niveau donné, que l'on peut voir comme un instant où cette valeur est définie; et que toute fonction définie à un niveau donné ne peut s'appliquer qu'à des valeurs définies à des niveaux inférieurs (des instants antérieurs). Tout terme t est décoré non seulement par un type τ, mais aussi par un niveau ℓ entier (ou un ordinal en général), ce que je note $t : \tau/\ell$. Maintenant la fonction $\lambda x \cdot t$ est intuitivement de type $\tau_1 \to \tau_2$ si x est supposée de type τ_1 et t est de type τ_2/ℓ_2 ... ou presque : nous devons en effet

$$\frac{}{x^{\ell}_{\tau} : \tau/\ell} (Var) \qquad \frac{u : \tau_1 \xrightarrow{\ell_1} \tau_2/\ell_2 \quad v : \tau_1/\ell_1}{uv : \tau_2/\ell_2} (App) \qquad \frac{u[y^{\ell_1}_{\tau_1}/x] : \tau_2/\ell_2 \quad \epsilon = E(\tau_1) \quad y \notin \mathrm{fv}(u)}{\lambda x_{\epsilon} \cdot u : \tau_1 \xrightarrow{\ell_1} \tau_2 / \max(\ell_1 + 1, \ell_2)} (Abs)$$

(x variable ou constante)

$$\frac{u : \tau/\ell \quad \tau \sqsubseteq \tau' \quad \ell \leq \ell'}{u : \tau'/\ell'} (Cml) \qquad \frac{b \text{ base type}}{b \sqsubseteq b} (Rfl) \qquad \frac{\tau_1' \sqsubseteq \tau_1 \quad \tau_2 \sqsubseteq \tau_2' \quad l(\tau_1') \leq \ell' \leq \ell}{\tau_1 \xrightarrow{\ell} \tau_2 \sqsubseteq \tau_1' \xrightarrow{\ell'} \tau_2'} (Sub)$$

Figure 3.2: Règles de typage

tenir compte d'une part du fait que x n'est censé être remplacée que par des termes d'un certain niveau ℓ_1 à préciser, d'autre part que nous devons définir le niveau résultant de l'expression $\lambda x{\cdot}t$. Le premier problème est simple à résoudre : il suffit de demander que si $x : \tau_1/\ell_1$ implique $t : \tau_2/\ell_2$, alors $\lambda x \cdot t$ est de type $\tau_1 \xrightarrow{\ell_1} \tau_2$ – où l'on enregistre le niveau demandé pour l'argument comme une décoration sur la flèche de type –, et d'un niveau ℓ à trouver. Or ℓ doit être au moins égal à $\ell_1 + 1$ pour que la condition de ramification soit réalisée, mais aussi à ℓ_2, car la fonction $\lambda x \cdot t$ doit intuitivement être considérée comme définie après son corps t. On aboutit ainsi à la règle (*Abs*) de la figure 3.2, à quelques subtilités près (cf. [GL97e]; $E(\tau)$ est le type simple obtenu à partir de τ en supprimant tous les niveaux décorant les flèches). On suppose dans ce système que les variables sont explicitement annotées avec leur type (à la Church). Les autres règles sont relativement standard : (Rfl), (Sub) et (Cml) expriment la cumulativité des niveaux, autrement dit que tout objet existant à un niveau ℓ (défini au temps ℓ) existe aussi à tout niveau supérieur (est encore défini aux temps ultérieurs); l'ordre $\leq$ sur les niveaux s'étend naturellement en un ordre $\sqsubseteq$ sur les types, défini par (Rfl) et (Sub).

Les règles de conversion sont la β et la η-reduction, et on interprète les termes modulo une règle de α-conversion de la forme : $\lambda x_{\epsilon} \cdot t = \lambda y_{\epsilon} \cdot t[y/x]$, où x et y sont deux variables du même type *simple* ϵ, autrement dit de deux types τ_1 et τ_2 éventuellement différents, mais tels que $E(\tau_1) = E(\tau_2)$. Si l'on demandait $\tau_1 = \tau_2$, le calcul ne serait pas confluent en présence de η – c'est le contre-exemple classique de Nederpelt. Cette subtilité est la raison de la forme spéciale de la règle (*Abs*). Le calcul a alors toutes les propriétés attendues : auto-réduction (si $u : \tau/\ell$ et $u \longrightarrow^* v$, alors $v : \tau/\ell$), confluence, terminaison, et vérification de type en temps polynomial (se ramenant à un problème d'existence de circuit de poids négatif dans un graphe). Le calcul a aussi une propriété moins classique :

Théorème 3.4.1 (Cercle vicieux) *Soient u_1, …, u_n des termes, $n \geq 1$, avec x libre dans un u_i, $1 \leq i \leq n$, au moins. Alors $xu_1 \dots u_n$ n'est pas typable dans le système de la figure 3.2.*

C'est la condition du cercle vicieux de RUSSELL : « whatever involves all of a collection must not be one of the collection », cf. [GL97e, Laa95].

L'unifiabilité est décidable dans ce langage, et en fait l'algorithme de HUET dans le cas non ramifié [Hue75] fournit un ensemble fini de préunificateurs. La raison intuitive que j'avais au départ – et qu'un referee au moins a eu aussi – était qu'à cause de la propriété de cercle vicieux, chaque étape de l'algorithme de Huet où une variable est liée à un terme diminue le stock de variables disponibles. Ceci est incorrect, parce que cet algorithme recrée sans cesse de nouvelles variables. Plus précisément, la procédure de Huet lie x à des objets de la forme $\lambda \overline{z} \cdot a(\lambda \overline{x_1} \cdot H_1 \overline{z}\, \overline{x_1}) \dots (\lambda \overline{x_p} \cdot H_p \overline{z}\, \overline{x_p})$, où H_1, …, H_p sont de nouvelles variables. La deuxième raison intuitive, alors, est que les nouvelles variables H_i doivent avoir un niveau inférieur à celui de x. Ceci n'est pas encore exact, mais c'est presque ça; c'est une notion plus compliquée, celle de *min-niveau* [GL97e], qui est nécessaire. Le min-niveau des H_i est toujours strictement inférieur à celui de

x, et donc la profondeur de l'arbre de recherche de pré-unificateurs est finie. Comme cette arbre est de branchement de type fini, par le lemme de König, l'arbre lui-même, donc l'ensemble complet de pré-unificateurs trouvés est fini.

Je ne connais pas la complexité de l'algorithme, mais il semble qu'à niveaux bornés par une constante donnée, l'algorithme soit en temps polynomial. Ceci reste à prouver. De plus, si les niveaux ne sont pas bornés et écrits en unaire, la complexité est exponentielle au moins.

Je n'ai pas consacré beaucoup de temps à ce problème, parce que le langage ramifié a un défaut plus grave : il est impossible de fabriquer un langage logique utile quelconque sur cette base. Plus précisément, la construction de la logique d'ordre supérieur par Church au-dessus du λ-calcul simplement typé se traduit ici en un langage logique d'un pouvoir expressif ridicule : si l'on code $A \wedge B$ par l'application de la constante $\wedge$ sur A et B, alors $(A \wedge B) \wedge C$ n'est pas typable ! C'est le théorème 3.4.1, avec $x = \wedge \ldots$

Je propose donc d'étendre le langage avec des opérateurs f, comme en logique du premier ordre, donnés avec une signature $\tau_1 \times \ldots \times \tau_m \Rightarrow \tau$, et d'étendre les règles de typage par :

$$\frac{\begin{array}{c} u_1 : \tau_1/\ell_1 \quad \ldots \quad u_m : \tau_m/\ell_m \\ f \text{ de signature } \tau_1 \times \ldots \times \tau_m \Rightarrow \tau \end{array}}{f(u_1, \ldots, u_m) : \tau/\max(l(\tau), \ell_1, \ldots, \ell_m)} \,(Alg)$$

On peut désormais construire un langage logique raisonnable d'ordre supérieur au-dessus du calcul ramifié, étendu avec des opérateurs – le calcul ramifié *impur* –, mais la procédure d'unification ci-dessus étendue au cas impur ne termine plus nécessairement, car on peut maintenant lier les variables x à des termes de la forme $\lambda\overline{z} \cdot f(\lambda\overline{x_1} \cdot H_1\overline{z}\ \overline{x_1}, \ldots, \lambda\overline{x_m} \cdot H_m\overline{z}\ \overline{x_m})(\lambda\overline{x_{m+1}} \cdot H_{m+1}\overline{z}\ \ \overline{x_{m+1}}) \ldots (\lambda\overline{x_p} \cdot H_p\overline{z}\ \ \overline{x_p})$, où le min-niveau de H_{m+1}, $\ldots$, H_p est toujours strictement inférieur à celui de x, mais où celui de H_1, $\ldots$, H_m est inférieur *ou égal* à celui de x. En fait, l'unification dans le cas ramifié impur est indécidable, et ce déjà à l'ordre 2, avec un opérateur binaire et deux constantes : la preuve de GOLDFARB [Gol81] s'applique directement.

Néanmoins, il semble que l'algorithme d'unification dans le cas ramifié impur ait davantage de chances de terminer que dans le cas simplement typé, et qu'il s'agisse d'une brique de base intéressante pour la preuve automatique dans des systèmes plus expressifs que la logique du premier ordre, mais quand même moins expressifs – donc a priori plus praticables – que la logique d'ordre supérieur. Un bon candidat pour cela est la logique qui sous-tend un sous-système faible de l'arithmétique de PEANO du second-ordre comme ACA_0, où l'axiome de compréhension $\exists X \cdot \forall x \cdot Xx \Leftrightarrow \Phi(x)$ (X du second ordre, x du premier ordre) est restreint aux formules Φ arithmétiques. ACA_0 est suffisamment expressif pour y développer une grande partie des mathématiques, et est en particulier nettement plus expressif que l'arithmétique de PEANO du premier ordre. Je suggère dans [GL97e] que le λ-calcul ramifié impur est un bon substrat pour coder ACA_0, en particulier l'axiome de compréhension restreint se traduit simplement par une contrainte de typage sur l'expression $\lambda x \cdot \Phi(x)$. Les autres axiomes de ACA_0 se codent de façon naturelle, mais je n'ai pas prouvé que l'axiomatisation que j'en propose définit exactement ACA_0.

D'autres questions ouvertes sont, d'abord, de savoir si l'on peut recoder d'autres systèmes faibles – mais suffisamment forts en pratique – de la logique ou de l'arithmétique d'ordre supérieur. D'une part des systèmes plus forts, comme Π^1_0-CA_0, mais aussi d'autre part des systèmes plus faibles comme FS_0 [Fef91], dont le pouvoir expressif est celui de l'arithmétique primitive récursive. Étant donné le caractère prédicatif des restrictions des axiomes de compréhension de ces systèmes, cela semble probable. Il est aussi possible qu'il soit nécessaire d'explorer des notions différentes de prédicativité pour ces cas.

3.5 BDD et logiques non-classiques

Revenons à des logiques moins expressives. Le rapport [GL96a] s'intéresse à la généralisation des

BDD – que j'avais utilisés en logique classique, cf. sections 3.1 et 3.2 – aux cas des logiques propositionnelles non classiques, et en particulier à la logique propositionnelle intuitionniste.

La difficulté principale est qu'un BDD est une structure *essentiellement* classique, car fondée sur l'unique règle simplificatrice $(A \Rightarrow F) \wedge (\neg A \Rightarrow F) \longrightarrow F$. Je propose dans [GL96a] d'utiliser un cousin proche des BDD, les *ZBDD* (*zero-suppressed BDD*) [Min93], pour représenter des tableaux de façon compacte. Je rappelle qu'un tableau est un ensemble de chemins, autrement dit de séquents; si la façon traditionnelle d'utiliser la méthode des tableaux est de développer chaque chemin dans le tableau indépendamment, on peut aussi représenter le tableau tout entier, et pas seulement une de ses branches, dans une structure représentant un ensemble (le tableau) d'ensembles (les chemins) de formules signées. Les ZBDD sont une structure informatique particulièrement compacte permettant de représenter des ensembles d'ensembles.

L'hybride résultant, que j'ai appelé *TDD* (*Tableau Decision Diagrams*) est donc non pas une nouvelle méthode de recherche de preuves, mais plutôt une nouvelle technique de réalisation des systèmes de tableaux (ou de séquents) usuels. Cette technique a des avantages et des inconvénients. D'un côté, elle est forcée de représenter des tableaux tout entiers dans une même structure de données, au lieu de n'avoir à représenter que des chemins isolés comme dans les réalisations conventionnelles. D'un autre côté, le partage de toutes les branches dans la structure de TDD permet de gagner en efficacité : si un TDD a n nœuds et contient N chemins (représentant un tableau de N séquents; en général $n \ll N$), la plupart des opérations sur le TDD prendra un temps polynomial, et souvent linéaire en le nombre n de nœuds alors que les mêmes opérations itérées sur chaque chemin dans le tableau prendrait un temps proportionnel à N. Le partage fait a priori gagner un facteur $O(N/n)$ en vitesse.

Le rapport [GL96a] développe la technique des TDD et présente une étude de cas approfondie portant sur un système de séquents à conclusions multiples pour la logique intuitionniste propositionnelle [Dyc92]. Une réalisation informatique en HimML a été effectuée pour évaluer l'efficacité de l'approche, mais les résultats pratiques sont mitigés : bien que le prouveur réalisé soit efficace, il n'est pas nettement plus efficace que les réalisations traditionnelles du système LJT de Roy DYCKHOFF, par exemple. Ce dernier système est un système logique plus intelligent que celui sur lequel j'ai fondé le prouveur, mais se prête aussi moins bien à une exploitation du partage dans les TDD. Il n'est donc pas clair qui est le gagnant en termes d'efficacité, de l'intelligence logique (LJT) ou de l'astuce informatique (TDD), et il faudrait mener des expériences plus poussées, de codage de LJT en TDD en premier lieu, mais aussi sur diverses autres logiques non classiques – modales, notamment – et voir si les TDD offrent un avantage en pratique ou non. Un point en défaveur des TDD (mais aussi des méthodes des sections 3.1 et 3.2) est que l'on n'obtient pas de terme de preuve en sortie, contrairement aux méthodes de tableaux classiques; autrement dit, une réponse positive ne peut être prise que comme un argument convaincant de validité d'un énoncé que si le prouveur lui-même est correct, ce qui est d'autant moins probable que le prouveur est complexe.

3.6 Logique du temps linéaire

L'idée que Peter H. SCHMITT et moi avons eue dans le cadre de la logique du temps linéaire (LTL) [SGL97b, SGL97a] est exactement à l'opposé. Les preuves fournies par cette méthode sont remarquablement lisibles. D'autre part, aucune astuce spéciale n'a été prise pour que la méthode de preuve soit efficace, et pourtant elle semble l'être en pratique. Le but de cette méthode était en fait de trouver une méthode de test de satisfiabilité la plus simple possible pour LTL, celles qui étaient connues jusqu'alors étant relativement complexes.

Nous nous sommes d'abord intéressés à un fragment simple de LTL, LTL_0 [SGL97b], dont le problème de satisfiabilité est connu pour n'être que NP-complet – il s'agit du fragment dont les seuls opérateurs temporels sont $\Box$ et $\Diamond$, $\Box F$ signifiant « à partir de maintenant, toujours F » et $\Diamond F$ « à un moment dans l'avenir, F » –, par opposition à la logique complète – avec opérateurs $\bigcirc$ (next), $\mathcal{U}$ (until), $\mathcal{W}$ (waiting-for) – qui est PSPACE-complète [SC85].

La procédure de [SGL97b] est simple, et procède en premier lieu à une décomposition de la formule dont on cherche un modèle, par des moyens sémantiques clairs, en une disjonction de branches, lesquelles branches sont des conjonctions de formules signées et de contraintes sur des constantes de temps (qui représentent des instants); elle cherche en second lieu à trouver un modèle de chaque branche tour à tour. La formule de départ est satisfiable si et seulement si au moins une branche a un modèle.

Au lieu de décrire en détail la méthode ici, je vais juste illustrer quelques-unes de ses règles les plus saillantes. Pour chercher si une formule F de LTL_0 est satisfiable, on initialise le tableau à une branche unique, contenant juste la formule signée $[c_0]F$, où c_0 est une constante de temps arbitraire (un symbole neuf) signifiant que l'on cherche un modèle de la formule F à un certain instant c_0 – l'intervalle $[c_0]$, qui est une abréviation de $[c_0, c_0]$ est le *signe*, c'est-à-dire ici une représentation de l'ensemble d'instants auquel on souhaite voir F réalisée. Les formules signées sont en général d'un format plus compliqué. Par exemple, $[s]\,\Box F$ se décomposera en la formule signée équivalente $[s, +\infty[F$, d'où la nécessité de signes qui soient des intervalles semi-ouverts $[s, +\infty[$. Ensuite, $[s, +\infty[F \vee G$ n'est pas équivalente à la disjonction de $[s, +\infty[F$ et de $[s, +\infty[G$, difficulté que nous contournons en nous autorisant à voir un *multi-ensemble* de formules derrière le signe; $[s, +\infty[F \vee G$ se décompose alors en la formule signée équivalente $[s, +\infty[F, G$. Nous aurons donc des formules signées de la forme $I\ C$, où I est un intervalle fermé $[s, t]$ ou semi-ouvert $[s, +\infty[$, et C est un multi-ensemble de formules. Les instants s et t sont des expressions symboliques simples représentant des instants. $I\ C$ est satisfaite dans une interprétation si et seulement si, pour tout instant i dans l'intervalle I (ou plutôt dans l'intervalle, valeur de I dans l'interprétation), au moins une formule de C est satisfaite en i.

Voici par exemple la règle de tableau permettant de décomposer une formule signée de signe un intervalle $[s, +\infty[$ et ayant un multi-ensemble de formules contenant une formule $\Box F$:

$$\frac{[s,+\infty[C, \Box F}{\begin{array}{c|c} [s,+\infty[C & \begin{array}{c} [s, u-1]C \\ [u, +\infty[F \\ s \leq u \\ u \text{ nouvelle constante} \end{array} \end{array}}$$

où C est le multi-ensemble des autres formules, la virgule dénotant l'union multi-ensemble. Cette règle s'interprète comme suit : si (la disjonction de) C ou $\Box F$ est vrai sur l'intervalle $[s, +\infty[$, alors soit C est vrai sur tout $[s, +\infty[$ (branche de gauche), soit il existe un plus petit instant $u \geq s$ où C devient faux; dans ce deuxième cas C est vrai de s à $u - 1$ compris, et $\Box F$ est vraie en u, donc F est vraie sur $[u, +\infty[$ (branche de droite). La réciproque est vraie : si l'une des branches filles est satisfaite, alors $[s, +\infty[C, \Box F$ est satisfaite : la règle est inversible.

On a en réalité besoin d'une troisième classe de formules signées, notées $|\infty|\,C$, et exprimant qu'infiniment souvent, toutes les formules de C sont vraies en même temps (C est donc une conjonction et non une disjonction, parce qu'ici c'est $|\infty|\,F \wedge G$ qui n'est pas équivalent à $|\infty|\,F$ et $|\infty|\,G$). Cette classe de formules apparaît dans la règle suivante :

$$\frac{[s,+\infty[C, \Diamond F}{\begin{array}{c|c|c} [s,+\infty[C & \begin{array}{c} [u]F \\ [u+1, +\infty[C \\ s \leq u \\ u \text{ nouvelle constante} \end{array} & |\infty|\,F \end{array}}$$

qui s'interprète comme la précédente, sauf qu'ici u est le *plus grand* instant où C devient faux (branche du milieu) si un tel u existe. La branche de droite exprime le cas où C finit par être faux, et où l'ensemble des u où C est faux n'est pas borné; alors cet ensemble de u est infini, et donc infiniment souvent $\Diamond F$ doit être vraie, c'est-à-dire que F elle-même est infiniment souvent vraie. Encore une fois, cette règle est inversible. En fait, toutes les règles de décomposition de [SGL97b] sont inversibles, ce qui autorise n'importe quelle stratégie de décomposition des formules signées, et donc une très grand liberté dans la recherche de modèles.

Toutes les règles de décomposition produisent des formules signées plus petites dans l'ordre multi-ensemble sur la taille des formules; donc n'importe quelle suite maximale d'applications de ces règles termine. Et elle termine nécessairement sur des branches dites *atomiques*, c'est-à-dire ne contenant – à part une liste de contraintes temporelles de la forme $s \leq t$ – que des formules signées de la forme $I\ C$ ou $|\infty|\ C$, où C est une *clause*, c'est-à-dire un multi-ensemble de littéraux, les littéraux étant les formules atomiques A ou leurs négations $\neg A$.

La deuxième partie de la méthode de [SGL97b] consiste à décider de la satisfiabilité des branches atomiques, puisque l'atomicité ne fait pas perdre de généralité. Tout d'abord, il existe des cas simples où l'on peut immédiatement conclure à l'insatisfiabilité de branches, même non atomiques : lorsque l'ensemble des contraintes temporelles sur la branche est contradictoire (ce qui se décide très efficacement par l'existence d'un circuit de poids négatif dans un graphe [SGL97b]), lorsqu'on obtient une formule signée de la forme $[s, +\infty[\epsilon$, où ϵ est le multi-ensemble vide, ou $|\infty|\ C, F, \neg F$. On peut aussi immédiatement simplifier les branches en remplaçant les formules signées de la forme $[s,t]\epsilon$ par la contrainte $s \geq t+1$, et en éliminant celles de la forme $[s,t]C, F, \neg F$, $[s, +\infty[C, F, \neg F$ ou $|\infty|\ \epsilon$. À ce point, on peut se demander si cela suffit : la réponse est non, comme en atteste la branche atomique $[s, +\infty[A$, $|\infty|\ \neg A$, à laquelle on ne peut appliquer aucune des règles énoncées jusqu'ici. C'est pourquoi on ajoute une règle exprimant qu'une branche qui contient à la fois une clause de la forme $[s, +\infty[C$ et une clause $|\infty|\ C', \neg C$ est contradictoire, où $\neg C$ est l'ensemble des littéraux négations de ceux de C.

Ceci ne suffit toujours pas : on ne peut toujours pas détecter l'insatisfiabilité de $[s, +\infty[A$, $[t, +\infty[\neg A$. C'est pourquoi nous cherchons à montrer dans [SGL97b] que la branche est insatisfiable en saturant les branches par une forme modifiée de résolution [Rob65, CL73], que l'on peut – abusivement – représenter par :

$$\frac{I\ C, F \qquad I'\ C', \neg F}{(I \cap I')\ C, C'}$$

où les prémisses ne sont plus remplacées par la conclusion, comme dans les règles de tableaux présentées jusqu'ici, mais où la conclusion est ajoutée à la branche courante, les clauses étant elles-mêmes vues comme des ensembles et non plus comme des multi-ensembles de littéraux. Cette notation est abusive, parce que le langage des intervalles dont on dispose n'est pas stable par intersection : la notation $I \cap I'$ est une figure de style. En réalité, selon les imbrications possibles des intervalles I et I', l'intersection de I et de I' peut prendre jusqu'à quatre formes possibles (si $I = [s,t]$ et $I' = [s',t']$, alors $I \cap I'$ peut être $[s,t]$, $[s,t']$, $[s',t]$ ou $[s',t']$ selon que $s \leq s'$ ou non, et que $t \leq t'$ ou non), donnant ainsi naissance à quatre branches dans le pire des cas.

Ce sont toutes les règles dont nous avons besoin, et la complétude de la procédure est presque triviale. L'idée de la preuve est que, pour tout instant $i \in \mathbb{N}$, on peut projeter l'ensemble des formules signées sur l'instant i, ce qui donne un ensemble de clauses propositionnelles); la complétude des règles ci-dessus provient alors simplement de la complétude de la résolution propositionnelle. (Il s'agit d'une nouvelle version de la technique de relèvement [CL73].) Le seul aspect technique est le traitement des formules signées $|\infty|\ F_1, \ldots, F_n$, qui se projettent en n clauses unitaires $F_1, \ldots, F_n$ en chaque instant i de l'ensemble infini où la conjonction $F_1, \ldots, F_n$ est satisfaite.

Ces règles terminent, du moment que l'on considère les branches et les clauses comme des ensembles, mais peuvent fabriquer un nombre exponentiel de branches de taille exponentielle. On obtient donc un algorithme décidant LTL_0, mais un qui est potentiellement inefficace, sachant que la satisfiabilité en LTL_0 est un problème NP.

À la question de l'efficacité, il y a deux réponses, selon que l'on est intéressé par la théorie (les classes de complexité) ou par la pratique. Quelques modifications simples de la méthode, décrites dans [SGL97b], suffisent à montrer que la satisfiabilité de LTL_0 est effectivement dans NP; ces modifications ont cependant peu d'intérêt pratique, dans la mesure où elles demandent un effort de divination de la part de la procédure qui, s'il reste faisable en temps polynomial, force la procédure à prendre un temps qui est toujours de l'ordre du cas le pire.

En pratique, quelques autres modifications simples permettent d'utiliser des raffinements standard de la résolution [CL73], notamment la résolution unitaire et l'hyperrésolution. Si elles ne permettent pas d'établir des bornes de complexité intéressantes, elles s'avèrent en revanche faciles à coder; d'autres modifications permettent aussi de tenir compte de certains cas particuliers fréquents dans les règles de décomposition de formules signées. Au total, les expériences menées sur 46 formules – dont 22 prouvables, c'est-à-dire de négation insatisfiable – à l'aide d'une réalisation semi-naïve en HimML montrent que la combinaison des règles de décomposition, de simplification et de résolution est en réalité extrêmement efficace. Sur une machine relativement lente (une SparcStation ELC sans disque), aucune formule ne demande plus de 100 ms. pour être prouvée ou pour être montrée satisfiable. En particulier, la formule de Dummett :

$$\Box(\Box(p \Rightarrow \Box p) \Rightarrow p) \Rightarrow \Diamond(\Box p) \Rightarrow p$$

est prouvée en 50 ms., par un tableau qui ne contient que 3 branches, et ce malgré sa complexité logique – nombre et niveau d'imbrication des modalités – relativement élevée.

Ceci mène au slogan : « les problèmes informatiques, aussi complexes (possiblement indécidables) qu'ils soient en théorie, sont simples en pratique ». La difficulté est de comprendre en quoi ils sont simples. Alors que la méthode de preuve au premier ordre de la section 3.2 cherchait à découvrir cette simplicité par des heuristiques élaborées de calcul d'entropie, durant la recherche de preuve, ici c'est l'architecture de la méthode de preuve elle-même qui est bien adaptée aux formules rencontrées en pratique.

Au total, l'article [SGL97b] présente une technique simple, efficace, dont la correction et la complétude sont prouvées par des arguments quasi-immédiats, et qui est donc très satisfaisante. De plus, les preuves de formules sont lisibles, ce qui n'est pas négligeable, que ce soit dans un but éducatif, ou dans l'industrie où, de plus en plus, les clients ayant des besoins en méthodes formelles demandent aussi à avoir des assurances quant aux méthodes de preuve, en particulier la production d'une preuve vérifiable par machine et lisible par un expert.

Les applications de LTL_0, qui est un langage assez restreint, sont relativement peu nombreuses. La logique de Unity [CM89] est essentiellement une version au premier ordre de LTL_0, à laquelle on pourrait donc appliquer les techniques de [SGL97b]. Mais la vérification de protocoles a souvent besoin des opérateurs $\bigcirc$, $\mathcal{W}$, $\mathcal{U}$, ainsi que des opérateurs du temps passé $\boxminus$, $\diamondminus$, $\ominus$, $\tilde{\ominus}$, $\mathcal{S}$, $\mathcal{B}$ [MP91], qui sont présents en LTL mais pas en LTL_0. L'article en préparation [SGL97a] étend la méthode ci-dessus à LTL tout entière. La principale difficulté est d'arriver à décider de la satisfiabilité de branches atomiques (la notion étant légèrement étendue); dans ce dernier article, je propose de l'accomplir en utilisant des techniques de vérification de modèles (model-checking) à base de BDD, inspirées de [BCMM+92], et en utilisant quelques astuces pour éviter l'explosion des BDD dans des cas où cette explosion est prévisible. Le codage de cette méthode est quasiment effectué, mais est en suspens depuis que je suis revenu de Karlsruhe, où j'ai effectué ce travail. Le défaut de cette approche comparée à l'approche par résolution est que l'on perd en grande partie la lisibilité des preuves obtenues. Je pense néanmoins que l'on devrait pouvoir étendre l'approche par résolution pour s'adapter à ce cas : ceci reste à explorer.

Chapitre 4

Langages fonctionnels

Comme je l'ai dit dans l'introduction, l'une de mes premières tâches chez Bull était de réaliser un outil de prototypage rapide pour VDM. Le principal problème était de coder des opérations ensemblistes de façon raisonnablement efficace. D'autre part, la pratique de VDM m'avait convaincu du confort qu'offrait une écriture de programmes en style ensembliste, ce qui m'a mené à concevoir et à réaliser un interprète pour une extension du langage ML avec des types et des structures de données ensemblistes, que j'ai appelée HimML. Les programmes que j'ai écrits pour tester les idées des sections 3.1, 3.2, 3.5 et 3.6 notamment l'ont été en HimML.

En section 4.1, je parle de la technique de réalisation particulière des ensembles en HimML. En section 4.2, c'est d'une idée dans le domaine de la compilation de langages à allocation mémoire implicite, celui de l'optimisation des opérations d'emboîtement (allocation) et de déboîtement (lecture d'un champ de la structure allouée) dans le code produit par un compilateur, que je traite. Finalement, je ferai mention en section 4.3 d'une curiosité : il est facile de faire vérifier des équations aux dimensions telles qu'utilisées par les physiciens, et même de les inférer, par une extension simple du système de typage de ML qui a été codée en HimML.

4.1 Hash-consing et ensembles en ML

HimML est une extension du langage ML qui inclut des types ensembles (τ `set` est le type des ensembles finis d'objets de type τ) et des types mappes (τ `-m>` τ' est le type des *mappes*, c'est-à-dire des ensembles finis d'associations $x \mapsto y$ d'objets x de type τ vers des objets y de type τ'), ainsi que des primitives et des constructions syntaxiques permettant de construire des expressions et des patterns ensemblistes. J'ai commencé à réaliser l'interprète HimML en décembre 1991. J'ai décrit le principe de la réalisation, avec une analyse théorique de l'efficacité et quelques premiers tests expérimentaux dans [Gou93e]. Le rapport [Gou93a] fournissait une étude plus poussée de l'efficacité des techniques, et le court article [Gou94e] fait une synthèse des techniques, des résultats théoriques et des résultats expérimentaux.

Les deux techniques fondamentales sont, premièrement, l'utilisation de *hash-tries* pour représenter les mappes (et les ensembles, τ `set` étant considéré comme une abréviation de τ `-m> unit`); et secondement, la technique de *hash-consing*, qui permet de partager deux objets structurellement égaux. Cette dernière technique, inventée par Ershov en 1957, est aussi la technique de base de réalisation des BDD, et est généralisée à tous les objets du langage – du moins ceux sur lesquels la fonction d'égalité a un sens. Ainsi, les objets du langage sont en bijection avec leurs adresses, et l'on peut utiliser cette adresse comme fonction de hachage pour organiser les hash-tries; ces dernières sont alors des formes canoniques des mappes, que l'on peut ainsi identifier de manière unique par leur adresse à leur tour.

La complexité théorique des fonctions du langage est étudiée dans [Gou93e]. Le test d'égalité s'opère en temps constant, quelle que soit la complexité des structures à comparer, y compris des ensembles, des ensembles d'ensembles, etc. La complexité dans le cas le pire des opérations ensemblistes n'est pas bornée, mais les complexités moyennes sont parmi les meilleures des réalisations de bibliothèques ensemblistes à usage général, et les écarts-types sont très faibles. Rechercher si un élément est dans un ensemble, ajouter un élément à un ensemble, supprimer un élément d'un ensemble s'effectuent en temps moyen $O(k \log n)$, où n est le cardinal de l'ensemble, et k est le temps d'allocation d'un élément mémoire de base (une cellule à deux champs, comme en Lisp). Les opérations binaires courantes – union, intersection, différence, différence symétrique, etc. – s'effectuent en temps moyen proportionnel à k fois le minimum des cardinaux des deux ensembles à combiner, plus une correction d'ordre logarithmique.

L'efficacité pratique dépend donc en premier lieu du temps k d'allocation d'une cellule de base, et ce en comptant le temps de partage : allouer une cellule contenant deux champs a et b demande de rechercher d'abord s'il en a déjà été construite une, et sinon d'allouer réellement un couple, enfin d'enregistrer que (a, b) a été construit. Cet enregistrement, et la recherche préalable, se font par la maintenance et la consultation d'une table de hachage globale. (Ceci constitue la technique de hash-consing.) De façon surprenante, j'ai constaté que l'allocation avec partage était rarement plus de 2 fois plus lente que l'allocation usuelle de cellules à partir d'une liste libre, et ce en utilisant une table de hachage non extensible avec listes de collisions dans chaque entrée. Ceci est le résultat d'un programme de test préliminaire ne testant que l'allocation. Les résultats sur l'interprète HimML et des exemples de programmes de test faits pour CAML [Gou93a, Gou94e], donc pas spécialement adaptés à HimML, montrent que le ralentissement dû à la complication de l'algorithme d'allocation est rarement perceptible, et qu'en fait les gains dûs notamment à l'accélération de la fonction d'égalité compensent, parfois largement, le ralentissement de l'allocation. Ces gains sont surtout perceptibles dans les programmes de manipulation symbolique – comme un démonstrateur de théorèmes – ce qui fait de HimML un interprète de choix pour réaliser des programmes comme ceux du chapitre 3.

L'efficacité pratique de la technique d'allocation, et donc de la technique de représentation des ensembles, qu'elle conditionne, dépend aussi de l'efficacité de la récupération mémoire (*garbage collection*). L'utilisation d'une table de hachage globale pose un problème de ce côté : tout objet libéré doit être supprimé de la table de hachage. Les techniques de récupération par marquage et récupération (mark-and-sweep) s'adaptent facilement, mais les méthodes par copiage (stop-and-copy) ne fonctionnent pas, puisque l'identité d'un objet est lié à son adresse en mémoire. Cependant, les techniques de récupération par générations (*generation scavenging*) s'adaptent au mark-and-sweep, comme indiqué dans [Gou94e], ce qui permet à l'interprète HimML de ne passer que 8 à 10 pour cent du temps en récupération mémoire – chiffre qui s'est amélioré et est passé aux alentours de 3 à 7 pour cent, depuis que l'interprète HimML est devenu un interprète de bytecode.

La technique de hash-consing systématique, servant à la réalisation par hash-tries de structures ensemblistes efficaces, pose quelques exigences sur la sémantique du langage et sur la gestion de la mémoire à l'exécution. En ce qui concerne cette dernière, c'est la raison pour laquelle j'ai codé l'interprète HimML en C, au lieu de réutiliser et d'étendre une réalisation existante de ML. En ce qui concerne la sémantique, elle dicte que seuls les objets immutables peuvent être comparés par contenu, les autres étant soit incomparables soit à comparer par adresse; elle dicte aussi une évaluation non paresseuse des arguments : il est nécessaire de connaître la valeur d'un objet pour pouvoir l'identifier à son adresse. C'est pourquoi HimML est une variante de Standard ML. L'adaptation à d'autres langages applicatifs, comme Lisp, Haskell ou même CAML ou CAML-light, semble plus difficile.

4.2 Emboîtements, déboîtements

J'ai commencé en octobre 1993 à écrire un compilateur pour HimML, qui devait traduire du code HimML en C. Le but était principalement d'accélérer le démonstrateur automatique de la section 3.2. La technique d'optimisation de base de ce compilateur, qui devait rester simple mais être capable d'effectuer des optimisations importantes que le compilateur C ne pourrait pas trouver par la suite, était une extension de la technique de détection des expressions communes (*common subexpression elimination*, ou *value numbering*). La technique est décrite dans [Gou94c].

La technique est adaptable à n'importe quel langage, applicatif ou non, typé ou non. Il est cependant nécessaire d'effectuer d'abord une traduction du programme à compiler en forme à affectation statique unique (*static single assignment form*, ou SSA), ce qui est plus aisé sur des langages applicatifs, mais est aussi faisable sur des langages impératifs. La technique de compilation est fondée sur une analyse de flots de données (*dataflow analysis*) en intraprocédural, qui n'a pas besoin d'information de types pour progresser. En interprocédural, l'heuristique que j'utilise, qui est proche du *call-forwarding* [DBDGK94], généralise l'intégration (*inlining*) de fonctions, et ne dépend pas non plus d'informations de types; elle n'empêche pas cependant d'utiliser des techniques utilisant les types ML, comme celles inventées par Xavier Leroy [Ler92].

La technique intraprocédurale consiste donc en une analyse de flots de données, et est une extension de la technique de détection de sous-expressions communes. En SSA, un programme est un graphe dont les nœuds sont des commandes, qui sont des nœuds d'entrée dans le code, de sortie du code, de test, de jointure (de deux branches de test), et des affectations $x := e$, où x est une variable (une variable par affectation) et e est une expression, qui peut être soit une variable y soit un terme de hauteur 1, c'est-à-dire de la forme $f(x_1, \ldots, x_n)$. Par exemple, une allocation de couple (a, b), où a est dans la variable x et b est dans la variable y est représentée par $z := cons(x, y)$; ce genre d'opération s'appelle un *emboîtement* de x et y à l'intérieur de z. Réciproquement, chercher la première (resp. seconde) composante du couple z est codé par $z' := fst(z)$ (resp. $z' := snd(z)$); ce genre d'opération, symétrique des précédentes, s'appelle un *déboîtement* de z.

Dans une expression $x := f(x_1, \ldots, x_n)$, f est l'une des primitives choisies du langage à compiler. Séparons les primitives à effet de bord des primitives f *pures*, c'est-à-dire définissant des fonctions mathématiques, ou encore telles que $\forall x_1, \ldots, x_n, y_1, \ldots, y_n \cdot x_1 \doteq y_1 \wedge \ldots \wedge x_n \doteq y_n \Rightarrow f(x_1, \ldots, x_n) \doteq f(y_1, \ldots, y_n)$ soit une équation valide du langage. Il est alors envisageable de détecter toutes les équations entre termes formés au-dessus des variables x du programme en utilisant les symboles de fonctions purs f. Ces termes sont des termes clos, les variables du programme agissant comme des constantes (il s'agit d'équations quantifiées universellement sur toutes les valeurs que peut prendre une variable du programme; herbrandiser ces quantifications conduit naturellement à considérer les variables du programme comme des constantes de Skolem). La détection de sous-expressions communes est donc la dérivation de théorèmes équationnels non quantifiés sur le langage engendré par les symboles de fonctions purs.

La détection de sous-expressions communes classique opère dans la sous-classe des équations entre variables uniquement (de la forme $x \doteq y$). Ceci permet de détecter quand une opération est inutile, parce que recalculant la même valeur qu'une valeur précédemment calculée de la même façon : si $x_1 := cons(x, y)$ et $x_2 := cons(x', y')$, et que l'on sait que $x \doteq x'$, $y \doteq y'$, alors on a $x_1 \doteq x_2$, et l'on peut réécrire la deuxième affectation en $x_2 := x_1$. L'idée de [Gou94c] est que l'on peut étendre l'algèbre des équations étudiées à une algèbre plus riche. Cette algèbre plus riche n'a pas été choisie comme l'algèbre de toutes les équations entre termes clos, qui poserait des problèmes dans le cas des boucles (le treillis des relations d'équivalence entre termes clos étant de hauteur infinie, une analyse exacte ne terminerait pas) et qui compliquerait aussi le choix des optimisations que l'on pourrait en déduire. L'algèbre des équations choisie est donc celle de toutes les équations de la forme $x \doteq y$ ou $x \doteq f(x_1, \ldots, x_n)$. Dans un code contenant par exemple $z := cons(x, y)$ puis $z' := fst(z)$, les équations déduites à partir de la première affectation sont $z \doteq cons(x, y)$, $x \doteq fst(z)$, $y \doteq snd(z)$, ce qui entraîne que $z' \doteq x$ sera déduite après la deuxième affectation, que l'on pourra alors simplifier en $z' := x$.

Ce genre d'optimisations n'est pas détectée directement par les techniques classiques de sous-expressions communes, mais est souvent nécessaire dans les langages applicatifs où tout calcul implique normalement des déboîtements, le calcul proprement dit et un emboîtement du résultat, mais où les enchaînements d'emboîtements et de déboîtements sont souvent superflus, et doivent être simplifiés.

La notion est généralisée à toutes les primitives sur lesquelles la sémantique du langage à compiler nous donne des informations équationnelles. Ceci est donc vrai non seulement pour les allocations de couples et les accès à leurs composantes comme ci-dessus, mais aussi pour les emboîtements et déboîtements de nombres en virgule flottante (qui pénalise lourdement les applications de calcul numérique), les constructions de fermetures (*closure*) et les appels de fonctions (le déboîtement correspondant), ou les constructions d'ensembles et les tests d'appartenance (un déboîtement correspondant) en HimML. Le seul défaut de la technique est qu'elle a tendance à augmenter la durée de vie des variables du programme, ce qui est gênant pour l'allocation efficace de registres; il faudrait donc étudier le problème plus sérieusement, mais pas avant d'avoir d'abord pu tester les idées de départ. Bien qu'une grande partie du code soit écrite, je n'ai pas encore trouvé le temps de le faire.

La technique interprocédurale consiste à considérer tout code de fonction comme consistant en un en-tête, dans lequel les arguments sont décomposés et testés (c'est là que se trouve le code de pattern-matching en entrée des fonctions ML), un corps, et un final, dans lequel le résultat à retourner est construit. Même si un code de fonction est jugé trop gros pour être intégré à son point d'appel, il est toujours possible et usuellement intéressant d'intégrer son en-tête (une version simplifiée de call-forwarding), son final (la version symétrique; on pourrait l'appeler *return-backwarding* par extension), et de laisser un appel à une fonction dont le code est le corps seul de la fonction de départ. Ceci permet de faire ressortir au site d'appel les opérations de déboîtement (de l'en-tête) et d'emboîtement (du final), qui peuvent ensuite être optimisées par la technique d'optimisation intraprocédurale.

Bien que je n'aie pas encore pu tester cette idée, il me semble qu'elle devrait fonctionner correctement en pratique, en donnant parfois des résultats meilleurs que la technique de Xavier Leroy, basée sur le typage du programme ML source. La grande faiblesse de mon idée, par contre, est qu'en ignorant les informations de types, elle ne peut optimiser en rien les appels d'une fonction à une fonction dans un autre module. Il me semble néanmoins que rien n'empêche d'intégrer la technique de Xavier Leroy à la mienne, au moins pour régler ce problème de compilation séparée. Encore une fois, j'attends de pouvoir tester mon idée sur une réalisation concrète pour pouvoir juger de l'intérêt de telle ou telle amélioration.

4.3 Inférence de dimensions et unités physiques

Lors de la réalisation de l'interprète HimML à partir de décembre 1991, j'ai réalisé que le système de types de ML se prêtait particulièrement bien à une extension permettant de vérifier et d'inférer automatiquement les équations aux dimensions. Par équations aux dimensions, j'entends l'interprétation abstraite effectuée couramment par les physiciens pour vérifier des équations entre quantités physiques : on peut ainsi comparer des distances entre elles, mais pas des distances avec des masses, de même que l'on peut additionner deux distances – même exprimées dans des unités différentes – mais pas additionner des distances à des masses.

Les distances, les masses sont autant de vecteurs indépendants dans l'espace vectoriel des dimensions. La somme dans cet espace vectoriel est le produit de dimensions, et le produit par un scalaire α est la mise à la puissance α d'une dimension. On peut alors étendre l'algèbre des types ML de la façon suivante. Les types sont de deux sortes possibles, la sorte des dimensions, et la sorte de tous les types, la sorte des dimensions étant incluse dans celle de tous les types. L'ensemble des variables de types est étendue par un ensemble infini de variables de dimensions `'#a`, `'#b`, `'#c`, ...de sorte dimension, les autres variables `'a`, `'b`, `'c` étant de la sorte de tous les types. Les dimensions elles-mêmes sont les produits de puissances de variables ou de constantes de dimensions, comme `distance^2 ‘ masse` – où `‘` est le symbole du produit de dimensions –, ou comme `'#a^2.5 ‘ masse^ ~1 ‘ '#b^0.4`. Les constantes de dimensions sont autant de vecteurs indépendants dans l'espace vectoriel des dimensions, et une construction `quantity` permet de créer une nouvelle constante de dimension. Les dimensions, vues comme des vecteurs, sont alors des combinaisons linéaires de vecteurs de base et de variables à valeurs vectorielles. Les types ML sont les variables de types, les constantes de types, les dimensions, ou les applications de fonctions de types (`->`, `*`, etc.) à d'autres types.

L'unification dans la nouvelle algèbre de types, qui est une algèbre à sortes ordonnées modulo la théorie équationnelle des espaces vectoriels, est encore faisable en temps polynomial, et possède la propriété que tout problème d'unification a au plus une solution la plus générale, comme l'unification syntaxique. La résolution d'un problème d'unification entre dimensions est en effet la résolution d'une contrainte d'égalité entre combinaisons linéaires sur des vecteurs de base et des variables, ce qui se résout trivialement en exprimant une des variables comme la combinaison linéaire appropriée des vecteurs de base et des autres variables, lorsque c'est possible. Le reste du système de typage de ML reste inchangé.

À l'exécution, chaque valeur de type τ, où τ est une dimension, fait partie d'un corps $\mathbb{K}$. (En général, une version approchée d'un corps, en réalité, le modèle étant celui des réels $\mathbb{R}$, ou celui des complexes $\mathbb{C}$ comme dans HimML.) L'espace sémantique des valeurs numériques est donc $\mathbb{K} \times D$, où D est l'espace des dimensions. La valeur $(1, d)$, pour chaque dimension d de D, est l'unité par défaut de la dimension d. On peut par exemple déclarer que l'unité par défaut des distances est le mètre, et déclarer des unités dérivées (le pied, par exemple, valant environ 0.31 mètre). Il est possible de déclarer d'autres unités, avec les facteurs de mise à l'échelle correspondants, et de faire faire non seulement les vérifications de dimensions par l'inférenceur de types ML, mais aussi les conversions entre unités par le compilateur.

Lorsque j'ai codé ceci en décembre 1991, je ne pensais pas que ça valait la peine d'être publié. La solution technique me semblait simple, et je n'avais pas l'impression qu'il y aurait suffisamment de gens intéressés par le sujet. Natarajan SHANKAR m'a convaincu du contraire à l'occasion de notre rencontre à l'IFIP WG 2.3 en juillet 1993 (lac Simcoe, Ontario, Canada). Je n'ai donc publié cette idée qu'en février 1994 [Gou94f]. J'y décris les principes esquissés ci-dessus, je montre comment j'ai réalisé le système en HimML – les dimensions sont codées comme des mappes envoyant les variables ou constantes de dimensions vers leurs exposants –, et j'explore les moyens d'étendre le système en présence d'un langage de modules dans le style de celui de Standard ML. L'exploitation des dimensions est raisonnablement utilisable, contrairement aux tentatives précédentes (citées dans l'article, à l'exception de celle de Mitchell WAND et Patrick O'KEEFE [WO91], dont je n'avais pas encore connaissance), et est très proche de la solution trouvée par Andrew KENNEDY [Ken94] cette même année. La différence entre mon système de typage et celui d'Andrew KENNEDY est que ce dernier se restreint à des exposants de dimensions entiers. Ceci fait que le problème d'unification qu'il a à traiter est celui de l'unification dans les groupes abéliens, qui est unitaire aussi, mais un peu plus compliqué à résoudre en pratique. Il ne traitait pas le problème de la conversion automatique d'unités dans [Ken94], ni les problèmes liés à l'intégration avec un système de modules, mais son travail s'est poursuivi jusqu'à montrer un principe d'invariance dimensionnelle des programmes avec inférence de dimensions, dans le style des résultats de John C. REYNOLDS sur les propriétés des fonctions de types polymorphes en λ-calcul typé par le système F [Ken97].

Le sujet en a intéressé d'autres. Mikael RITTRI s'est intéressé plus particulièrement au typage des définitions récursives polymorphes de fonctions à valeurs numériques, en utilisant la semi-unification modulo la théorie des groupes abéliens, respectivement des espaces vectoriels. (La semi-unification est l'outil permettant de typer les définitions récursives polymorphes de ML+, dû à Alan MYCROFT [Myc84]; ce problème est malheureusement indécidable [KTU93].) Mikael RITTRI a montré que la semi-unification modulo la théorie des groupes abéliens était décidable dans le cas d'un problème de semi-unification sans symboles de fonctions [Rit94], ce qui semble trop restrictif pour être utilisable. Par contre, il a aussi récemment montré que la semi-unification modulo la théorie des espaces vectoriels était toujours décidable [Rit95], ce qui rend le typage des définitions récursives de fonctions numériques envisageable. Plus récemment, Bruno BLANCHET [Bla96] a effectué un codage complet d'un système d'inférences de dimension et de compilation de conversion d'unités en CAML-light. Finalement, Iain STUART et Sophia DROSSOPOULOU [SD95] ont récemment proposé encore un système de type de dimensions, mais moins expressif car limité aux types monomorphes.

Chapitre 5

Logique modale S4

La logique modale S4 est l'une des logiques modales normales les plus fondamentales. J'ai commencé à m'y intéresser en juillet 1994, alors que je cherchais à me familiariser avec les logiques modales, les systèmes de séquents et la notion de correspondance de entre preuves et programmes, dite de Curry-Howard. L'intérêt informatique d'une correspondance de Curry-Howard est que les règles de déduction modale en S4 correspondent intuitivement à des règles de typage d'opérateurs d'évaluation et de citation de code, comme les opérateurs `eval` et `quote` de Lisp.

5.1 Introduction et motivations

Considérons pour l'instant une variante de S4 minimale, c'est-à-dire où les formules ont pour syntaxe :

$$F ::= b \mid F \Rightarrow F \mid \Box F$$

et définie par un système de Hilbert, dont les deux règles sont :

- (MP) de F et $F \Rightarrow G$, déduire G;
- (Nec) de F, déduire $\Box F$.

et les axiomes sont :

- toutes les tautologies propositionnelles (intuitionnistes);

(K) $\Box(F \Rightarrow G) \Rightarrow \Box F \Rightarrow \Box G$;

(T) $\Box F \Rightarrow F$;

(4) $\Box F \Rightarrow \Box\Box F$.

On peut représenter les preuves de formules n'utilisant que (MP) et les tautologies propositionnelles par des λ-termes, ou par des termes de l'algèbre combinatoire fondée sur les combinateurs S et K – ce qui est plus naturel dans un système de Hilbert.

Si l'on souhaite interpréter les autres axiomes, on commence par comprendre $\Box F$ comme étant le type de toutes les « boîtes » contenant un objet de type F. L'axiome (K), maintenant, est essentiellement une façon d'exprimer que l'on peut appliquer la règle (MP) à l'intérieur des boîtes : si u est de type $\Box(F \Rightarrow G)$ (u contient une fonction de type $F \Rightarrow G$) et v est de type $\Box F$ (v contient un argument de type F), alors l'application d'une constante $\star : \Box(F \Rightarrow G) \Rightarrow \Box F \Rightarrow \Box G$ à u et v, que je noterai $u \star v$, effectue une application de la fonction à l'intérieur de u à l'argument à l'intérieur de v, et met le résultat dans une boîte de type $\Box G$.

D'autre part, l'axiome (T) s'interprète comme le type d'un opérateur de déboîtement , (lire: « eval »), et la règle de nécessitation (Nec) est la règle de typage pour l'opération inverse, le mécanisme de *citation* ‘ (lire: « quote »). En fait, , ressemble beaucoup à la primitive `eval` de Lisp, et ‘ ressemble beaucoup à la forme spéciale `quote` de Lisp. En effet, étant donnée une expression Lisp u, `(quote` u`)` – abrégée en `'`u – est une structure de donnée qui, lorsqu'on lui applique `eval`, redonne la valeur de u. Cette structure de donnée est construite à l'aide d'opérations de listes, fondées sur la primitive `cons` de création de couple, qui est similaire à l'opérateur $\star$ ci-dessus – jusqu'à un certain point, notamment il n'y a pas d'opérateur permettant de récupérer les composantes d'une application syntaxique $u \star v$ en S4. Cette structure de donnée peut aussi contenir des représentations emboîtées de λ-expressions, c'est-à-dire de fonctions. Par exemple, l'expression Lisp `'(lambda (x) x)` devient $‘(\lambda x \cdot x)$ en S4, c'est-à-dire $‘(SKK)$ si l'on utilise une représentation à l'aide de combinateurs. D'un autre côté, on aurait pu représenter le même objet par $(‘S \star ‘K) \star ‘K$, ce qui revient à écrire l'expression Lisp `(list 'lambda (list 'x) 'x)`. Ceci suggère les règles de réduction :

- $’(‘u) \longrightarrow u$;
- $’(u \star v) \longrightarrow (’u)(’v)$;
- $‘(uv) \longrightarrow (‘u) \star (‘v)$;

Ces règles préservent les types, autrement dit ce sont des règles de transformation de preuves valides.

On peut aussi interpréter d'une façon duale les opérateurs $’$ et $‘$. Si l'on interprète $‘$ par la notation d'anticitation ` (la forme spéciale `quasiquote` de Scheme [CR91]) et $’$ par le méta-caractère , (la forme spéciale `unquote` en Scheme), alors les règles de réduction ci-dessus sont admissibles, pourvu que l'on autorise aussi la règle de réduction:

- $‘(’u) \longrightarrow u$;

En effet, `` `(...,``u `...)` est en Lisp l'objet identique à `'(...*` `...)`, où l'astérisque $*$ est remplacée par la valeur courante de u – et non le *symbole* u lui-même. Les constructions de citation ` et , obéissent intuitivement aux équations `` `,``$u = u$ et `` ,` ``$u =$, et si l'on définit $u \star v$ comme `` `(,``u `,`v`)`, alors on a aussi `,(`$u{\star}v$`) = (,`u`) (,`v`)` – qui est déductible des précédentes – ainsi que `'(`uv`) = '`$u\star$ `'`v, où `'`u `=` `` ` ``u lorsque u ne contient aucun sous-terme de la forme `,`v.

Finalement, l'axiome (4) correspond à l'opérateur `kwote` de Lisp : si $u : \Box F$, alors `(kwote` u`)`: $\Box\Box F$ et `(kwote` u`)` s'évalue en `'`u. Je vais expliquer ceci plus en détail. On peut penser à `` ` ``u non pas comme à l'opérateur ` appliqué à $u$, mais plutôt comme à un programme entièrement nouveau $u'$, qui a la propriété que `,`u' a la même sémantique que u. Un compilateur voyant l'expression `` ` ``u va ainsi produire du code qui fabrique u', et c'est effectivement ce qui se passe en Lisp : typiquement, `` `(,x ,y) `` est compilé sous forme de `(list x y)`, où `list` est l'équivalent de $\star$. L'opérateur `kwote` sert à appliquer `quote` à son argument *durant l'exécution* du programme. Et dans un cadre typé, il est naturel d'utiliser le typage pour restreindre les cas où l'application de `kwote` a un sens. La contrainte imposée par une logique comme S4 est que `kwote` ne peut s'appliquer qu'à des objets qui sont déjà des citations. Ceci n'exclut pas de fabriquer des versions citées d'autres objets, mais n'autorise pas à citer n'importe quel objet à l'exécution, notamment les objets de types fonctionnels.

Il est donc raisonnable de voir en $’$, resp. $‘$ des analogues de `eval` et `,`, resp. de ` et `'`. Par contre, les règles de réduction ne sont pas claires : doit-on autoriser $‘(’u) \to u$ par exemple ? L'ensemble des règles écrites jusqu'ici sont-elles complètes ? Si oui, dans quel sens ? Plus grave est le fait que la gestion de la portée des variables n'est pas claire. Dans une expression Lisp comme `` (lambda (x) `x) ``, le deuxième `x` n'est en fait pas lié par le `lambda`, car il ne sera évalué (par `eval`) que plus tard, et donc dans un environnement différent. Mon effort s'est donc porté dans [GL96b] sur la définition précise d'un calcul pour `eval` et `quote`; la méthode suivie est l'application du dogme de Curry-Howard, c'est-à-dire l'analyse du processus d'élimination des coupures dans la logique qui semble adaptée à la notion, ici S4.

J'ai parlé jusqu'à présent de l'analogie entre S4 et les mécanismes d'évaluation (`eval`) et de citation (`quote`) en Lisp, mais ces mécanismes doivent être compris dans un sens large, celui où la citation permet de retarder un calcul, et l'évaluation de déclencher un calcul retardé. Ceci a été remarqué par Frank Pfenning et Rowan Davies [DP96], où S4 est utilisée pour fournir un système de types pour un langage avec annotations de temps d'évaluation, justifiant ainsi la technique de *binding-time analysis* utilisée en évaluation partielle. Il semble cependant qu'une logique du temps linéaire avec unique opérateur temporel $\bigcirc$ soit parfois mieux adaptée à ce but [Dav96].

Un autre intérêt qu'il y a à étudier S4 est que, d'un point de vue logique, S4 est une version simplifiée de la logique linéaire : la modalité $\Box$ obéit aux mêmes règles de déduction que le ! de la logique linéaire, mais on n'a pas en S4 à considérer les complexités venant de l'aspect de gestion des ressources de la logique linéaire. (Ce sera le cas notamment dans le cadre du $\lambda\mathsf{ev}Q$-calcul, dont je parlerai sommairement en section 5.3.) S4 est donc aussi une passerelle vers une meilleure compréhension des exponentielles ! et ? de la logique linéaire. Différents auteurs ont déjà fait le parallèle, parmi lesquels Gavin BIERMAN et Valeria DE PAIVA [BdP92], ou Simone MARTINI et Andrea MASINI [MM96].

5.2 Élimination des coupures

Partant du dogme qu'il fallait comprendre le mécanisme d'élimination des coupures en S4 pour mieux comprendre les mécanismes d'évaluation et de citation, j'ai commencé à étudier ce processus en détail en [GL96b].

La plupart des systèmes proposés pour S4 jusqu'ici ont des défauts : ou bien l'autoréduction n'est pas respectée – les preuves peuvent se transformer en des objets qui ne sont plus des preuves, comme dans la proposition de TROELSTRA –, ou bien le calcul est incomplet dans la mesure ou il n'élimine pas tous les coupures – c'est-à-dire que les programmes correspondants peuvent bloquer sans fournir de valeur, comme chez BIERMAN et DE PAIVA –, ou le calcul peut être composé de bric et de broc, comme chez PFENNING et WONG [PW95], où la règle de calcul n'est pas une règle de réécriture.

Le rapport [GL96b] est à la fois une présentation élémentaire de la problématique et des difficultés à surmonter, et un exposé d'un calcul qui n'a aucun des problèmes mentionnés ci-dessus. Ce calcul, le λ_{S4}-calcul, est une variante du calcul de Gavin BIERMAN et Valeria DE PAIVA [BdP92], montrée en figure 5.1, lequel calcul a comme pendant le système de séquents de la figure 5.2. Le λ_{S4}-calcul simplement typé correspond naturellement aux preuves dans le système de déduction naturelle standard pour S4 intuitionniste (la figure 5.1 sans les termes de preuve).

Dans ce système, une formule est dite *emboîtée* si elle est de la forme $\Box F$, et un contexte Γ est dit emboîté s'il consiste en des liaisons $x : \Box F$ uniquement. Le langage de termes de preuve est :

$$\begin{array}{rcl} T & ::= & \mathcal{V} \mid \lambda\mathcal{V}\cdot T \mid TT \\ & \mid & \mathsf{unbox}\ T \\ & \mid & \mathsf{box}\ T\ \mathsf{with}\ T,\ldots,T\ \mathsf{for}\ \mathcal{V},\ldots,\mathcal{V} \end{array}$$

où $\mathcal{V}$ est un ensemble infini de variables, et où la partie with d'un box-terme doit avoir exactement autant de termes que la partie for a de variables, et toutes les variables de la partie for sont distinctes deux à deux. De plus, tous les termes de la forme :

$$\mathsf{box}\ u\ \mathsf{with}\ v_1,\ldots,v_n\ \mathsf{for}\ x_1,\ldots,x_n$$

doivent être tels que les variables libres dans u sont toutes parmi x_1, …, x_n, et aucune variable parmi ces dernières n'est libre dans un v_i, $1 \leq i \leq n$. L'ensemble des variables libres d'un tel terme est l'union des ensembles de variables libres des v_1, …, v_n. Un tel terme est en fait une *clôture syntaxique* au sens de [BR88], c'est-à-dire une suspension de l'exécution de u, les variables libres de u étant liées à des valeurs v_1, …, v_n.

On identifie de plus tous les termes convertibles par α-renommage :

$$\begin{array}{l} \lambda x\cdot u \sim \lambda y\cdot u[y/x] \\ \mathsf{box}\ u\ \mathsf{with}\ v_1,\ldots,v_n\ \mathsf{for}\ x_1,\ldots,x_n \\ \quad \sim \mathsf{box}\ u[y_1/x_1,\ldots,y_n/x_n] \\ \qquad \mathsf{with}\ v_1,\ldots,v_n\ \mathsf{for}\ y_1,\ldots,y_n \end{array}$$

et par la relation de conversion commutative :

$$\begin{array}{c} \mathsf{box}\ u\ \mathsf{with}\ v_1,\ldots,v_n\ \mathsf{for}\ x_1,\ldots,x_n \\ \sim \mathsf{box}\ u\ \mathsf{with}\ v_{\pi(1)},\ldots,v_{\pi(n)}\ \mathsf{for}\ x_{\pi(1)},\ldots,x_{\pi(n)} \end{array}$$

pour toute permutation π de $\{1,\ldots,n\}$.

On peut alors définir la construction d'évaluation $,u$ de la règle $(\Box L)$ comme étant unbox u, et la construction de citation $u`$ de la règle $(\Box R)$ comme étant :

$$\begin{array}{l} \mathsf{box}\ u[z_1/x_1,\ldots,z_n/x_n] \\ \quad \mathsf{with}\ x_1,\ldots,x_n \\ \quad \mathsf{for}\ z_1,\ldots,z_n \end{array}$$

$$(Ax) \quad \frac{}{\Gamma, x : \Phi \vdash x : \Phi}$$

$$(\Rightarrow I) \quad \frac{\Gamma, x : \Phi_1 \vdash u : \Phi_2}{\Gamma \vdash \lambda x \cdot u : \Phi_1 \Rightarrow \Phi_2} \qquad (\Rightarrow E) \quad \frac{\begin{array}{c}\Gamma \vdash u : \Phi_1 \Rightarrow \Phi_2 \\ \Gamma \vdash v : \Phi_1\end{array}}{\Gamma \vdash uv : \Phi_2}$$

$$(\Box\mathcal{I}) \quad \frac{\begin{array}{c}\Gamma \vdash v_1 : \Box\Phi_1 \quad \ldots \quad \Gamma \vdash v_n : \Box\Phi_n \\ x_1 : \Box\Phi_1, \ldots, x_n : \Box\Phi_n \vdash u : \Phi\end{array}}{\Gamma \vdash \mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_n : \Box\Phi} \qquad (\Box\mathcal{E}) \quad \frac{\Gamma \vdash u : \Box\Phi}{\Gamma \vdash \mathsf{unbox}\ u : \Phi}$$

Figure 5.1: Règles de typage du λ_{S4}-calcul

$$(Ax) \quad \frac{}{\Gamma, x : \Phi \vdash x : \Phi}$$

$$(\Rightarrow L) \quad \frac{\begin{array}{c}\Gamma, x : \Phi_2 \vdash u : \Phi_3 \\ \Gamma \vdash v : \Phi_1\end{array}}{\Gamma, y : \Phi_1 \Rightarrow \Phi_2 \vdash u[yv/x] : \Phi_3} \qquad (\Rightarrow R) \quad \frac{\Gamma, x : \Phi_1 \vdash u : \Phi_2}{\Gamma \vdash \lambda x \cdot u : \Phi_1 \Rightarrow \Phi_2}$$

$$(\Box L) \quad \frac{\Gamma, x : \Phi_1 \vdash u : \Phi_2}{\Gamma, y : \Box\Phi_1 \vdash u[{,}y/x] : \Phi_2} \qquad (\Box R) \quad \frac{\Gamma \vdash u : \Phi}{\Gamma, \Gamma' \vdash u\text{`} : \Box\Phi}$$

$$(\Gamma \text{ emboîté})$$

$$(Cut) \quad \frac{\Gamma \vdash u : \Phi_1 \quad \Gamma', x : \Phi_1 \vdash v : \Phi_2}{\Gamma, \Gamma' \vdash v[u/x] : \Phi_2}$$

Figure 5.2: Système **LS4**

où $x_1, \ldots, x_n$ sont les variables libres de u et $z_1, \ldots, z_n$ sont de nouvelles variables distinctes deux à deux et distinctes de $x_1, \ldots, x_n$ – construction qui n'est bien définie que modulo $\sim$ et α-conversion.

Les règles naturelles correspondant à l'élimination des coupures sont, outre la règle β :

$$(\lambda x \cdot u)v \to u[v/x]$$

représentant l'interaction principale entre $(\Rightarrow R)$ et $(\Rightarrow L)$, la règle (unbox) :

$$\begin{array}{l} \mathsf{unbox}\ (\mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_n) \\ \to u[v_1/x_1, \ldots, v_n/x_n] \end{array}$$

correspondant à l'interaction principale entre $(\Box R)$ et $(\Box L)$, et représentant la règle intuitive que l'évaluation d'un terme cité doit retourner le terme sous la citation; on a aussi quelques règles administratives – oubliées dans [BdP92] au même titre que la conversion commutative $\sim$ –, dont la règle (box) :

$$\begin{array}{l} \mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_n \\ \to \mathsf{box}\ u[(\mathsf{box}\ v'_i\ \mathsf{with}\ z_1, \ldots, z_m\ \mathsf{for}\ y_1, \ldots, y_m)/x_i] \\ \qquad \mathsf{with}\ v_1, \ldots, v_{i-1}, w_1, \ldots, w_m, v_{i+1}, \ldots, v_n \\ \qquad \mathsf{for}\ x_1, \ldots, x_{i-1}, z_1, \ldots, z_m, x_{i+1}, \ldots, x_n \end{array}$$

pourvu que v_i soit de la forme $\mathsf{box}\ v'_i\ \mathsf{with}\ w_1, \ldots, w_m\ \mathsf{for}\ y_1, \ldots, y_m$, et où $z_1, \ldots, z_m$ sont de nouvelles variables – ce qui correspond grosso modo à l'équation `(..., `u ...) → `(...u ...) –; ensuite la règle de *garbage collection* (gc) :

$$\begin{array}{l} \mathsf{box}\ u\ \mathsf{with}\ v_1, \ldots, v_n\ \mathsf{for}\ x_1, \ldots, x_n \\ \to \mathsf{box}\ u \\ \qquad \mathsf{with}\ v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n \\ \qquad \mathsf{for}\ x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n \end{array}$$

si x_i n'est pas libre dans u – dont le contenu informatique est clair –; et enfin la règle de *contraction* (ctract) :

$$\begin{array}{l} \mathsf{box}\ u \mathsf{with}\ v_1, \ldots, v_i, \ldots, v_j, \ldots, v_n \\ \qquad \mathsf{for}\ x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n \\ \to \mathsf{box}\ u[x_i/x_j] \\ \qquad \mathsf{with}\ v_1, \ldots, v_i, \ldots, v_{j-1}, v_{j+1}, \ldots, v_n \\ \qquad \mathsf{for}\ x_1, \ldots, x_i, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n \end{array}$$

si $i \neq j$ et $v_i = v_j$.

Le rapport [GL96b] consistait à définir ce calcul, qui est la version corrigée du calcul de [BdP92], où seules les règles (β) et (unbox) étaient mentionnées, et la conversion commutative $\sim$ était oubliée. Je montre que ce calcul est confluent (même dans le cas non typé), que sa version simplement typée a la propriété d'auto-réduction et termine, que l'inférence de type est faisable en temps polynomial et que tout terme typable a un typage principal, que les termes typés – ou plutôt les classes de termes typés modulo la plus petite congruence contenant $\sim$, et la conversion par (gc) et (ctract) – sont isomorphes aux preuves en S4, et enfin que les termes typés normaux sont des représentants de preuves sans coupures — autrement dit, la version logique de la propriété d'absence de blocage des programmes.

Le calcul est néanmoins relativement compliqué, et surtout il est aussi satisfaisant d'un point de vue logique qu'il est peu satisfaisant d'un point de vue informatique. En particulier, son caractère est très peu opérationnel, et la réalisation d'un interprète efficace — même d'un sous-ensemble non-trivial — laisse perplexe. En fait, si la notion de substitution nécessaire à la formalisation de la règle (β) est déjà problématique en λ-calcul, que dire alors des autres règles modales? Les règles (box) et (gc) demandent notamment une bonne dose d'astuce, et la règle (ctract) n'est même pas une règle linéaire à gauche, ni une règle locale (on ne peut l'appliquer qu'après avoir déterminé si deux termes v_i et v_j, aussi éloignés qu'ils soient à l'intérieur du with, sont égaux).

Le calcul dont je parle dans la section suivante, le $\lambda\mathsf{ev}Q$-calcul, a de moins bonnes propriétés logiques mais est bien mieux adapté à la réalisation d'interprètes. Si j'ai introduit le λ_{S4}-calcul, c'est parce qu'il offre un langage de termes de preuve qui s'est avéré pratique pour parler des relations entre $\lambda\mathsf{ev}Q$-termes typés et preuves en S4, sans avoir à écrire de vraies preuves de S4.

5.3 Le $\lambda\mathsf{ev}Q$-calcul

Pour dériver le $\lambda\mathsf{ev}Q$-calcul, je procède d'une façon entièrement différente, et j'essaie d'analyser la difficulté qu'il y a à interpréter la remontée des coupures dans une preuve en S4 au moyen d'un langage de termes.

Le problème principal est le terme $u^{\lq}$, c'est-à-dire l'interaction des coupures avec la règle $(\Box R)$ (cf. figure 5.2). Supposons que $u^{\lq}$ soit un terme quelconque, sans relation a priori avec u, et regardons ce que les divers mouvements nécessaires de remontée des coupures imposent sur les relations entre $u^{\lq}$ et u. Une analyse exhaustive des différents cas que j'éviterai de répéter (cf. [GL96b]) montre que, d'une part l'ensemble des variables libres de $u^{\lq}$ devrait être identique à celui de u, et d'autre part $,(u^{\lq})$ devrait se réduire en u. Par contre, on ne peut pas avoir $u^{\lq}[v/x] = (u[v/x])^{\lq}$ en général, mais un passage est autorisé entre les deux si v est lui-même de la forme $w^{\lq}$, ou plus généralement $w^{\lq}\sigma$, où σ est une substitution, auquel cas on veut que $u^{\lq}[w^{\lq}\sigma/x]$ se réduise en $(u[w/x])^{\lq}\sigma$.

La solution du λ_{S4}-calcul était de définir explicitement $u^{\lq}$ comme vérifiant ces propriétés. L'idée derrière le $\lambda\mathtt{ev}Q$-calcul [GL96c] est de définir un langage de plus bas niveau, et une fonction $u \mapsto u^{\lq}$ des termes vers les termes (qui ne sera pas définissable dans le langage typé, de même que $u \mapsto u^{\lq}$ n'est pas typable en λ_{S4} en général) qui ait les propriétés demandées. D'un point de vue informatique, c'est-à-dire Lispien ici, il s'agit de la traduction qu'un analyseur syntaxique Lisp effectue pour transformer les anti-citations, par exemple `` `(u ,v) ``, en expressions plus élémentaires équivalentes comme `(list 'u v)`. D'un point de vue logique, il s'agit d'effectuer une élimination non seulement de la règle de coupure (Cut) des preuves, mais aussi de la règle problématique $(\Box R)$, au profit de règles plus aisément manipulables (il s'agit donc de ce qui est parfois appelé une élimination des coupures *hétérodoxe*).

Je ne vais pas décrire en détail pourquoi et comment cette méthode fonctionne, mais je vais tenter d'en donner une idée intuitive.

Pour éliminer $(\Box R)$ d'une preuve, je regarde la dernière règle utilisée juste au-dessus d'une instance de $(\Box R)$. Plutôt que de montrer ce qui se passe dans le calcul des séquents, et qui est relativement peu lisible, je vais illustrer le processus en imaginant que cette dernière règle est $(\Rightarrow E)$ – qui est une règle de déduction naturelle, mais je sacrifie ici le caractère formel au caractère intuitif :

$$\dfrac{\dfrac{\begin{matrix}\vdots\\ \Box\Gamma \vdash u : \Phi_1 \Rightarrow \Phi_2\end{matrix} \quad \begin{matrix}\vdots\\ \Box\Gamma \vdash v : \Phi_1\end{matrix}}{\Box\Gamma \vdash uv : \Phi_2}\,(\Rightarrow E)}{\Box\Gamma, \Gamma' \vdash (uv)^{\lq} : \Box\Phi_2}\,(\Box\mathcal{I})$$

(remarquer que $(\Box R)$ et $(\Box\mathcal{I})$ sont deux noms différents pour une même règle, et je parle donc bien ici aussi de $(\Box R)$), alors en créant un nouvel opérateur $\star^1$ (application emboîtée) obéissant à la règle de type :

$$(\Box \Rightarrow E) \quad \dfrac{\Gamma \vdash u : \Box(\Phi_1 \Rightarrow \Phi_2) \quad \Gamma \vdash v : \Box\Phi_1}{\Gamma \vdash u \star^1 v : \Box\Phi_2}$$

(qui est une déduction valide en S4, en fait c'est le typage de l'opérateur $\star$ de la section 5.1), on peut faire remonter la règle $(\Box R)$ dans l'exemple pour obtenir :

$$\dfrac{\dfrac{\begin{matrix}\vdots\\ \Box\Gamma \vdash u : \Phi_1 \Rightarrow \Phi_2\end{matrix}}{\begin{matrix}\Box\Gamma, \Gamma' \vdash\\ u^{\lq} : \Box(\Phi_1 \Rightarrow \Phi_2)\end{matrix}}\,(\Box\mathcal{I}) \quad \dfrac{\begin{matrix}\vdots\\ \Box\Gamma \vdash v : \Phi_1\end{matrix}}{\begin{matrix}\Box\Gamma, \Gamma' \vdash\\ v^{\lq} : \Box\Phi_1\end{matrix}}\,(\Box\mathcal{I})}{\Box\Gamma, \Gamma' \vdash u^{\lq} \star^1 v^{\lq} : \Box\Phi_2}\,(\Box \Rightarrow E)$$

La règle qui correspond à $(\Box \Rightarrow E)$ dans le calcul des séquents est :

$$(\Box \Rightarrow L) \quad \dfrac{\Gamma, x : \Box\Phi_2 \vdash w : \Phi \qquad \Gamma \vdash v : \Box\Phi_1}{\Gamma, y : \Box(\Phi_1 \Rightarrow \Phi_2) \vdash w[y \star^1 v/x] : \Phi}$$

et l'avantage de cette règle par rapport à $(\Box R)$ est que celle-ci ne pose aucun problème vis-à-vis de la remontée des coupures – ce qui signifie en pratique que $(u \star^1 v)\sigma = u\sigma \star^1 v\sigma$ pour toute substitution σ.

On fait de même pour toutes les autres règles logiques, et le seul vrai problème vient de la règle $(\Rightarrow R)$, qui demande à interpréter l'action de $\lambda x\cdot$ sous les boîtes $\Box$. Une analyse du problème [GL96c] montre que la remontée de $(\Box R)$ par-dessus les autres règles logiques revient en fait à créer un langage de preuves *interne*, c'est-à-dire essentiellement un langage du premier ordre muni de règles de réduction capables de simuler fidèlement la β-réduction en λ-calcul. Les candidats idéaux pour ce faire sont les λ-calculs avec substitutions explicites [ACCL90]; l'opérateur $\star^1$ ci-dessus n'est rien d'autre que l'opérateur d'application du $\lambda\sigma$-calcul.

Les types sont enrichis, et $\Box\Phi$ est maintenant vu comme une abréviation de $\top \overset{\Box}{\Rightarrow} \Phi$, où $\overset{\Box}{\Rightarrow}$ est une « implication modale », et $\top$ est le type des piles vides. Si $u : \varsigma \overset{\Box}{\Rightarrow} \tau_1 \Rightarrow \tau_2$ et $v : \varsigma \overset{\Box}{\Rightarrow} \tau_1$, alors $u \star^1 v : \varsigma \overset{\Box}{\Rightarrow} \tau_2$, et l'opérateur d'abstraction λ^1 du $\lambda\sigma$-calcul obéit à la règle de typage typique d'une curryfication :

$$\frac{\Gamma \vdash u : \tau_1 \times \varsigma \overset{\Box}{\Rightarrow} \tau_2}{\Gamma \vdash \lambda^1 u : \varsigma \overset{\Box}{\Rightarrow} \tau_1 \Rightarrow \tau_2}\,(\Box \Rightarrow I)$$

(dans le calcul de séquents, cette règle se nomme $(\Box \Rightarrow R)$). Ceci résout le problème de la remontée de $(\Box R)$ par-dessus $(\Rightarrow R)$, qui produit un $(\Box R)$ au-dessus d'un $(\Box \Rightarrow R)$.

$(\Box R)$ doit aussi remonter au-dessus des règles que j'ai rajoutées, en particulier $(\Box \Rightarrow L)$ et $(\Box \Rightarrow R)$, et ceci fournit des règles $(\Box^2 \Rightarrow L)$ (typant un opérateur d'application $\star^2$ sous deux boîtes), $(\Box^2 \Rightarrow R)$ (typant un opérateur d'abstraction λ^2 sous deux boîtes), et de même sous trois, quatre, $\ldots$, n boîtes, pour tout $n \geq 1$.

J'ai appelé le calcul obtenu le $\lambda\mathtt{ev}Q$-calcul. On y retrouve en plus un opérateur binaire $\mathtt{ev}^\ell$ pour tout $\ell \geq 1$: $\mathtt{ev}^1 u()$ correspond à l'opérateur `eval` de Lisp, et $\mathtt{ev}^1 uv$ provoque l'évaluation de u dans la pile v (définir `eval` par une fonction d'évaluation binaire est une technique commune en Lisp), et en général $\mathtt{ev}^\ell uv$ effectue l'évaluation de u dans la pile v sous $\ell - 1$ boîtes. On y trouve aussi un opérateur Q^ℓ pour tout $\ell \geq 1$, qui est formellement un opérateur `kwote` sous $\ell - 1$ boîtes.

Le $\lambda\mathtt{ev}Q$-calcul est compliqué d'un point de vue logique : il est en effet défini par 63 schémas de règles – dont 3 règles, 30 schémas indexés par un entier $\ell \geq 1$, et 30 indexés par deux entiers ℓ et $\mathcal{L}$ tels que $1 \leq \ell < \mathcal{L}$ – se répartissant comme suit (cf. [GL96c] pour la liste) :

- Groupe (A) : (β)-réduction usuelle (mécanisme calculatoire au niveau 0, qui est un λ-calcul);
- groupe (B) : pour chaque $\ell \geq 1$, un $\lambda\sigma$-calcul travaillant au niveau ℓ (c'est-à-dire que les règles ne font intervenir que des opérateurs λ^ℓ, $\star^\ell$, etc. avec le même ℓ, par exemple la règle (β^ℓ) : $(\lambda^\ell u) \star^\ell v \to u \circ^\ell_T (v \bullet^\ell id^\ell)$);
- groupe (C) : pour chaque niveau $\ell \geq 1$, une description du comportement de la fonction `eval` à ce niveau, par exemple $(\mathtt{ev}\star^\ell)$: $\mathtt{ev}^\ell(u \star^\ell v)w \to (\mathtt{ev}^\ell uw) \star^{\ell-1} (\mathtt{ev}^\ell vw)$ (où $\star^0$ dénote l'application invisible du λ-calcul habituel), ou par exemple $(\mathtt{ev}Q^\ell)$: $\mathtt{ev}^\ell(Q^\ell u)v \to u$, qui représente l'essentiel de la règle ,'$u = u$);
- groupe (D) : propagation des substitutions créées au niveau $\ell \geq 1$ à travers les termes de niveaux $\mathcal{L} > \ell$;
- groupe (E) : évaluation au niveau $\ell \geq 1$ (par $\mathtt{ev}^\ell$) de termes de niveaux $\mathcal{L} > \ell$;
- groupe (F) : citation au niveau $\ell \geq 1$ (par Q^ℓ) de termes de niveaux $\mathcal{L} > \ell$.

Cependant, la taille du système de réécriture n'est pas effrayante d'un point de vue informatique, si l'on considère que les interprètes de bytecode courants, et les microprocesseurs, possèdent souvent bien davantage d'instructions élémentaires. La comparaison est fondée, car un système de réécriture comme $\lambda\mathtt{ev}Q$ peut être vu comme une description abstraite d'une machine, à peu de choses près [ACCL90].

Un point remarquable est que l'on retrouve un certain nombre de constructions Lisp classiques, comme la fonction `eval` à deux arguments, mais surtout la hiérarchisation de l'univers de termes en niveaux entiers, qui correspond exactement à la tour réflexive de Lisp, qui est ici décrite sans avoir à faire appel à la notion de méta-continuation [WF86].

Le reste du rapport [GL96c] consiste essentiellement à montrer que la notion de réduction de $\lambda\mathtt{ev}Q$ simule effectivement celle de λ_{S4}. J'y introduis aussi une extension modalement extensionnelle, c'est-à-dire avec une règle de style η mais portant sur les connecteurs modaux. Ceci fournit les calculs λ_{S4H}, qui a la règle supplémentaire :

$$\begin{array}{ll}(\eta\mathsf{box}) & \mathsf{box}\ (\mathsf{unbox}\ x_i) \\ & \quad \mathsf{with}\ v_1, \ldots, v_n \\ & \quad \mathsf{for}\ x_1, \ldots, x_n \\ & \to v_i\end{array}$$

correspondant à l'égalité Lisp intuitive ' $, u = u$, et $\lambda\mathsf{ev}Q_H$, dont la règle supplémentaire la plus importante est :

$$(\eta\mathsf{ev}^\ell) \quad \mathsf{ev}^{\ell+1}(Q^\ell u)v \to u \circ^\ell v$$

pour tout $\ell \geq 1$.

Les rapports suivants [GL96d, GL97b] établissent les propriétés logiques du $\lambda\mathsf{ev}Q$-calcul. Comme ce calcul contient des calculs de substitutions explicites riches, ce calcul ne termine pas même dans le cas typé, le contre-exemple de Paul-Andé MELLIÈS s'appliquant [Mel94, Mel95]. Je montre en [GL96d] que le $\lambda\mathsf{ev}Q$-calcul typé est confluent et est une extension conservative – pour l'égalité induite par la réduction – du λ_{S4}-calcul typé, c'est-à-dire de la théorie de l'égalité entre preuves de S4 induite par la remontée des coupures. Le $\lambda\mathsf{ev}Q$-calcul est donc un langage adéquat lui aussi pour décrire des preuves de S4, mais l'extension étant stricte, il décrit en fait une classe d'objets plus généraux. Les preuves sont longues et relativement difficiles, et occupent la majorité des 53 pages du rapport. La difficulté principale est en fait de montrer qu'un certain sous-système de $\lambda\mathsf{ev}Q$ nommé Σ, et qui contient toutes les règles sauf (β) et les (β^ℓ), $\ell \geq 1$, termine – Σ joue ici le rôle du sous-système σ du $\lambda\sigma$-calcul. Il s'avère que Σ ne termine en fait pas dans le cas non typé (contrairement à σ), mais termine dans le cas typé.

Le rapport [GL97b] montre que le calcul modalement extensionnel $\lambda\mathsf{ev}Q_H$ est lui aussi confluent dans le cas typé, alors qu'il ne l'est pas dans le cas non typé (le cas de $\lambda\mathsf{ev}Q$ non typé reste, lui, ouvert). Il s'ensuit que le $\lambda\mathsf{ev}Q_H$-calcul typé est une extension conservative du λ_{S4H}-calcul typé. Ceci, de plus, dépend du typage, mais pas tant de propriétés de terminaison par exemple, que de conditions de cohérence des piles assurées par le système de types : des extensions de ces calculs avec des opérateurs de point fixe ont en effet les mêmes propriétés de confluence et de conservativité de l'extension définie par le $\lambda\mathsf{ev}Q$-calcul enrichi par rapport au λ_{S4}-calcul enrichi.

Une autre démonstration relativement longue établit que le $\lambda\mathsf{ev}Q_H$-calcul simplement typé termine faiblement, mais échoue à montrer que le $\lambda\mathsf{ev}Q$-calcul typé termine faiblement (un autre problème ouvert). La démonstration procède par une traduction vers le λ-calcul simplement typé. Cette même traduction se simplifie un tant soit peu dans le cas des $\lambda\sigma$-calculs et fournit une preuve de terminaison faible de plusieurs calculs de substitutions explicites, preuve que j'ai présentée dans [GL97c, GL97d].

J'ai, et ce depuis presque un an, un rapport en préparation sur l'extension à la version classique de S4 des calculs ci-dessus, mais que je ne voulais pas terminer avant d'avoir terminé la partie IIIb. Une des motivations que j'avais à étudier S4 classique était l'étude des diverses traductions connues de la logique intuitionniste vers S4 classique, mais l'on obtient déjà des résultats intéressants par la traduction de KRIPKE (traduisant les formules atomiques A en $f(A) = \Box A$ et posant $f(\Phi_1 \Rightarrow \Phi_2) = \Box(f(\Phi_1) \Rightarrow f(\Phi_2))$, ou mieux : $f(\Phi_1 \Rightarrow \Phi_2) = f(\Phi_1) \overset{\Box}{\Rightarrow} f(\Phi_2))$ qui fonctionne même avec S4 intuitionniste comme cible. La façon a mon sens la plus naturelle de procéder produit alors une version en appel par valeur des combinateurs séquentiels de Healfdene GOGUEN [Gog97], et qui est un sous-système de Σ (ce qui explique pourquoi Σ ne termine pas dans le cas non typé, puisque les combinateurs séquentiels, resp. par valeur, interprètent le λ-calcul, resp. le λ-calcul par valeur). Un point intéressant est que ma preuve de terminaison forte de Σ typé [GL96d, GL97b] montre que la restriction par valeur des combinateurs séquentiels termine fortement; je cherche en ce moment à voir si ceci implique que les combinateurs séquentiels typés sans la restriction par valeur terminent – une conjecture de [Gog97] –, mais ceci est loin d'être évident.

Un autre point motivant l'étude de S4 classique est qu'il semble que ce soit la bonne façon de comprendre comment les mécanismes d'évaluation et de citation, et en général de retardement d'exécution, se mélangent avec ceux de gestion d'exceptions – une question non triviale. Techniquement, un point intéressant est que λ_{S4} semble relativement inadapté comme candidat à l'extension classique : la confluence est immédiatement perdue, et ceci est dû au fait qu'en un certain sens, les mécanismes de gestion des exceptions doivent définir, dans un terme où l'on peut capturer une continuation à deux endroits différents, quel est celui qui doit avoir priorité. La solution du $\lambda\mu$-calcul de PARIGOT, par exemple (dans le cadre de la logique classique non modale), est de donner la priorité à u dans le cas de l'application uv. Dans le cas de λ_{S4}, il n'y a aucun moyen simple d'établir ce genre de priorité à partir d'un ordre textuel (ici, u avant v), à cause de la règle de conversion commutative $\sim$ – on peut probablement établir un tel ordre par des conditions plus complexes, cependant. Le $\lambda\mathtt{ev}Q$-calcul, au contraire, s'étend sans problème à un calcul contenant des mécanismes d'exception à chaque niveau de la tour réflexive, c'est-à-dire des moyens de décrire les preuves en S4 classique. Je décrirai dans un rapport futur cette extension, qui est fondée sur une variante du système de λ-calcul avec substitutions explicites de Philippe AUDEBAUD [Aud94].

Mais l'un des points les plus étranges et les plus fascinants concernant le $\lambda\mathtt{ev}Q$-calcul, ou plus exactement le $\lambda\mathtt{ev}Q_H$-calcul simplement typé, est ma découverte fin décembre 1995 que l'espace des $\lambda\mathtt{ev}Q_H$-termes typés de types de la forme $\top \overset{\Box}{\Rightarrow} \ldots \overset{\Box}{\Rightarrow} \top\tau$, où τ est un type ne commençant pas par $\overset{\Box}{\Rightarrow}$, quotienté par la congruence engendrée par la notion de réduction du calcul, est un *complexe simplicial* [May67] (de même dans le cas classique). Ceci implique que, tout du long, j'ai essentiellement étudié un objet de topologie algébrique sans le savoir. Ceci implique aussi qu'il existe une notion naturelle d'homotopie entre preuves de S4, c'est-à-dire d'équivalence à déformation près. De plus, cette notion n'est pas triviale – il existe des preuves homotopes non égales – et n'est pas naturellement définissable en λ_{S4H} (mais presque). L'étude que j'ai commencé à mener avec mon frère, Éric GOUBAULT, sur la question, avance lentement. La première question à laquelle nous nous sommes intéressés était d'établir si le complexe simplicial vérifiait la condition d'extension de KAN, à savoir qu'il existe « suffisamment de simplexes ». Il semble désormais que ceci soit le cas, mais encore une fois la preuve n'est pas immédiate. Savoir que le complexe simplicial est un complexe de KAN permet d'avoir une notion simple d'homotopie. Les étapes suivantes consisteront à tenter de comprendre non seulement la signification informatique de la notion de déformation de preuves (de programmes), mais aussi d'étudier la structure mathématique des groupes d'homotopie et d'homologie. Finalement, la proximité de S4 et de la logique linéaire suggère que les réseaux de preuve de la logique linéaire devraient être eux aussi équipés d'une structure similaire; le candidat naturel est la structure de *complexe cubique*, dont mon frère a fait son instrument de travail dans l'étude de modèles du parallélisme [Gou95a] – un domaine où l'on sait déjà l'importance de la logique linéaire.

Chapitre 6

Enseignement

Pour le moment, mon activité d'enseignement a consisté en un cours à l'École Nationale Supérieure des Techniques Avancées (ENSTA) aux élèves de troisième année en 1995, 1996 et 1997, un cours au DEA Informatique, Mathématiques et Applications (IMA) à l'École Polytechnique de novembre 1994 à février 1995 et un cours au DEA Sémantique, Preuves et Programmation à Paris VII, filière preuves, en 1996 et 1997.

Le cours de troisième année à l'ENSTA est une introduction aux langages et méthodes de spécification formelle, portant sur un sous-ensemble simple de VDM en 1995, puis sur le langage Z en 1996 et 1997, ce dernier présentant l'avantage de n'être essentiellement qu'une notation pratique au-dessus d'une théorie des ensembles typée en logique classique du premier ordre. Ce cours présente les notions fondamentales de cycle de vie, de spécification formelle, de raffinement, et de preuve formelle. La partie preuve a notamment été renforcée en 1997, ainsi que l'étude d'exemples réels – cet année, la spécification et l'examen critique d'une version simplifiée du protocole réseau ATMR (Asynchronous Transfer Mode Ring), qui est particulièrement bien adaptée aux exigences de réalisme pour une application des méthodes formelles, et en même temps à la durée limitée du cours.

Plus lié aux activités de démonstration automatique était le cours du DEA IMA [GM95], intitulé “Théorie de la preuve et démonstration automatique”. J'ai conçu et enseigné ce cours avec Ian MACKIE, de l'Imperial College à Londres et actuellement au LIX, à l'École Polytechnique. Ce cours avait pour but de présenter un éventail large d'outils mathématiques de preuve et informatiques de recherches de preuves, ou d'interprétations de preuves par des langages de programmations (Curry-Howard, Prolog). Le programme était le suivant :

1. Introduction : notion de vérité, de preuve (Ian MACKIE);
2. Logique propositionnelle classique : syntaxe, sémantique, systèmes de preuve (de HILBERT, déduction naturelle, séquents de GENTZEN), méthodes de preuve (résolution, tableaux, DAVIS-PUTNAM, BDD) (Jean GOUBAULT-LARRECQ);
3. Autres logiques propositionnelles : logiques propositionnelles intuitionniste, linéaire, affine (*relevant logic*), systèmes de preuve, sémantique de BROUWER-HEYTING-KOLMOGOROV de la logique intuitionniste (Ian MACKIE);
4. La correspondance de CURRY-HOWARD : logique intuitionniste et λ-calcul, interprétations calculatoires de la logique linéaire, décoration de preuves intuitionnistes ou classiques sous forme de preuves linéaires (Ian MACKIE);
5. Logiques modales : utilisations informatiques, syntaxe, sémantique de KRIPKE, systèmes de preuve (étude de cas sur S4); autres logiques modales (logique de

HENNESSY-MILNER, logique propositionnelle dynamique, CTL, CTL*, μ-calcul modal) et vérification de modèles (model-checking) (Jean GOUBAULT-LARRECQ);

6. Logique classique du premier ordre : syntaxe, sémantique, calcul des séquents, élimination des coupures, cohérence, complétude, compacité, skolémisation et théorèmes de HERBRAND (Jean GOUBAULT-LARRECQ);

7. Résolution en logique classique du premier ordre : correction, complétude, stratégies et optimisations (subsomption, tautologies, clauses pures, résolution à littéraux ordonnés, set-of-support, hyperrésolution, résolution linéaire) (Jean GOUBAULT-LARRECQ);

8. Méthodes des tableaux au premier ordre : tableaux, connexions et matings, tableaux à variables libres, élimination de modèles et relation avec la résolution linéaire (Jean GOUBAULT-LARRECQ);

9. Intégration de théories : théorie de l'égalité et réécriture, complétion de KNUTH-BENDIX, complétion sans échec, paramodulation orientée, méthode des matings équationnels; théories équationnelles, théories générales, utilisation de sortes (Jean GOUBAULT-LARRECQ);

10. Programmation logique : Prolog, Prolog parallèles et logique linéaire, langages à contraintes (Ian MACKIE).

Ce programme fournit donc un éventail large d'outils mathématiques et informatiques de base. Certains domaines ont été écartés du cours oral, qui était déjà assez dense. Un des exemples les plus flagrants est constitué par les logiques d'ordre supérieur, qui n'étaient mentionnées que dans le support de cours écrit. Le domaine est vaste, et nous avons préféré concevoir ce cours comme une base large et si possible relativement peu profonde, mais avec des pointeurs vers d'autres livres ou articles permettant aux étudiants désireux d'approfondir un sujet de pouvoir trouver les références essentielles.

Ce cours a, semble-t-il, remporté un certain succès. Les cours du DEA SPP qui en ont pris la suite comportaient essentiellement le même programme, à ceci près que les points marqués du nom de Ian MACKIE n'étaient plus effectués – Ian enseignant ces thèmes avec Vincent DANOS dans une autre filière du DEA –, que je n'enseigne plus le point « intégration de théories », qui ferait essentiellement double emploi avec les cours du tronc commun, que j'ai ajouté, à la demande de Gérard HUET, une introduction à l'algorithme d'unification d'ordre supérieur, et donc aussi aux logiques d'ordre supérieur, et que finalement Michaël RUSINOWITCH enseigne pendant une séance du cours les techniques de résolution et paramodulation ordonnée, dont il avait développé dans sa thèse les méthodes de preuve de complétude.

Malgré le succès répété de ce cours, il ne m'a malheureusement pas été possible d'encadrer par la suite des étudiants lors de leurs stages de DEA, du moins jusqu'à il y a peu. J'avais pu encadrer un étudiant chinois du DEA IMA en 1991, mais à partir de 1992, l'emploi de stagiaires chez Bull a commencé à être découragé, puis interdit à partir de 1993, pour des raisons budgétaires. Il ne m'a donc pas été possible dans ces conditions de proposer seulement des sujets de stage en 1994/1995, ce que je ne peux que regretter. Ma nouvelle situation au sein du G.I.E. Dyade a débloqué la situation en 1997, et un étudiant, Karim GAM, a commencé à travailler sur un codage par réflexion en Coq d'une tactique de vérification/réfutation de la connaissance de messages par un intrus dans le cadre de la vérification de protocoles cryptographiques. Il s'agit essentiellement de prouver correcte une méthode de tableaux pour un système logique que j'ai appelé – un peu pompeusement, certes – la logique de la connaissance cryptographique dans [GL97a], et d'en extraire automatiquement une tactique Coq correspondante.

Bibliographie

[ACCL90] Martín ABADI, Luca CARDELLI, Pierre-Louis CURIEN, and Jean-Jacques LÉVY. « Explicit Substitutions ». In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 31–46, San Francisco, California 1990. January.

[And86] Peter B. ANDREWS. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Computer Science and Applied Mathematics. Academic Press, 1986.

[Aud94] Philippe AUDEBAUD. « Explicit Substitutions for the Lambda-Mu-Calculus. ». Research Report RR94-26, LIP, ENS Lyon, 1994.

[BCMM+92] J. R. BURCH, Edmund M. CLARKE, Kenneth L. MC MILLAN, David L. DILL, and L. J. HWANG. « Symbolic Model Checking: 10^{20} States and Beyond ». *Information and Computation*, 98(2):142–170, juin 1992.

[BdP92] Gavin BIERMAN and Valeria de PAIVA. « Intuitionistic Necessity Revisited ». In *Logic at Work*, Amsterdam, the Netherlands, 1992.

[BHlS93] Bernhard BECKERT, Reiner HÄHNLE, and Peter H. SCHMITT. « The Even More Liberalized δ-Rule in Free Variable Semantic Tableaux ». In Gottlob et al. [GLM93].

[Bil87] Jean-Paul BILLON. « Perfect Normal Forms for Discrete Functions ». Technical Report 87019, Bull, 1987.

[BL94] Matthias BAAZ and Alexander LEITSCH. « On Skolemization and Proof Complexity ». *Fundamenta Informaticae*, 20(4), 1994.

[Bla96] Bruno BLANCHET. « Inférence de types avec dimensions sous Caml-Light ». Projet d'informatique de maîtrise, disponible en `http://www.eleves.ens.fr:8080/home/blanchet/cldim-eng.html`, 1996.

[Bou90] Nicolas BOURBAKI. *Théorie des ensembles.* Éléments de mathématiques. Masson, 1990. chapitres 1 à 4.

[BR88] Alan BAWDEN and Jonathan REES. « Syntactic Closures ». In *1988 ACM Conference on Lisp and Functional Programming*, pages 86–95, 1988.

[Bry86] Randall E. BRYANT. « Graph-Based Algorithms for Boolean Function Manipulation ». *IEEE Transaction on Computers*, C35(8):677–692, 1986.

[CL73] Chin-Lian CHANG and Richard Char-Tung LEE. *Symbolic Logic and Mechanical Theorem Proving.* Computer Science Classics. Academic Press, 1973.

[CM89] K. M. CHANDY and Jayadev MISRA. *Parallel Program De-*

sign. Addison-Wesley, Austin, Texas, mai 1989.

[CR91] William CLINGER and Jonathan REES. « The Revised[4] Report on the Algorithmic Language Scheme ». *LISP Pointers*, 4(3):1–55, 1991.

[Dav96] Rowan DAVIES. « A Temporal Logic Approach to Binding-Time Analysis ». In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS'96)*, New Brunswick, New Jersey, juillet 1996.

[DBDGK94] Koen DE BOSSCHERE, Saumya DEBRAY, David GUDEMAN, and Sampath KANNAN. « Call Forwarding: A Simple Interprocedural Optimization Technique for Dynamically Typed Languages ». In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages (POPL'94)*, pages 409–420, 1994.

[DG79] Burton DREBEN and Warren D. GOLDFARB. *The Decision Problem — Solvable Classes of Quantificational Formulas*. Addison–Wesley, Reading, Massachussetts, 1979.

[DHK95] Gilles DOWEK, Thérèse HARDIN, and Claude KIRCHNER. « Higher-order unification via explicit substitutions ». In *Proceedings of the 10th Annual IEEE Symposium on Logics in Computer Science (LICS'95)*, San Diego, CA, USA, juillet 1995.

[Dil88] Antoni DILLER. *Compiling Functional Languages*. John Wiley and Sons, 1988.

[DJ92] Daniel J. DOUGHERTY and Patricia JOHANN. « A Combinatory Logic Approach to Higher-Order E-Unification ». In D. KAPUR, editor, *11th International Conference on Automated Deduction (CADE-11)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 79–93, Saratoga Springs, New York, USA, juin 1992. Springer Verlag.

[dK95] Eric de KOGEL. « Rigid E-Unification Revisited ». In *Proceedings of the Fourth Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, St. Goar am Rhein, Allemagne, mai 1995.

[DMV96] Anatoli DEGTYAREV, Yuri MATIYASEVICH, and Andrei VORONKOV. « Simultaneous Rigid E-Unification and Related Algorithmic Problems ». In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS'96)*, pages 494–502, New Brunswick, NJ, 1996.

[DP96] Rowan DAVIES and Frank PFENNING. « A Modal Analysis of Staged Computation ». In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 258–270, St. Petersburg Beach, Florida, 21–24 janvier 1996.

[DST80] Peter K. DOWNEY, Ravi SETHI, and Robert Endre TARJAN. « Variations on the Common Subexpression Problem ». *Journal of the ACM*, 27(4):758–771, 1980.

[DV96a] Anatoli DEGTYAREV and Andrei VORONKOV. « Equality elimination for the tableau method ». In J. CALMET and C. LIMONGELLI, editors, *Design and Implementation of Symbolic Computation Systems (DISCO'96)*, pages

46–60, Karlsruhe, Allemagne, 1996. Springer Verlag Lecture Notes in Computer Science 1128.

[DV96b] Anatoli DEGTYAREV and Andrei VORONKOV. « The Undecidability of Simultaneous Rigid E-Unification ». *Theoretical Computer Science*, 166(1–2):291–300, 1996.

[Dyc92] Roy DYCKHOFF. « Contraction-Free Sequent Calculi for Intuitionistic Logic ». *Journal of Symbolic Logic*, 57(3):795–807, 1992.

[Ede92] Elmar EDER. *Relative Complexities of First-Order Calculi*. Artificial Intelligence. Vieweg Verlag, Wiesbaden, Allemagne, 1992.

[Fef91] Solomon FEFERMAN. « Reflecting on Incompleteness ». *Journal of Symbolic Logic*, 56(1):1–49, mars 1991.

[Gal87] Jean H. GALLIER. *Logic for Computer Science — Foundations of Automatic Theorem Proving*. John Wiley and Sons, 1987.

[Gir87a] Jean-Yves GIRARD. « Linear Logic ». *Theoretical Computer Science*, 50:1–102, 1987.

[Gir87b] Jean-Yves GIRARD. *Proof Theory and Logical Complexity*, volume 1 of . Bibliopolis, 1987.

[GJ79] Michael R. GAREY and David S. JOHNSON. *Computers and Intractability — A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., San Francisco, 1979.

[GL96a] Jean GOUBAULT-LARRECQ. « Implementing Tableaux by Decision Diagrams ». Prépublication (Interner Bericht) 1996-32, Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, Allemagne, 1996.

[GL96b] Jean GOUBAULT-LARRECQ. « On Computational Interpretations of the Modal Logic S4 I. Cut Elimination ». Prépublication (Interner Bericht) 1996-35, Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, Allemagne, 1996. Disponible sur `ftp://theory.doc.ic.ac.uk/theory/guests/GoubaultJ/S4/`.

[GL96c] Jean GOUBAULT-LARRECQ. « On Computational Interpretations of the Modal Logic S4 II. The $\lambda \mathtt{ev} Q$-Calculus ». Prépublication (Interner Bericht) 1996-34, Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, Allemagne, 1996. Disponible sur `ftp://theory.doc.ic.ac.uk/theory/guests/GoubaultJ/S4/`.

[GL96d] Jean GOUBAULT-LARRECQ. « On Computational Interpretations of the Modal Logic S4 IIIa. Termination, Confluence, Conservativity of $\lambda \mathtt{ev} Q$ ». Prépublication (Interner Bericht) 1996-33, Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, Allemagne, 1996. Disponible sur `ftp://theory.doc.ic.ac.uk/theory/guests/GoubaultJ/S4/`.

[GL96e] Jean GOUBAULT-LARRECQ. « Ramified Higher-Order Unification ». Prépublication (Interner Bericht) 1996-31, Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, Allemagne, 1996. Version revue et corrigée en `ftp://theory.doc.ic.ac.uk/theory/guests/GoubaultJ/`.

[GL97a] Jean GOUBAULT-LARRECQ. « A Formalization of Cryptographic Knowledge ». En attente de soumission, 31 pages, avril 1997.

[GL97b] Jean GOUBAULT-LARRECQ. « On Computational Interpretations of the Modal Logic S4 IIIb. Confluence, Termination of $\lambda \mathtt{ev} Q_H$ ». Prépublication (rapport de recherche) (en cours), INRIA, Projet Coq, Rocquencourt, France, mai 1997.

[GL97c] Jean GOUBAULT-LARRECQ. « A Proof of Weak Termination of the Simply-Typed $\lambda\sigma$-Calculus ». Prépublication (rapport de recherche) 3090, INRIA, Projet Coq, Rocquencourt, France, janvier 1997.

[GL97d] Jean GOUBAULT-LARRECQ. « A Proof of Weak Termination of Typed $\lambda\sigma$-Calculi ». In *Proceedings of the TYPES'96 Workshop*, Aussois, France, 1997. Soumis. Disponible sur `http://www.dyade.fr/fr/actions/vip/jgl/types96.ps.gz`.

[GL97e] Jean GOUBAULT-LARRECQ. « Ramified Higher-Order Unification ». In *12th Annual IEEE Symposium on Logics in Computer Science (LICS'97)*, Varsovie, Pologne, juillet 1997.

[GLM93] G. GOTTLOB, A. LEITSCH, and D. MUNDICI, editors. *3rd Kurt Gödel Colloquium*, volume 713 of *Lecture Notes in Computer Science*, Brno, république tchèque, août 1993. Springer Verlag.

[GM95] Jean GOUBAULT and Ian MACKIE. « Proof Theory and Automated Deduction ». Rapport interne, École Polytechnique, mars 1995. Cours du DEA IMA, filière sémantique.

[GNRS92] Jean GALLIER, Paliath NARENDRAN, Stan RAATZ, and Wayne SNYDER. « Theorem proving using equational matings and rigid E-unification ». *Journal of the ACM*, 39(2):377–429, avril 1992.

[Gog97] Healfdene GOGUEN. « Sequent Combinators: A Hilbert System for the Lambda Calculus ». In *Proceedings of the TYPES'96 Workshop*, Aussois, France, 1997. Soumis. Disponible sur `http://www.dcs.ed.ac.uk/home/hhg/seq.dvi.gz`.

[Gol81] Warren D. GOLDFARB. « The undecidability of the second-order unification problem ». *Theoretical Computer Science*, 13:225–230, 1981.

[Gou91] Jean GOUBAULT. « A Natural Abstract Syntax and Semantics for the Palladio/VDM Specification Language ». Technical Report RT/91016, Bull S.A., juin 1991.

[Gou93a] Jean GOUBAULT. « Adding sets to ML: design, implementation and experiments ». Technical Report 93039, Bull, 1993.

[Gou93b] Jean GOUBAULT. « *Démonstration automatique en logique classique : complexité et méthodes* ». PhD thesis, École Polytechnique, Palaiseau, France, septembre 1993.

[Gou93c] Jean GOUBAULT. « A Rule-Based Algorithm for Rigid E-Unification ». In Gottlob et al. [GLM93], pages 202–210.

[Gou93d] Jean GOUBAULT. « Syntax Independent Connections ». In D. Basin *et al.*, editor, *Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, number MPI-I-93-213, Marseille, France, mai 1993. Max Planck Institut für Informatik, Saarbrücken, Allemagne.

[Gou93e] Jean GOUBAULT. « Une implémentation efficace de structures de données ensemblistes, fondée sur le hash-consing ». In *Journées francophones des langages applicatifs (JFLA'93)*, pages 222–238, Annecy, février 1993. INRIA.

[Gou94a] Jean GOUBAULT. « A BDD-Based Skolemization Procedure ». In Marcello D'AGOSTINO, editor, *Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, Abingdon, UK, mai 1994.

[Gou94b] Jean GOUBAULT. « The complexity of resource-bounded first-order classical logic ». In *Proceedings of the 11th Symposium on Theoretical Aspects of Computer Science (STACS'94)*, Caen, France, février 1994.

[Gou94c] Jean GOUBAULT. « Generalized Boxings, Congruences and Partial Inlining ». In *Proceedings of the 1st Static Analysis Symposium (SAS'94)*, Namur, Belgique, octobre 1994. Springer Verlag.

[Gou94d] Jean GOUBAULT. « Higher-Order Rigid E-Unification ». In *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning (LPAR'94)*, Kiev, Ukraine, juillet 1994.

[Gou94e] Jean GOUBAULT. « HimML: Standard ML with Fast Sets and Maps ». In *5th ACM SIGPLAN Workshop on ML and its Applications (ML'94)*, Orlando, FL, USA, juin 1994.

[Gou94f] Jean GOUBAULT. « Inférence d'unités physiques en ML ». In Pierre COINTE, Christian QUEINNEC, and Bernard SERPETTE, editors, *Journées francophones des langages applicatifs (JFLA'94)*, pages 3–20, L'Épine, Noirmoutier, France, janvier–février 1994.

[Gou94g] Jean GOUBAULT. « Proving with BDDs and Control of Information ». In A. BUNDY, editor, *12th International Conference on Automated Deduction (CADE-12)*, volume 814 of *Lecture Notes in Artificial Intelligence*, Nancy, France, juin–juillet 1994. Springer Verlag.

[Gou94h] Jean GOUBAULT. « Rigid $\vec{E}$-Unifiability is DEXPTIME-Complete ». In *Proceedings of the 9th Annual IEEE Symposium on Logics in Computer Science*, pages 498–506, Paris, France, juillet 1994.

[Gou95a] Éric GOUBAULT. « *The Geometry of Concurrency* ». PhD thesis, École Normale Supérieure, Paris, France, 1995.

[Gou95b] Jean GOUBAULT. « A BDD-Based Simplification and Skolemization Procedure ». *Journal of the IGPL (Special Interest Group in Pure and Applied Logics)*, 3(6), 1995. Disponible en `http://www.mpi-sb.mpg.de/igpl/Journal/V3-6/`.

[GP94] Jean GOUBAULT and Joachim POSEGGA. « BDDs and Automated Deduction ». In *Proceedings of the 8th International Symposium on Methodologies for Intelligent Systems (ISMIS'94)*, Charlotte, NC, USA, octobre 1994. Springer Verlag, Lecture Notes in Artificial Intelligence.

[Gro95] Jan Friso GROOTE. « Binary Decision Diagrams for First Order

Predicate Logic ≫. Non publié, envoyé directement par l'auteur pour avis (`JanFriso.Groote@phil.ruu.nl`), janvier 1995.

[GRS87] Jean GALLIER, Stan RAATZ, and Wayne SNYDER. ≪ Theorem proving using rigid E-unification equational matings ≫. In *Proceedings of the 2nd IEEE Conference on Logics in Computer Science (LICS'87)*, pages 338–346, , 1987.

[GSNP88] Jean GALLIER, Wayne SNYDER, Paliath NARENDRAN, and David PLAISTED. ≪ Rigid E-Unification is NP-Complete ≫. In *Proceedings of the 3rd IEEE Conference on Logics in Computer Science (LICS'88)*, pages 218–227, , 1988.

[Har96] John HARRISON. ≪ Stålmarck's Algorithm as a HOL Derived Rule ≫. In *9th international Conference on Theorem Proving in Higher Order Logics (TPHOL'96)*, pages 221–234, Karlsruhe, Allemagne, août 1996. Springer Verlag Lecture Notes in Computer Science 1125.

[Haz83] Allen HAZEN. Predicative Logics. In D. GABBAY and F. GUENTHER, editors, *Handbook of Philosophical Logic I: Elements of Classical Logic*, Chapitre I.5, pages 331–407. D. Reidel Publishing Company, Dordrecht, The Netherlands, 1983. (Synthese library Volume 164).

[HB39] David HILBERT and Paul BERNAYS. *Grundlagen der Mathematik II*, volume 50 of *Die Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen mit besonderer Berücksichtigung der Anwendungsgebiete*. Springer Verlag, 1939.

[Hue75] Gérard P. HUET. ≪ A Unification Algorithm for Typed λ-Calculus ≫. *Theoretical Computer Science*, 1:27–57, 1975.

[Jay94] Edwyn T. JAYNES. ≪ Probability Theory: The Logic of Science ≫. Livre en préparation; premiers chapitres disponibles en `http://omega.albany.edu:8008/JaynesBook.html`, juin 1994.

[Jon90] Cliff B. JONES. *Systematic Software Development Using VDM*. Prentice-Hall International Series in Computer Science, 2nd edition, 1990.

[Ken94] Andrew KENNEDY. ≪ Dimension types ≫. In D. SANELLA, editor, *Proceedings of the 5th European Symposium on Programming (ESOP'94)*, pages 348–362, Edinburgh, UK, 1994. Springer Verlag Lecture Notes in Computer Science 788.

[Ken97] Andrew KENNEDY. ≪ Relational Parametricity and Units of Measure ≫. In *Principles of Programming Languages*, Paris, France, janvier 1997. ACM.

[KN92] Deepak KAPUR and Paliath NARENDRAN. ≪ Double-Exponential Complexity of Computing a Complete Set of AC-Unifiers ≫. In *Proceedings of the 7th IEEE Conference on Logics in Computer Science (LICS'92)*, pages 11–21, 1992.

[KTU93] Andrzej J. KFOURY, Jerzy TIURYN, and Paweł URZYCZYN. ≪ The undecidability of the semi-unification problem ≫. *Information and Computation*, 102(1):83–101, 1993.

[Laa95] Twan LAAN. ≪ A Formalization of the Ramified Type Theory ≫. Technical Report 94/33, Eindhoven

Univerity of Technology, 1995. 40 pages. Available at `http://www.win.tue.nl/pub/techreports/laan/csn94-33.ps.gz`.

[Lar92] Peter Gorm LARSEN. « The Dynamic Semantics of the BSI/VDM Specification Language ». Technical Report PGL 1/2.1, IFAD, Lyngby, Danemark, 1992.

[Ler92] Xavier LEROY. « Unboxed Objects and Polymorphic Typing ». In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 177–188, 1992.

[LP92] Shie-Jue LEE and David A. PLAISTED. « Eliminating Duplication with the Hyper-Linking Strategy ». *Journal of Automated Reasoning*, 9(1):25–42, 1992.

[May67] J. Peter MAY. *Simplicial Objects in Algebraic Topology*. Chicago Lectures in Mathematics. The University of Chicago Press, 1967.

[Mel94] Paul-André MELLIÈS. « Typed lambda-calculi with explicit substitutions may not terminate ». In *Proceedings of the CONFER workshop*, München, April 1994.

[Mel95] Paul-André MELLIÈS. « Typed lambda-calculi with explicit substitutions may not terminate ». In M. DEZANI-CIANCAGLINI and G. PLOTKIN, editors, *2nd International Conference on Typed Lambda-Calculi and Applications (TLCA'95)*, pages 328–334, Edinburgh, UK, avril 1995. Springer Verlag LNCS 902.

[Min93] Shin-Ichi MINATO. « Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems ». In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 272–277, Dallas, TX, juin 1993. ACM Press.

[MM96] Simone MARTINI and Andrea MASINI. « A computational interpretation of modal proofs ». In H. WANSING, editor, *Proof Theory of Modal Logic*, pages 213–241. Kluwer, 1996.

[MP91] Zohar MANNA and Amir PNUELI. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991.

[Myc84] Alan MYCROFT. « Polymorphic Type Schemes and Recursive Definitions ». In M. PAUL and B. ROBINET, editors, *Proceedings of the 6th International Symposium on Programming*, pages 217–228, Toulouse, France, 1984. Springer Verlag Lecture Notes in Computer Science 167.

[NO80] Greg NELSON and Derek C. OPPEN. « Fast Decision Procedures Based on Congruence Closure ». *Journal of the ACM*, 27(2):356–364, avril 1980.

[Par73] Rohit PARIKH. « Some Results on the Lengths of Proofs ». *Transactions of the American Mathematical Society*, 177:29–36, 1973.

[Pla95] David PLAISTED. « Special Cases and Substitutes for Rigid E-Unification ». Technical Report MPI-I-95-2-010, Max-Planck-Institut für Informatik, novembre 1995.

[Plo72] Gordon PLOTKIN. « Building in Equational Theories ». *Machine Intelligence*, 7:73–90, 1972.

[Pos93] Joachim POSEGGA. *Deduktion mit Shannongraphen für Prädikatenlo-*

gik erster Stufe. Infix Verlag, Sankt Augustin, 1993.

[PW95] Frank PFENNING and Hao-Chi WONG. « On a Modal λ-Calculus for S4 ». In *11th Conference on Mathematical Foundations of Programming Semantics*, 1995. Extended Abstract.

[Rit94] Mikael RITTRI. « Semi-Unification of Two Terms in Abelian Groups ». *Information Processing Letters*, 52(2):61–68, 1994. Corrigendum in 53(4):235, 1995.

[Rit95] Mikael RITTRI. « Dimension inference under polymorphic recursion ». In *Proceedings of the 7th ACM Conference on Functional Programming Languages and Computer Architecture (FPLCA'95)*, La Jolla, CA, USA, juin 1995.

[Rob65] J. Alan ROBINSON. « A Machine-Oriented Logic Based on the Resolution Principle ». *Journal of the ACM*, 12(1), 1965.

[SC85] A. Prasad SISTLA and Edmund M. CLARKE. « The Complexity of Propositional Linear Temporal Logics ». *Journal of the ACM*, 32(3):733–749, juillet 1985.

[SD95] Iain STUART and Sophia DROSSOPOULOU. « Units and Dimensions in Programming Languages ». Technical Report, Imperial College, novembre 1995. Disponible en `http://hypatia.dcs.qmw.ac.uk/authors/D/DrossopoulouS/papers/UnitsDim.ps.Z`.

[SG89] Wayne SNYDER and Jean GALLIER. « Higher Order Unification Revisited: Complete Sets of Tranformations ». *Journal of Symbolic Computation*, 8(1 & 2):101–140, 1989.

[SGL97a] Peter H. SCHMITT and Jean GOUBAULT-LARRECQ. « A Tableau System for Full Linear-TIME Temporal Logic ». En cours de soumission, 1997.

[SGL97b] Peter H. SCHMITT and Jean GOUBAULT-LARRECQ. « A Tableau System for Linear-TIME Temporal Logic ». In *3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, Université de Twente, Enschede, Pays-Bas, avril 1997. Springer Verlag.

[Sim85] Steven G. SIMPSON. « Reverse Mathematics ». In A. NERODE and R. A. SHORE, editors, *Recursion Theory*, pages 461–471. AMS, 1985. Proceedings of Symposia in Pure Mathematics, vol. 42.

[Sti88] Mark E. STICKEL. « A Prolog Technology Theorem Prover ». In E. LUSK and R. OVERBEEK, editors, *9th International Conference on Automated Deduction (CADE-9)*, volume 310 of *Lecture Notes in Computer Science*, pages 752–753, Argonne, Illinois, USA, mai 1988. Springer Verlag.

[SW49] Claude E. SHANNON and Warren WEAVER. *The Mathematical Theory of Communication*. Illinois University Press, 1949. Réédité par Illini Books, 1963.

[Vea97] Margus VEANES. « The undecidability of simultaneous rigid E-unification with two variables ». In *Proceedings of the 5th Kurt G-del Colloquium (KGC'97)*. Springer Verlag Lecture Notes in Computer Science, 1997.

[WF86] Mitchell WAND and Daniel P. FRIEDMAN. « The Mystery of the Tower Revealed: A Non-Reflexive Description of The Reflexive Tower ≫. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 298–307, 1986. Extended Abstract.

[WO91] Mitchell WAND and Patrick O'KEEFE. ≪ Automatic Dimensional Inference ≫. In J.-L. LASSEZ and G. PLOTKIN, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 479–483. MIT Press, 1991.

Table des matières