ORSAY
N$^o$ d'ordre : 910

UNIVERSITÉ DE PARIS-SUD 11

CENTRE D'ORSAY

# THÈSE

présentée
pour obtenir

L'HABILITATION A DIRIGER DES RECHERCHES
DE L'UNIVERSITÉ PARIS XI

PAR

Claude MARCHÉ

—⋆—

SUJET :

**Preuves mécanisées de propriétés de programmes**

soutenue le 12 décembre 2005 devant la commission d'examen

| | | |
|---|---|---|
| MM. | Joffroy BEAUQUIER | Président |
| | Gilles BARTHE | Rapporteurs |
| | Jürgen GIESL | |
| | Claude KIRCHNER | |
| | Pierre LESCANNE | Examinateurs |
| | K. Rustan M. LEINO | |

# Chapitre 1

# Introduction

Mon travail de recherche se situe dans le domaine de la démonstration de théorèmes par ordinateur. La démonstration de théorèmes mathématiques est devenu un objet de recherche en soit depuis la seconde moitié du 19e siècle, lorsque les mathématiciens ont commencé à se poser des questions fondamentales sur la cohérence de leurs raisonnements. Puis, dans la seconde moitié du 20e siècle, avec l'invention de l'ordinateur, on a eu la possibilité d'utiliser celui-ci pour *mécaniser* les preuves, c'est-à-dire d'utiliser le caractère mécanique de l'ordinateur pour produire des preuves sans erreur, du moins l'espère-t-on. Enfin, les preuves mécanisées ont été mises en œuvre non plus pour prouver des théorèmes mathématiques mais des propriétés des programmes exécutés sur un ordinateur.

## 1.1 Preuve formelle en mathématiques

C'est donc dans la seconde moitié du 19e siècle que les mathématiciens ont commencé à s'intéresser à la démonstration de théorèmes en tant qu'objet d'étude, et ont cherché à formaliser ce qu'est une preuve correcte. Hilbert, avec ses *fondements de la géométrie*, a souhaité expliciter tous les axiomes utilisés dans les raisonnements géométriques. La théorie des ensembles proposée par Cantor et Frege visait à définir précisément les énoncés mathématiques vrais.

Hilbert pensait que l'on devrait pouvoir mécaniquement décider si un énoncé est une vérité mathématique ou non. Dans son fameux programme de 23 problèmes à résoudre pour le 20e siècle, énoncé en 1900, il a inclut une partie de ce *programme formaliste*. Le deuxième problème de Hilbert était d'établir la cohérence de l'arithmétique, et le dixième problème demandait d'énoncer un procédé mécanique de résolution des équations diophantiennes arbitraires. Ces deux problèmes sont a priori assez éloignés mais en fait, leurs réponses, toutes deux négatives, sont intimement liées.

Les tentatives de formalisation ont abouti à des échecs inattendus. Russell a exhibé en 1902 une contradiction dans le système de Cantor et Frege — le paradoxe de Russell — qui apparaît si l'on a le droit d'écrire une formule comme $x \in x$, un ensemble qui appartient à lui-même. Des solutions au paradoxe de Russell ont été proposées : avec Whitehead, Russell lui-même proposa la théorie des types (dite aujourd'hui théorie des types simples,

étendue ensuite par Martin-Löf) ; et Zermelo en 1908 proposa une théorie des ensembles plus élaborée. Un fait marquant fut apporté par Gödel en 1931, qui démontra d'abord l'impossibilité d'énumérer récursivement les énoncés vrais de l'arithmétique, puis ensuite et par conséquent l'impossibilité de démontrer sa cohérence. Enfin, en 1936, Turing introduisit ses machines, et montra l'impossibilité de décider la terminaison de l'exécution de telles machines[138, 139].

La mécanisation des preuves semblait alors être une utopie irréalisable.

## 1.2 Preuve mécanisée par ordinateur

Avec l'invention des premiers calculateurs électroniques, puis les progrès rapides en matière de programmation des ordinateurs, un regain d'intérêt pour la preuve mécanique eut lieu. Deux familles d'approches ont vu le jour.

La première famille d'approches est celle de la démonstration automatique, pour des logiques suffisamment simples pour être décidables ou semi-décidables. En 1961, Davis et Putnam [39] proposent un algorithme de décision de la satisfaisabilité de formules propositionnelles. En 1965, Robinson [125] propose le mécanisme de résolution pour raisonner sur les clauses du premier ordre, puis en 1969 avec Wos ils proposent la paramodulation, pour traiter le prédicat d'égalité. En 1970, Knuth et Bendix [84] proposent leur algorithme de complétion pour les problèmes purement égalitaires, Huet montre en 1980 que cela était une procédure de semi-décision. En 1980 encore, Downey, Sethi et Tarjan [51] et indépendemment Nelson et Oppen [114], proposent l'algorithme appelé ensuite *clôture de congruence*, pour décider les égalités closes. Ces recherches ont débouché aujourd'hui sur des nombreux outils effectifs de démonstration. Les compétitions CASC (affiliée à la conférence CADE) et SMT (affilié à la conférence CAV) font concourir et progresser ces outils. On peut citer un succès de ces outils : la conjecture de Robbins, que les mathématiciens ne savaient pas prouver, a été résolue automatiquement par McCune et son prouveur EQP en 1996 (`http://www-unix.mcs.anl.gov/~mccune/papers/robbins/`). Le prouveur automatique *Simplify*, issu de la thèse de Greg Nelson en 1980, a été et est encore utilisé intensivement par des outils de vérification comme ESC/Java [49, 1, 2].

La seconde famille d'approches est celle de la preuve interactive. Dans ce cas, la logique considérée peut être arbitrairement complexe et donc expressive, mais par contre on ne cherche pas à établir automatiquement la validité d'un énoncé, au contraire on attend d'un utilisateur humain de décrire les étapes de raisonnement, le rôle de l'ordinateur étant alors de vérifier que les étapes de raisonnement sont correctes. Le premier système selon ce principe fut Automath en 1968. Le système LCF en 1969 introduit l'idée de tactique pour construire les preuves. Le prouveur Nqthm de Boyer et Moore est créé en 1971, et son descendant ACL2 est encore beaucoup utilisé aujourd'hui. D'autres systèmes d'aide à la preuve ont vu le jour à cette époque, comme ceux basés sur la théorie des types simples : PVS (`http://pvs.csl.sri.com/`, HOL (`http://www.cl.cam.ac.uk/Research/HVG/HOL/`) et Isabelle (`http://www.cl.cam.ac.uk/Research/HVG/Isabelle/` ; également encore utilisés aujourd'hui, et même de plus en plus. Le système Coq (`http://coq.inria.fr/`), qui commence à être déve-

loppé en 1985, va plus loin encore, car il utilise une théorie des types pour laquelle les preuves sont des objets du langage ; un aspect essentiel pour mes travaux de recherche. Pour montrer un des succès de ces approches, on peut citer le théorème des quatre couleurs, qui a été entièrement prouvé en Coq par Gonthier et Werner en 2005 (`http://www.inria.fr/actualites/2005/theoreme4couleurs.en.html`).

## 1.3 Preuve de programmes

Les progrès obtenus pour la preuve par ordinateur ont permis de traiter mécaniquement de nombreux énoncés mathématiques. Mais l'invention des ordinateurs et leur programmation ont donné le jour à une autre classe d'énoncés : les propriétés des programmes eux-mêmes.

Dans un article de 1949 qui a peu marqué à son époque [140], mais depuis réhabilité et commenté par Jones et Morris en 1984 [113], Turing se pose pour la première fois la question d'établir formellement la preuve qu'un programme effectue bien la tâche qui lui est confiée. Il énonce des formules logiques portant sur les valeurs successives prises par les variables du programme, qui expriment la tâche voulue. Il établit la validité de cette *spécification* en passant en revue les transitions du programme. Il n'oublie pas également de prouver la terminaison de son programme pour toutes les valeurs de ses entrées.

Floyd en 1967 [62] puis Hoare en 1969 [71] introduisent la logique de Floyd-Hoare, dont le principe est de poser des *préconditions* et des *post-conditions* sur la ou les procédures du programme considéré. Ce principe est encore actuel.

Au contraire de l'approche de Floyd-Hoare, les programmes que l'on spécifie et que l'on prouve avec Coq, PVS ou les systèmes analogues, sont essentiellement des programmes fonctionnels, c'est-à-dire sans effet de bord. En 1998, Jean-Christophe Filliâtre [56] propose une approche pour utiliser les systèmes basés sur une théorie des types pour prouver des programmes impératifs, ce travail est le point de départ de mes travaux sur ce sujet, et l'approche en question sera détaillée dans ce document.

## 1.4 La vérification de programmes aujourd'hui

De nos jours, le logiciel a pris une place très importante dans la vie quotidienne, parfois de manière tellement discrète que nous n'en avons pas forcément conscience.

Dans certaines situations, la vie humaine dépend du bon fonctionnement de logiciels : dans le domaine des transports en premier lieu, avec le logiciel qui assure le bon fonctionnement des avions, des trains et même de plus en plus des voitures particulières (exemple des régulateurs de vitesse). En second lieu, l'informatique est devenu un instrument de médecine à part entière, et il est certain qu'à l'avenir l'informatique, par l'intermédiaire de robots, elle sera de plus en plus amenée à assister le chirurgien lors des opérations qu'il effectue. Les industries de pointe en général font appel au logiciel pour surveiller et contrôler les installations, pour le nucléaire par exemple.

Le logiciel occupe maintenant la place centrale dans le domaine financier et la gestion : même si dans ce cas la vie humaine n'est pas directement menacée par un dysfonction-

nement éventuel, les conséquences économiques pourraient être si importantes qu'il est néanmoins crucial de prévenir de tels dysfonctionnements.

Enfin, en particulier depuis les toutes dernières années du siècle dernier, le logiciel a permis un développement extrêmement rapide de nouveaux moyens de communications, en particulier la téléphonie mobile, mais aussi Internet, concept qui bien qu'étant ancien, s'est démocratisé seulement quand le grand public a pu accéder aux moyens technologiques nécessaires.

L'accroissement très rapide de la place occupée par le logiciel dans la vie quotidienne suscite un intérêt grandissant pour les méthodes qui, dans le développement de ces logiciels, offrent des vérifications de sécurité de fonctionnement. Les méthodes existantes ne sont pas toutes de même nature. Une première famille de méthodes passe par la simulation : on teste sur le programme sur des valeurs particulières des données. Une seconde famille de méthodes par une approximation du programme, autrement dit une abstraction :

- Les techniques dites *d'interprétation abstraite* approximent un programme afin de calculer les intervalles possibles de valeurs des variables, dans le but de détecter des erreurs : si une variable peut prendre une valeur non autorisée, un avertissement est signalé. Comme il s'agit d'une approximation, des avertissements peuvent être produits de manière infondé, ce sont des « faux-positifs ». Au contraire, les erreurs qui seraient d'un autre ordre que le fait de sortir d'un intervalle prévu ne sont pas détectées. Ces méthodes ont eu des succès important, par exemple le système Astrée [18] a été utilisé pour les programmes embarqués de l'Airbus A380.
- Les techniques de *model-checking* approximent un programme en un graphe fini de transitions. Des propriétés peuvent alors être établies et surtout invalidées par exploration exhaustive du graphe. Mais comme il s'agit d'une abstraction, des erreurs peuvent exister au niveau du programme et disparaître par l'abstraction. Le model-checking ne garantit donc pas la correction du programme, par contre ces techniques ont beaucoup de succès car elles détectent bien les erreurs.

Ces méthodes approchées se caractérisent par leur automatisation complète, un aspect qui leur permet d'être pris en main et adopté rapidement par les utilisateurs, en particulier industriels.

L'approche que j'ai suivie est différente : on ne souhaite pas faire d'approximation, mais procéder par preuve formelle, en suivant le principe de la logique de Floyd-Hoare. L'objectif est d'avoir des méthodes qui assure la correction, c'est-à-dire de ne pas laisser passer d'erreur. Le prix à payer est que certaines preuves doivent être faites interactivement.

## 1.5 Résumé de mes contributions

Au cours de mes activités de recherche, je me suis essentiellement intéressé d'une part à la propriété de terminaison, et d'autre part aux propriétés fonctionnelles de programmes impératifs, spécifiquement les programmes sources C et Java. Ces deux thèmes font l'objet des deux chapitres centraux de ce document. Voici un résumé de mes résultats : sur les techniques de terminaison :

- Extension du critère des paires de dépendance au cas des symboles associatifs et/ou

commutatifs (section 2.2, [106, 90]) ;
- Utilisation des ordres polynomiaux dans le contexte du critère des paires de dépendances, et recherche automatique efficace de solutions (section 2.3, [34]) ;
- Méthode incrémentale de preuves de terminaison, pour arriver à traiter les systèmes de grande taille, c'est le résultat principal de la thèse de Xavier Urbain que j'ai encadrée (section 2.4, [143, 142, 144, 107])
- Preuve de terminaison de programmes exprimés dans d'autres formalismes que les systèmes de réécriture (section 2.5), en se ramenant à ceux-ci : en préliminaire les systèmes de réécriture conditionnelle (section 2.5.1, [99]), puis les programmes logiques (section 2.5.2, [122, 123]) et programmes équationnels avec conditions d'appartenance (section 2.5.3, [53, 52])

et sur les preuves de programmes impératifs :
- Modélisation de la mémoire puis schéma de traduction fonctionnelle des programmes Java [105, 104].
- Modélisation de la mémoire, supportant l'arithmétique de pointeurs, et schéma de traduction des programmes C [61]
- Études de cas sur des applettes JavaCard [80, 28]
- Études de cas sur des programmes C [75]

## 1.6 Organisation de ce mémoire

Le chapitre 2 présente mes contributions sur la preuve automatique de terminaison. Le chapitre 3 présente mes contributions sur la preuve de programmes impératifs. Chacun de ces chapitres se termine par une conclusion sur le thème concerné. Le chapitre 4 présente une conclusion globale, des perspectives à court ou moyen terme sur les deux axes de recherche précédents, et des perspectives à long terme.

Ce document ne contient pas de détails techniques sur l'obtention des résultats théoriques fondamentaux, mais plutôt des présentations par des exemples des problèmes abordés et comment ils ont été traités. En effet, lors de mes recherches j'ai toujours cherché à mettre en pratique les résultats théoriques obtenus.

# Chapitre 2

# Preuves de terminaison

Ce chapitre aborde donc le premier de mes deux principaux axes de recherche : la preuve mécanisée de terminaison.

## 2.1 Contexte

### 2.1.1 Historique

Comme on l'a vu dans le chapitre introductif, la preuve de terminaison de programmes est un souci ancien, puisqu'il est mentionné par Alan Turing en 1949 [140], et bien entendu c'est également Turing qui a montré l'indécidabilité de la terminaison en général [138, 139].

Néanmoins, la grande diversité des formalismes de calculs a fait que pendant longtemps aucune approche générale n'a été proposée : la seule technique générale était le principe issu des mathématiques qui consiste à prouver la finitude d'une séquence en associant à chacun de ses termes une mesure qui décroit strictement, à valeur dans les entiers positifs ou dans un autre domaine bien fondé.

En 1970, un article marquant dû à Zohar Manna et Stephen Ness [100], s'intéresse à la preuve de terminaison « d'algorithmes de Markov », qui s'avèrent être en fait ce qu'on appelle communément aujourd'hui des systèmes de réécriture. C'est dans cet article que la preuve de terminaison est réduite à un nombre fini de contraintes de la forme $s > t$, pour des termes $s$ et $t$ donné, pour lesquelles on doit chercher une relation d'ordre $>$ qui soit un ordre dit *de réduction* : il doit satisfaire les trois propriétés suivantes :

1. être bien fondé : il n'admet pas de suite infinie décroissante $s_1 > s_2 > s_3 \cdots$ ;
2. être *monotone* : si $s > t$ alors $C[s] > C[t]$ pour tous termes $s$ et $t$ et pour tout contexte $C[.]$ ;
3. être *stable par instanciation* : si $s > t$ alors $s\sigma > t\sigma$ pour tous termes $s$ et $t$ et toute substitution $\sigma$.

Ce principe consistant à ramener la preuve de terminaison à une nombre fini de contraintes devant être satisfaites par un ordre satisfaisant des propriétés additionnelles (ici être un ordre de réduction) est fondateur de toutes les techniques qui vont suivre. Néanmoins, cet article

n'est pas orienté *a priori* vers la mécanisation, puisque aucune méthode n'est proposée pour rechercher un ordre de réduction.

En 1970 également, un autre article marquant est paru, dû à Donald Knuth et Bendix [84], dans lequel est introduit pour la première fois la notion de procédure de *complétion*. Cet article ne s'intéresse pas en priorité à la terminaison mais plutôt à l'autre propriété fondamentale que l'on recherche pour un système de réécriture, la *confluence*, ceci dans le but de décider le *problème du mot* dans des théories équationnelles. Néanmoins, l'algorithme de complétion de Knuth et Bendix a besoin pour fonctionner de la donnée d'un ordre de réduction (au sens précédent). Pour traiter des exemples concrets, Knuth et Bendix introduisent une construction de tels ordres, à partir de la donnée d'une *précédence* (c'est-à-dire un ordre sur les symboles de fonctions) et de poids (des entiers) pour chaque symbole de la signature. La famille d'ordre de réduction ainsi obtenue est restée sous le nom d'ordres de Knuth-Bendix (KBO). Cet article ne donne néanmoins aucune approche pour rechercher automatiquement une précédence et des poids appropriés.

Le début des années 1970 a ainsi mis en avant la réécriture comme un formalisme de calcul suffisamment simple et général, et surtout de nature purement syntaxique, qui en a fait le formalisme le mieux adapté au développement de théories et de techniques pour décider la terminaison d'algorithmes.

En 1978, Nachum Dershowitz et Zohar Manna proposent une nouvelle classe d'ordre qui seront par la suite dénommés *multiset path orderings* (MPO) [47, 42]. Ces ordres sont paramétrés par une précédence, et les termes sont comparés par un algorithme récursif sur leurs sous-termes, ce qui en fait une classe d'ordres dits *syntaxiques*, et facile à programmer (bien qu'une attention particulière doit être de mise pour contrôler la complexité du test de comparaison). Une variante de cette classe d'ordres de réduction a ensuite été proposée par Kamin et Lévy en 1980 [81], en permettant de comparer les sous-termes non pas en tant que multi-ensemble mais comme des séquences ordonnées lexicographiquement : les *lexicographic path orderings* (LPO). Ces deux classes ont été finalement généralisées en une classe dite des *recursive path orderings* (RPO) par Dershowitz [43], où chaque ordre est paramétré par une précédence ainsi que la donnée, pour chaque symbole de la signature, d'un statut pouvant être multiset, lexicographique de gauche à droite, ou de droite à gauche. Comme sur une signature finie donnée, il n'y a qu'un nombre fini de précédences possibles, et également un nombre fini de statuts, on voit immédiatement qu'il est décidable de savoir si un système de réécriture donné est inclus dans un RPO. Pour rechercher si un système de réécriture est inclus dans un RPO, il existe évidemment des méthodes plus efficaces pour rechercher un RPO convenable que d'énumérer les précédences et les statuts, en procédant plutôt par une résolution de contraintes à partir des termes à comparer, et une telle méthode a été implantée en 1983 dans le logiciel REVE [95, 63]. Ce logiciel peut être considéré comme le premier outil permettant une preuve automatique de terminaison d'un système de réécriture.

Un autre article pionnier publié en 1979 est celui de Dallas S. Lankford [94], qui est considéré comme initiateur de l'approche dite par *interprétations polynomiales*. Il propose une classe d'ordre de réduction, où chaque ordre peut être défini par la donnée, pour chaque symbole de fonction $f$ d'arité $n$, d'un polynôme à $n$ indéterminées. Cet article donne des conditions suffisantes sur les coefficients pour que l'ordre obtenu soit un ordre de réduction.

Néanmoins, une fois de plus, il ne donne aucune méthode pour rechercher automatiquement des polynômes qui pourraient convenir.

En 1986, Ahlem Ben Cherifa et Pierre Lescanne [17, 16] ont proposé une condition suffisante de positivité d'un polynôme, mécanisable, ce qui a donné lieu à la première implantation de la vérification d'une preuve de terminaison par interprétations polynomiales. Ils ont par ailleurs établi une condition nécessaire et suffisante (et toujours utilisée aujourd'hui, par exemple dans C*i*ME) de compatibilité de l'ordre engendré avec l'associativité et la commutativité, par contre il n'y avait toujours aucune approche proposé pour la recherche automatique d'interprétations adéquates. En 1990, Jocelyne Rouyer et Pierre Lescanne ont proposé un critère de positivité beaucoup plus élaboré, basé sur le théorème de Sturm [126], mais dont la complexité de lui a pas permis de s'imposer. D'autre part, l'approche a été étendue par Pierre Lescanne en 1992 [96] aux interprétations *élémentaires*, c'est-à-dire avec des fonctions exponentielles.

Des avancées importantes ont été accomplies par Joachim Steinbach à partir de 1992 [131, 132] : on lui doit l'idée de la recherche automatique de polynômes de forme particulière (les linéaires, les polynômes *simples* définis ci-après 2.3.3, les *simple-mixed*) et un algorithme de recherche automatique de polynômes appropriés, basé sur l'inégalité des moyennes arithmético-géométrique, qui permet une linéarisation des inégalités à obtenir. Ce dernier algorithme est dépassé aujourd'hui, mais la classification des formes de polynômes reste d'actualité. Il faut aussi noter que J. Steinbach a pu, grâce à sa méthode, prouver automatiquement la terminaison d'une large collection de systèmes de réécriture.

En 1995, Jürgen Giesl [65] s'est attaqué au problème de la recherche automatique d'ordre par interprétations polynomiales, et a proposé deux nouveautés. Premièrement, des conditions suffisantes de positivité meilleures que celles de Ben Cherifa et Lescanne et de Steinbach, basées sur les dérivées successives des polynômes. Deuxièmement, une méthode de recherche de polynômes adéquats basée sur la résolution de contraintes polynomiales dans les réels au lieu des entiers, problème qui contrairement au cas des entiers est en théorie décidable [134] mais néanmoins d'une complexité algorithmique très élevée [38], et cette variante a montré en pratique très peu d'amélioration par rapport à la technique de Steinbach.

### 2.1.2 La situation en 1997

En 1997, le logiciel CiME implantant la technique de complétion normalisée proposée dans ma thèse de Doctorat [101, 102, 31] a commencé à être re-structuré pour en faire une boite à outils pour la réécriture, fournissant de multiples opérations. Il s'est avéré en particulier utile d'implanter des techniques de recherche de preuve de terminaison, et nous avons implanté la famille des RPO. En effet, à cette date plusieurs raisons faisaient que la classe des ordres par interprétations polynomiales étaient moins populaires que les méthodes syntaxiques comme RPO. D'un point de vue théorique, la classe des systèmes de réécriture pouvant être inclus dans un ordre par interprétations polynomiale est restreinte : premièrement, la longueur des dérivations a une borne doublement exponentielle [72] ; deuxièmement, la fonction calculée appartient nécessairement à une classe de complexité réduite [29, 21]. Par

exemple, la terminaison de la fameuse fonction de Ackermann-Peter :

$$\begin{aligned} A(0,y) &\rightarrow s(y) \\ A(s(x),0) &\rightarrow A(x,s(0)) \\ A(s(x),s(y)) &\rightarrow A(x,A(s(x),y)) \end{aligned}$$

est aisément prouvée avec le LPO engendré par la précédence $A > s$ et le statut lexicographique gauche-droite pour $A$, mais il n'existe pas d'interprétation polynomiale adéquate. D'un point de vue pratique, les ordres basés sur des précédences sont plus simples à implanter, la recherche automatique est décidable, alors que la recherche d'ordre par interprétations polynomiales est nécessairement incomplète.

Néanmoins, il existe aussi des exemples de systèmes de réécriture pouvant être prouvés terminant par des ordres par interprétations, mais pas par un RPO. Un exemple est le calcul de l'addition des entiers binaires[1] :

$$\begin{aligned} \#0 &\rightarrow \# \\ x+\# &\rightarrow x \\ \#+x &\rightarrow x \\ x0+y0 &\rightarrow (x+y)0 \\ x0+y1 &\rightarrow (x+y)1 \\ x1+y0 &\rightarrow (x+y)1 \\ x1+y1 &\rightarrow ((x+y)+\#1)0 \end{aligned}$$

que nous avons proposé [33], mais aussi de fameux exemples de lambda-calculs avec substitutions explicites sur lesquels nous avons travaillé [141], qui nous ont donc demandé d'implanter dans CiME les techniques d'ordres par interprétations.

En 1997 eut lieu une petite révolution dans les techniques de preuve de terminaison de la réécriture : la méthode des paires de dépendance, proposée par Thomas Arts et Jürgen Giesl [6, 7], offrît une alternative au critère de terminaison de Manna et Ness, et permit d'élargir de façon très significative la collection des systèmes de réécriture pouvant être automatiquement prouvés terminant.

Enfin, en 1998, Hoon Hong et Dalibor Jakuš [73] ont repris les travaux précédents sur les critères de positivité de polynômes, et ont établi une hiérarchie claire sur la puissance des différentes méthodes, à l'exception notable de la méthode de Rouyer [126] qui ne semblait pas être connue de ces auteurs. En gros, il faut considérer que la méthode des dérivées successives de Giesl est meilleure que les précédentes de Steinbach et Ben Cherifa-Lescanne, et résultat surprenant mais essentiel, est équivalente aussi bien en puissance qu'en complexité algorithmique à la méthode triviale qui regarde seulement si tous les coefficients sont positifs (après « translation » du polynôme en 0, voir ci-après à la section 2.3.2). Ce résultat est essentiel pour la méthode de recherche de polynômes utilisée de nos jours par C*i*ME, et sera présenté formellement dans la section 2.3.3.

[1] la constante $\#$ représente $0$, les deux constructeurs unaires $0$ et $1$ sont notés de façon postfixe, où $(x)0$ représente $2x$ et $(x)1$ représente $2x+1$. Ainsi 6 est codé par $\#110 : 2 \times (2 \times (2 \times 0 + 1) + 1)$. Le codage d'un nombre est unique si l'on supprime les zéros inutiles au début, d'où la règle $\#0 \rightarrow \#$.

Ces deux dernières avancées — critère des paires de dépendance et critère de positivité de Hong et Jakuš — nous ont permis de de démarrer de fructueuses activités de recherche sur la preuve automatique de terminaison, avec pour objectif d'être capable de traiter efficacement des cas « pratiques » : d'une part, de nombreux exemples de système de réécriture de la littérature ne pouvaient pas encore être automatiquement prouvés terminants, et d'autre part la puissance accrue de ces techniques de terminaison de la réécriture permet enfin d'envisager de procéder à la preuve de terminaison de programmes exprimés dans d'autre formalismes, en procédant par traduction vers la réécriture. Nous avons ainsi étudié la preuve de terminaison de programmes Prolog, et de programmes équationnels avec conditions d'appartenance, formalisme utilisé dans la plateforme Maude [30]. Les exemples ainsi étudiés avaient une taille (en terme de nombre de règles principalement) de plus en plus grande, et pour résoudre des problèmes de passage à l'échelle de nos techniques, nous avons été amené à proposer de nouveaux critères de terminaison, permettant des preuves *modulaires* (c.-à-d. où la terminaison d'un système se ramène à la terminaison de ses composants plus petits) et *incrémentales* (c.-à-d. où lorsqu'on ajoute un composant à un système déjà prouvé terminant, on prouve la terminaison de l'ensemble en considérant seulement des contraintes sur l'incrément).

### 2.1.3 Résultats obtenus

Les résultats que nous avons obtenus se déclinent en quatre points :

- Extension du critère des paires de dépendance au cas des symboles associatifs et/ou commutatifs (section 2.2, [106, 90]) ;
- Utilisation des ordres polynomiaux dans le contexte du critère des paires de dépendances, et recherche automatique efficace de solutions (section 2.3, [34]) ;
- Méthode incrémentale de preuves de terminaison, pour arriver à traiter les systèmes de grande taille, c'est le résultat principal de la thèse de Xavier Urbain que j'ai coencadrée (section 2.4, [143, 142, 144, 107])
- Preuve de terminaison de programmes exprimés dans d'autres formalismes que les systèmes de réécriture (section 2.5), en se ramenant à ceux-ci : en préliminaire les systèmes de réécriture conditionnelle (section 2.5.1, [99]), puis les programmes logiques (section 2.5.2, [122, 123]) et programmes équationnels avec conditions d'appartenance (section 2.5.3, [53, 52])

Les problèmes ouverts et les enjeux futurs seront passés en revue dans la section 2.6 et le chapitre 4.

## 2.2 Généralités sur la terminaison et les ordres de réduction

Nous adoptons les définitions et notations standards en réécriture [46, 10, 135]. On note $T(F, X)$ l'ensemble des termes du premier ordre sur une signature $F$ et un ensemble de variables $X$. On note $t\sigma$ l'application d'une substitution $\sigma$ à un terme $t$.

### 2.2.1 Paires d'ordres de réduction

La termination des systèmes de réécriture est traditionnellement, en fait depuis le critère de Manna et Ness en 1970 [100] prouvée à l'aide d'ordres stricts de *réduction* [44], c'est-à-dire d'une relation d'ordre strict qui soit bien-fondée, stable par instanciation et monotone.

**Proposition 2.1 (Manna, Ness, 1970)** *Un système de réécriture $R$ termine ssi il existe un ordre de réduction $>$ tel que $l > r$ pour toute règle $l \to r$ de $R$.*

La popularisation de la technique des paires de dépendance a nécessité le besoin de comparer des termes à la fois au sens large et au sens strict, mais d'un point de vue pratique une difficulté survient : dans une telle situation on n'utilise pas un ordre $\succeq$ et sa partie stricte définie par $t_1 \succ t_2$ si $t_1 \succeq t_2$ et $t_2 \not\succeq t_1$. En effet, l'ordre strict ainsi défini n'est pas stable en général, comme le montre l'exemple suivant.

**Exemple 2.2** *Pour une signature contenant une constante $a$, et pour un ordre fonctionnant par interprétation dans les entiers naturels avec $[\![a]\!] = 0$, on aura, pour toute variable $x$, $x \succeq a$ et $a \not\succeq x$, et si on prend pour $\succ$ la partie stricte associée à $\succeq$, on aura $x \succ a$. Mais alors $\succ$ ne peut pas être stable car si l'on instancie $x$ par $a$, cela donnerait $a \succ a$.*

Une solution possible serait d'utiliser pour $\succ$ la *partie stricte stable* [117] de $\succeq$, c.-à-d. $t_1 \succ t_2$ si pour toute substitution $\sigma$, $t_1\sigma \succeq t_2\sigma$ et $t_2\sigma \not\succeq t_1$. Mais alors cette relation n'est plus calculable en général, et donc pour la mécanisation des preuves, il nous faut prendre une relation calculable $>$ incluse dans la partie stricte stable de $\succeq$.

**Définition 2.3** *Une* paire d'ordres compatibles *est un couple $(\succeq, >)$ de relations sur un ensemble telles que :*
- *$\succeq$ est un préordre, c.-à-d. une relation réflexive et transitive ;*
- *$>$ est un ordre strict, c.-à-d. une relation anti-réflexive et transitive ;*
- *$>$ est* compatible *avec $\succeq$, c'est-à-dire pour tous $x, y, z, t$, si $x \succeq y > z \succeq t$ alors $x > t$.*

Pour alléger le vocabulaire, nous continuerons à appeler *ordre sur les termes* une paire d'ordres compatibles sur une algèbre de termes.

**Définition 2.4** *Un ordre sur les termes $(\succeq, >)$ est dit :*
- bien-fondé *s'il n'existe par de suite infinie décroissante $t_1 > t_2 > \cdots$ ;*
- stable *si à la fois $>$ et $\succeq$ sont stables par substitution, c.-à-d. pour tous termes $t_1$ et $t_2$ et toute substitution $\sigma$, si $t_1 > t_2$ alors $t_1\sigma > t_2\sigma$, et si $t_1 \succeq t_2$ alors $t_1\sigma \succeq t_2\sigma$ ;*
- faiblement monotone *si pour tous termes $t_1$ et $t_2$, tout contexte $C[.]$, si $t_1 \succeq t_2$ alors $C[t_1] \succeq C[t_2]$ ;*
- strictement monotone *si pour tous termes $t_1$ et $t_2$, pour tout contexte $C[.]$, si $t_1 > t_2$ alors $C[t_1] > C[t_2]$.*

*Un ordre sur les termes est appelé :*
- *un* ordre de réduction faible *s'il est bien-fondé, stable et faiblement monotone ;*
- *un* ordre de réduction strict *s'il est bien-fondé, stable et à la fois faiblement et strictement monotone.*

Cette notion d'ordre de réduction faible est un cas particulier de la notion de *paire de réduction faible* (*weak ordering pair*) proposée en 1999 par Kusakari et Toyama [91]. Les conditions de la définition de Kusakari et Toyama sont très générales (en particulier la transitivité des relations n'est pas exigée), mais pour notre objectif de mécanisation, notre cas particulier suffit.

### 2.2.2 Critères des paires de dépendance

**Définition 2.5** *Soit $R$ un système de réécriture sur la signature $F$. On appelle* symboles définis *de $R$ les symboles qui apparaissent au moins une fois en tête d'un membre gauche d'une règle de $R$. Les autres sont appelés symboles* constructeurs.

*Une* paire de dépendance *de $R$ est un couple $\langle u, v\rangle$ tel que $u \to w$ est une règle de $R$ et $v$ un sous-terme de $w$ dont le symbole de tête est défini. L'ensemble des paires de dépendances de $R$ est noté $DP(R)$.*

*On note $\hat{F}$ la signature obtenue en ajoutant à $F$ une copie $\hat{f}$ de chaque symbole défini $f$. Une paire de dépendance marquée est une paire de termes sur $T(\hat{F}, X)$ obtenue à partir d'une paire de dépendance « normale » $\langle u, v\rangle$ en marquant les symboles de tête de $u$ et $v$.*

*Une* chaîne de dépendance *est une suite de paires de dépendance, munie d'une substitution $\sigma$, tel que pour toutes paires consécutives $\langle u_i, v_i\rangle$ et $\langle u_{i+1}, v_{i+1}\rangle$,*

$$v_i\sigma(\xrightarrow[R]{\neq\Lambda})^* u_{i+1}\sigma$$

*dans cette définition, $\neq\Lambda$ signifie que les réécritures n'ont jamais lieu à la racine. De plus, on suppose implicitement que les variables apparaissant dans les paires ont été renommées pour n'apparaître que dans une paire au plus, ceci évite d'introduire une substitution $\sigma_i$ par pas. Une telle chaîne de dépendance est dite minimale si $\sigma$ est fortement normalisable pour $R$.*

*Les chaînes de dépendance marquées, éventuellement minimales, sont définies de la même façon en utilisant les paires marquées.*

Le critère des paires de dépendance est dû à Arts et Giesl en 1997 [9] :

**Proposition 2.6 (Arts & Giesl, 1997)** *$R$ termine ssi il n'existe pas de chaîne de dépendance (marquée ou non, minimale ou non) infinie.*

Ce critère induit une technique pour prouver la terminaison à l'aide d'un ordre de réduction sur les termes (implicitement une paire d'ordre).

**Proposition 2.7** *$R$ termine ssi il existe un ordre de réduction faible sur $T(F, X)$ (resp. sur $T(\hat{F}, x)$) tel que $l \succeq r$ pour toute règle $l \to r$ de $R$ et $u > v$ pour toute paire de dépendance (resp. paire marquée).*

**Exemple 2.8** *Pour montrer la terminaison du système à une règle*

$$f(f(x)) \quad \to \quad f(g(f(x)))$$

*on calcule d'abord ses paires de dépendance non-marquées : $f$ est défini et $g$ est un constructeur, donc les paires de dépendance sont au nombre de deux :*

$$\begin{array}{lll} \langle f(f(x)) & , & f(g(f(x)))\rangle \\ \langle f(f(x)) & , & f(x)\rangle \end{array}$$

*nous devons donc trouver un ordre tel que*

$$\begin{array}{lcl} f(f(x)) & > & f(g(f(x))) \\ f(f(x)) & > & f(x) \\ f(f(x)) & \succeq & f(g(f(x))) \end{array}$$

*Une solution est d'utiliser une interprétation dans les entiers définie récursivement par :*

$$\begin{array}{lcl} [\![f(x)]\!] & = & [\![x]\!] + 1 \\ [\![g(x)]\!] & = & 0 \end{array}$$

*L'interprétation de $f(f(x))$ est alors $[\![x]\!] + 2$, supérieure à celle de $f(g(f(x)))$ qui est $1$, etc. On verra en section 2.3 pourquoi cela définit bien un ordre de réduction faible.*

### 2.2.3 Critères des graphes de dépendance

C'est un raffinement qui permet de relaxer les contraintes de terminaison à résoudre, et qui s'avère très efficace pour la mécanisation.

**Définition 2.9** *On appelle* graphe de dépendance *d'un système de réécriture $R$ le graphe orienté dont :*

- *les sommets sont les paires de dépendances de $R$ ;*
- *il y a une arête de $\langle u_1, v_1\rangle$ vers $\langle u_2, v_2\rangle$ ssi il existe une substitution $\sigma$ telle que*

$$v_1\sigma \xrightarrow[R]{\neq\Lambda}{}^{*} u_2\sigma$$

La proposition suivante est inspirée de [8], mais énoncée sous une forme plus actuelle.

**Proposition 2.10** *Si pour toute composante fortement connexe $C$ du graphe de dépendance de $R$, il existe un ordre de réduction faible $(\succeq, >)$ tel que*

- *$l \succeq r$ pour tout règle $l \to r$ de $R$ ;*
- *$u \succeq v$ pour chaque paire de $C$*
- *$u > v$ pour au moins une paire de $C$*

*alors il n'existe aucune chaîne de dépendance infinie de $R$, et donc $R$ termine.*

Ce critère raffiné du graphe de dépendance n'est pas mécanisable en l'état car le calcul du graphe est indécidable : pour un système de règles arbitraire $R$, on ne peut pas décider l'existence d'un substitution $\sigma$ telle que

$$v_1\sigma \xrightarrow[R]{\neq\Lambda}{}^{*} u_2\sigma$$

**Définition 2.11** *Pour tout terme $t$, on note $cap_R(t)$ le terme obtenu en remplaçant par une nouvelle variable chaque sous-terme de $t$ qui est soit une variable, soit un terme dont le symbole de tête est défini dans $R$.*

**Lemme 2.12** *Si*

$$v_1\sigma \xrightarrow[R]{\neq\Lambda}{}^{*} u_2\sigma$$

*alors $cap_R(v_1)$ et $u_2$ sont unifiables.*

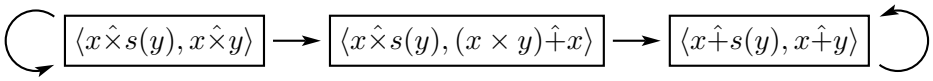
**Définition 2.13** *On appelle graphe de dépendance estimé (EDG) de $R$ le graphe*
- *les sommets sont les paires de dépendances de $R$ ;*
- *il y a une arête de $\langle u_1, v_1\rangle$ vers $\langle u_2, v_2\rangle$ ssi $\cap_R(v_1)$ et $u_2$ sont unifiables.*

Pour mécaniser le critère raffiné, on remplace le graphe de dépendance par ce graphe approché, qui possède au moins autant d'arêtes.

**Exemple 2.14** *Pour montrer la terminaison du système classique de l'addition et de la multiplication des entiers de Peano :*

$$\begin{array}{rcl} x+0 & \to & x \\ x+s(y) & \to & s(x+y) \\ x\times 0 & \to & 0 \\ x\times s(y) & \to & x+(x\times y) \end{array}$$

*on construit le graphe de dépendance approché suivant :*




*Il y a deux composantes fortement connexes, la paire du milieu étant exclue. Il faut donc trouver deux ordres tels que*

$$\begin{array}{rcl} x\hat{+}s(y) & >_1 & x\hat{+}y \\ x\hat{\times}s(y) & >_2 & x\hat{\times}y \end{array}$$

*et $l \succeq_i r$ pour $i = 1, 2$ et chaque règle $l \to r$.*

### 2.2.4 Symboles associatifs et commutatifs

Lorsque l'on spécifie équationnellement des opérations algébriques, il arrive régulièrement que l'on veuille affirmer la commutativité d'une opération $f$, c'est-à-dire donner l'identité

$$f(x, y) = f(y, x)$$

pour tous $x, y$. C'est le cas pour l'addition et la multiplication des nombres, l'union l'intersection des ensembles et multi-ensembles, la conjonction et la disjonction booléenne, etc.

Malheureusement, la commutativité ne peut pas être utilisée en tant que règle de réécriture, car alors elle induirait immédiatement une non-terminaison. Ce problème est généralement contourné en considérant la réécriture *modulo* commutativité, c'est-à-dire où l'équation de commutativité est rendue implicite, en travaillant sur les classes d'équivalence de termes modulo commutativité. Ainsi, si $+$ est un symbole commutatif et avec la règle

$$a + b \to c$$

on se permet de récrire également $b + a$ en $c$. De manière générale, cela nécessite de remplacer le filtrage standard par le filtrage modulo commutativité.

Une autre identité classique est l'associativité :

$$f(f(x, y), z) = f(x, f(y, z))$$

or si l'on oriente cette identité en une règle, par exemple de gauche à droite, alors cela induit une non-terminaison modulo commutativité :

$$f(f(a, b), c) \to f(a, f(b, c)) = f(f(b, c), a) \to \cdots$$

et donc classiquement, en présence à la fois de commutativité et d'associativité, on travaille modulo ces deux identités, avec un filtrage approprié, le filtrage AC [78, 133, 124].

Le critère standard de terminaison de Manna et Ness s'étend très facilement au cas AC : il suffit d'utiliser un ordre de réduction compatible avec AC. Depuis 1986, Ben Cherifa et Lescanne [16] ont donné un critère nécessaire et suffisant pour qu'un ordre défini par interprétations polynomiales soit compatible avec AC. Par contre, obtenir un ordre compatible AC à partir d'un RPO est particulièrement ardu, et ce problème a généré de nombreux travaux [13, 16, 130, 11, 127]. Pour contourner ce problème une autre piste a été explorée : les méthodes par transformations, pour ramener la terminaison d'un système à la terminaison d'un autre [67, 15, 12, 55]

L'extension du critère des paires de dépendance au cas AC n'est pas aussi immédiate. Avec Xavier Urbain, j'ai proposé une première version en 1998 [106], puis Kusakari et Toyama [92], Kapur et Giesl [66], et enfin de nouveau Kusakari, Urbain et moi [90] ont proposé d'autres variantes.

L'expérience acquise, par la pratique de la preuve automatique de terminaison sur de nombreux exemples, a montré que le critère le plus utile était finalement le dénominateur commun des critères précédents, consistant simplement à ne pas utiliser de marques pour les symboles AC. On trouvera une discussion détaillée sur ce point dans [107].

**Définition 2.15** *Les paires de dépendances AC (dans le cas simple) sont obtenues en calculant au préalable, pour chaque règle de réécriture dont le symbole de tête est AC, la règle dite étendue : l'extension de $f(l_1, l_2) \to r$ si $f$ est AC est la règle $f(x, l_1, l_2) \to f(x, r)$ pour une variable fraîche $x$. Les paires de dépendances AC sont alors les paires de dépendances standard calculées sur ce système étendues. Dans le cas des paires marquées, on ne marque pas les symboles AC.*

$$
\begin{array}{rcl}
(\#)0 & \to & \# \\
\# + x & \to & x \\
x + \# & \to & x \\
(x)0 + (y)0 & \to & (x + y)0 \\
(x)0 + (y)1 & \to & (x + y)1 \\
(x)0 + (y)j & \to & (x + y)j \\
(x)1 + (y)1 & \to & (x + y + (\#)1)j \\
(x)j + (y)j & \to & (x + y + (\#)j)1 \\
(x)1 + (y)j & \to & (x + y)0 \\
opp(\#) & \to & \# \\
opp((x)0) & \to & (opp(x))0 \\
opp((x)1) & \to & (opp(x))j \\
opp((x)j) & \to & (opp(x))1 \\
x - y & \to & x + opp(y) \\
\# * x & \to & \# \\
(x)0 * y & \to & (x * y)0 \\
(x)1 * y & \to & (x * y)0 + y \\
(x)j * y & \to & (x * y)0 - y
\end{array}
$$

FIG. 2.1 – Addition et multiplication des entiers relatifs codés en ternaire

*Les chaînes de dépendances sont définies de la même manière que dans le cas standard, mais avec réécriture modulo AC : pour toutes paires consécutives $\langle u_i, v_i\rangle$ et $\langle u_{i+1}, v_{i+1}\rangle$,*

$$v_i\sigma(\xrightarrow[R/AC]{\neq\Lambda})^* u_{i+1}\sigma$$

**Proposition 2.16 (Marché, Urbain, 1998)** *$R$ termine modulo AC ssi il n'existe pas de chaînes de dépendance AC (marquées ou non) infinies.*

Le raffinement du graphe de dépendance s'étend alors simplement au cas AC, en utilisant l'unification AC pour calculer le graphe de dépendance approché, comme nous l'avons montré dès la première variante que nous avons proposée en 1998 [106]. L'automatisation de ces critères se fait en considérant les mêmes contraintes que dans les propositions 2.7 et 2.10, mais avec cette fois un ordre compatible avec AC.

**Exemple 2.17** *En 1997, Evelyne Contejean, Landy Rabehasaina et moi[33] avons proposé le système de réécriture donné sur la figure 2.1 pour calculer l'addition et la multiplication des entiers relatifs codés en ternaire.*

*Dans ce codage [83], les nombres sont engendrés par la constante $\#$ qui représente $0$ ; et les trois constructeurs unaires $0$, $1$ et $j$, notés de façon postfixe, où $(x)0$ représente $3x$, $(x)1$ représente $3x+1$ et $(x)j$ représente $3x-1$. Ainsi 6 est codé par $\#1j0$ ($3\times(3\times(3\times 0+1)-1)$) et $-13$ par $\#jjj$ ($3\times(3\times(3\times 0-1)-1)-1$). Le codage d'un nombre est unique si l'on supprime les zéros inutiles au début, d'où la règle $(\#)0\to\#$.*

*Pour ce système, les symboles $+$ et $*$ sont supposés AC. Sa terminaison était montré à la main, très laborieusement [33], et il était mentionné comme challenge de produire une preuve automatique de sa terminaison. À l'heure actuelle, avec le critère de graphe de dépendance AC tel qu'il est codé dans CiME (et aussi avec la recherche automatique d'interprétations polynomiales détaillée dans la section suivante), il suffit de quelques millisecondes pour établir automatiquement sa terminaison ! Voilà ainsi une manière très concrète et très satisfaisante de mesurer les avancées récentes sur la preuve automatique de terminaison.*

## 2.3 Terminaison par interprétations polynomiales

L'objet de cette section est de présenter mes travaux sur l'utilisation des interprétations polynomiales dans la recherche automatique de preuve de terminaison de systèmes de réécriture. L'essentiel de cette section est paru dans un article écrit avec Evelyne Contejean, Ana Paula Tomás et Xavier Urbain [34], paru dans le numéro spécial sur la terminaison de la revue *Journal of Automated Reasoning* en 2005.

Au début de ces travaux vers 1997, plusieurs croyances erronées circulaient à propos de cette classe d'ordre. Par exemple, certains croyaient que les constantes devaient être obligatoirement interprétées par des entiers au moins égaux à 2, alors que ce n'est que la méthode de test de positivité de BenCherifa-Lescanne qui l'exige.

### 2.3.1 Ordres définis par interprétation et leurs propriétés

On se donne un ensemble non vide $D$ arbitraire, muni d'un ordre $\geq_D$, que l'on appelera *domaine d'interprétation*. On définit $>_D$ comme la relation $\geq_D \setminus \leq_D$.

**Définition 2.18** *Soit $\phi$ une application qui à chaque terme clos $t\in T(F)$ associe un élément de $D$. L'ordre sur $T(F)$ engendré par $\phi$ est défini par*

$$t_1 \geq_\phi t_2 \quad \textit{ssi} \quad \phi(t_1)\geq_D \phi(t_2)$$
$$t_1 >_\phi t_2 \quad \textit{ssi} \quad \phi(t_1)>_D \phi(t_2)$$

**Proposition 2.19** *Si $\geq_D$ est bien-fondé alors $(\geq_\phi, >_\phi)$ est un ordre bien-fondé.*

Une généralisation naturelle de cette construction aux termes non clos est de dire $t_1\geq_\phi t_2$ lorsque $t_1\sigma\geq_\phi t_2\sigma$ pour toute substitution close $\sigma$, c'est en fait la plus grande extension stable de l'ordre sur les termes clos. Néanmoins, une telle définition n'esst pas adaptée à la mécanisation, ne serait-ce que parce qu'elle n'est pas décidable en général. On préfère procéder d'une manière différente qui conduira à une relation plus petite (dans le sens où $t_1\geq_\phi t_2$ impliquera $t_1\sigma\geq_\phi t_2\sigma$ pour tout $\sigma$, mais pas l'inverse).

L'idée est en fait assez naturelle : plutôt que d'interpréter un terme non-clos $t$ en un élément de $D$, ce qui n'a pas vraiment de sens, on donne une fonction qui à chaque interprétation possible dans $D$ des variables de $t$ associe un élement de $D$. Autrement dit, $\phi(t)$ est une application de $X \to D$ dans $D$. L'ensemble $(X \to D) \to D$ des telles applications est naturellement équipé d'ordres définis par :

$$\begin{array}{lll} f \geq_{D,X} g & \text{ssi} & \text{pour tout } I \in X \to D,\ f(I) \geq_D g(I) \\ f >_{D,X} g & \text{ssi} & \text{pour tout } I \in X \to D,\ f(I) >_D g(I) \end{array}$$

Il faut bien remarquer ici que $>_{D,X}$ n'est pas identique à $\geq_{D,X} \setminus \leq_{D,X}$, et c'est pourquoi on a besoin d'utiliser des paires d'ordres. On voit ainsi que l'on a reporté le problème de décider $t_1\sigma \geq_\phi t_2\sigma$ pour tout $\sigma$ en le problème de décider $\geq_{D,X}$ et $>_{D,X}$, ce qui pourra se faire de manière au cas par cas suivant $D$. Dans la section 2.3.2 on verra comment automatiser ces ordres quand $D$ est $\mathbb{N}$ ou plus généralement l'ensemble des entiers relatifs au moins égal à une valeur donnée.

**Définition 2.20** *Soit $\phi$ une fonction qui à $t \in T(F,X)$ associe une fonction de $X \to D$ dans $D$. L'ordre sur $T(F,X)$ engendré par $\phi$ est défini par :*

$$\begin{array}{lll} t_1 \geq_\phi t_2 & \textit{ssi} & \phi(t_1) \geq_{D,X} \phi(t_2) \\ t_1 >_\phi t_2 & \textit{ssi} & \phi(t_1) >_{D,X} \phi(t_2) \end{array}$$

**Proposition 2.21** *Si $>_D$ est bien fondé alors $(\geq_\phi, >_\phi)$ est bien fondé.*

**Définition 2.22** *On définit une interprétation* homomorphique, *ou plus simplement un* morphisme, *en donnant pour chaque $f$ d'arité $n$ une fonction $[\![f]\!]_\phi$ de $D^n$ dans $D$, et alors par récurrence structurelle, pour tout $I \in X \to D$ :*

$$\begin{array}{rcl} \phi(f(t_1,\ldots,t_n))(I) & = & [\![f]\!]_\phi(\phi(t_1)(I),\ldots,\phi(t_n)(I)) \\ \phi(x)(I) & = & I(x) \end{array}$$

**Proposition 2.23** *Si $\phi$ est un morphisme alors $\geq_\phi$ et $>_\phi$ sont stables.*

Avec ce résultat, on sait maintenant que la relation d'ordre sur les termes non-clos est effectivement incluse dans la plus grande extension stable, mais la réciproque est fausse en général, comme le montre l'exemple 2.2.

**Définition 2.24** *Pour un symbole $f \in F$ d'arité $n \geq 1$, on dit que $[\![f]\!]$ est* faiblement *(resp.* strictement*)* croissant en son $i$-ème argument, $1 \leq i \leq n$, *si pour tout $a, b, c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n$ dans $D$, $a \geq_D b$ implique*

$$[\![f]\!](c_1,\ldots,c_{i-1},a,c_{i+1},\ldots,c_n) \geq_D [\![f]\!](c_1,\ldots,c_{i-1},b,c_{i+1},\ldots,c_n)$$

*(resp. $a >_D b$ implique*

$$[\![f]\!](c_1,\ldots,c_{i-1},a,c_{i+1},\ldots,c_n) >_D [\![f]\!](c_1,\ldots,c_{i-1},b,c_{i+1},\ldots,c_n))$$

**Proposition 2.25** *Si pour tout symbole $f$, $[\![f]\!]$ est faiblement (resp. strictement) croissant en chacun de ses arguments, alors $\geq_\phi$ (resp. $>_\phi$) est monotone.*

Pour mécaniser les preuves de terminaison par interprétations, il est nécessaire de fixer un domaine d'interprétation particulier : les entiers. Comme cet ensemble n'est pas bien ordonné, on considère en fait l'ensemble des entiers au moins égaux à une valeur donnée $\mu$.

**Définition 2.26** *Pour un $\mu \in \mathbb{Z}$ donné, soit $D_\mu$ le domaine $\{x \in \mathbb{Z} \mid x \geq \mu\}$ muni de l'ordre usuel, qui est clairement bien-fondé. Les interprétations dans $D_\mu$ sont appelées* $\mu$-arithmétiques, *ou simplement arithmétique quand la valeur de $\mu$ est claire dans le contexte.*

*Une interprétation arithmétique homomorphique est appelée* polynomiale *si pour tout $f \in F$, $[\![f]\!]$ est une fonction polynomiale.*

Ayant fixé ainsi un domaine d'interprétation réduit, il maintenant temps de se poser la question de la mécanisation, pour commencer de la vérification qu'une interprétation donnée est bonne pour prouver une terminaison puis dans un second temps la mécanisation de la recherche d'interprétation.

Pour résumer, pour vérifier qu'un ordre donné (ou un ensemble d'ordres dans le cas des critères de graphes de dépendance) défini par interprétation polynomiale est suffisant pour prouver la termination d'un système de règles, il faut être capable :

1. de vérifier que chacun des polynômes envoit effectivement $D_\mu^n$ dans $D_\mu$ ;
2. de vérifier que ces polynômes sont strictement (ou faiblement pour le critère des paires de dépendance) croissants en leurs arguments ;
3. étant donné deux termes $t_1$ et $t_2$, de vérifier que $t_1 \geq_\phi t_2$ ou $t_1 >_\phi t_2$.

Le point (1) peut être traité ainsi : étant donné un polynôme $P$ à $n$ variables, $P$ envoie $D_\mu^n$ dans $D_\mu$ si et seulement si le polynomial $P - \mu$ est toujours positif ou nul sur $D_\mu^n$.

Le point (2) peut être traité ainsi : étant donné un polynôme $P$ à $n$ variables, $P$ is faiblement croissant en sa $i$-ème variable si et seulement si le polynôme

$$P(X_1, \ldots, X_{i-1}, X_i + 1, X_{i+1}, \ldots, X_n) - P(X_1, \ldots, X_{i-1}, X_i, X_{i+1}, \ldots, X_n)$$

est toujours positif ou nul sur $D_\mu^n$ ; et $P$ est strictement croissant en sa $i$-ème variable si et seulement si le polynôme

$$P(X_1, \ldots, X_{i-1}, X_i + 1, X_{i+1}, \ldots, X_n) - P(X_1, \ldots, X_{i-1}, X_i, X_{i+1}, \ldots, X_n) - 1$$

est toujours positif ou nul sur $D_\mu^n$.

Le point (3) peut être traité ainsi : étant donnés $t_1$ et $t_2$, $\phi(t_1)$ et $\phi(t_2)$ peuvent être calculés formellement sous forme de polynômes $P_1$ et $P_2$ sur les variables de $t_1$ et $t_2$, et alors $t_1 \geq_\phi t_2$ si et seulement si le polynômes $P_1 - P_2$ is positif ou nul sur $D_\mu^n$, et $t_1 >_\phi t_2$ si et seulement si le polynôme $P_1 - P_2 - 1$ est positif ou nul sur $D_\mu^n$.

En conclusion, vérifier la terminaison d'un système de règles avec une $\mu$-interprétation polynomiale donnée peut être mécanisée à partir du moment où l'on sait mécaniser la vérification qu'un polynôme donné est positif ou nul sur $D_\mu^n$.

**Exemple 2.27** *Voici l'un des exemples de Ben Cherifa et Lescanne [16]. Il s'agit d'un système de règles valides pour un morphisme de monoïde.*

$$\begin{array}{rcl}(x \times y) \times z & \to & x \times (y \times z) \\ f(x) \times f(y) & \to & f(x \times y) \\ f(x) \times (f(y) \times z) & \to & f(x \times y) \times z\end{array}$$

*Pour prouver la termination de ce système, avec le critère standard, on affirme que l'ordre polynomial défini par* $\mu = 1$, $[\![f]\!](x) = 2x$ *et* $[\![\times]\!](x, y) = xy + x$ *est adéquat. Voici alors comment en effectuer la vérification :*

1. *D'abord,* $[\![f]\!]$ *envoie* $D_1$ *into* $D_1$*, car*

$$[\![f]\!](x) - 1 = 2x - 1$$

*est positif ou nul quand* $x \geq 1$*. De même, pour* $[\![\times]\!]$*, on vérifie que*

$$[\![\times]\!](x, y) - 1 = xy + x - 1$$

*est positif ou nul quand* $x, y \geq 1$*.*

2. *Puis, pour vérifier que* $[\![f]\!]$ *est strictement croissante, on vérifie que*

$$\begin{array}{rcl}[\![f]\!](x + 1) - [\![f]\!](x) - 1 & = & 2(x + 1) - 2x - 1 \\ & = & 1\end{array}$$

*est positif ou nul quand* $x \geq 1$*. Pour vérifier que* $[\![\times]\!]$ *est strictement croissante en sa première variable, on vérifie que*

$$\begin{array}{rcl}[\![f\times]\!](x + 1, y) - [\![f]\!](x, y) - 1 & = & ((x + 1)y + (x + 1)) - (xy + x) - 1 \\ & = & xy + y + x + 1 - xy - x - 1 \\ & = & y\end{array}$$

*est positif ou nul quand* $x, y \geq 1$*. Enfin, pour vérifier que* $[\![\times]\!]$ *est strictement croissante en sa deuxième variable, on vérifie que*

$$\begin{array}{rcl}[\![f\times]\!](x, y + 1) - [\![f]\!](x, y) - 1 & = & (x(y + 1) + x) - (xy + x) - 1 \\ & = & xy + x + x - xy - x - 1 \\ & = & x - 1\end{array}$$

*est positif ou nul quand* $x, y \geq 1$*.*

3. *Enfin, on doit vérifier que pour chaque règle, le membre gauche est supérieur au membre droit. Pour la première règle on a*

$$\begin{array}{rcl}[\![(x \times y) \times z]\!] & = & (xy + x)z + (xy + x) \\ [\![x \times (y \times z)]\!] & = & x(yz + y) + x\end{array}$$

*donc*

$$\begin{aligned}
&[\![(x \times y) \times z]\!] - [\![x \times (y \times z)]\!] - 1 \\
&\quad = \quad [(xy + x)z + (xy + x)] - [x(yz + y) + x] - 1 \\
&\quad = \quad xyz + xz + xy + x - xyz - xy - x - 1 \\
&\quad = \quad xz - 1
\end{aligned}$$

*qui est positif ou nul quand* $x, y, z \geq 1$. *Pour la deuxième règle, on a*

$$\begin{aligned}
[\![f(x) \times f(y)]\!] &= (2x)(2y) + 2x \\
[\![f(x \times y)]\!] &= 2(xy + x)
\end{aligned}$$

*donc*

$$\begin{aligned}
[\![f(x) \times f(y)]\!] - [\![f(x \times y)]\!] - 1 &= [4xy + 2x] - [2(xy + x)] - 1 \\
&= 2xy - 1
\end{aligned}$$

*qui est positif ou nul quand* $x, y \geq 1$. *Pour la troisième règle on a*

$$\begin{aligned}
[\![f(x) \times (f(y) \times z)]\!] &= 2x(2yz + 2y) + 2x \\
[\![f(x \times y) \times z]\!] &= 2(xy + x)z + 2(xy + x)
\end{aligned}$$

*donc*

$$\begin{aligned}
&[\![f(x) \times (f(y) \times z)]\!] - [\![f(x \times y) \times z]\!] - 1 \\
&\quad = \quad [2x(2yz + 2y) + 2x] - [2(xy + x)z + 2(xy + x)] - 1 \\
&\quad = \quad 4xyz + 4xy + 2x - 2xyz - 2xz - 2xy - 2x - 1 \\
&\quad = \quad 2xyz + 2xy - 2xz - 1 \\
&\quad = \quad 2xz(y - 1) + (2xy - 1)
\end{aligned}$$

*est positif ou nul quand* $x, y, z \geq 1$.

Notons que, comme on le voit sur la dernière vérification de cet exemple, vérifier qu'un polynôme est positif ou nul sur $D_\mu$ peut être non trivial. C'est l'objet de la section suivante.

### 2.3.2 Test de positivité des polynômes

La conclusion de la section précédente est que les conditions à vérifier sur les interprétations peuvent être ramenées à tester si certains polynômes sont positifs ou nuls pour chaque valeurs dans $D_\mu$ de leurs indéterminées.

On se concentre donc sur le *problème de positivité* des fonctions polynomiales :

| | |
|---|---|
| Entrées : | un entier $\mu$ et un polynôme à coefficients entiers $P \in \mathbb{Z}[X_1, \ldots, X_n]$ |
| Sortie : | a-t-on $P(x_1, \ldots, x_n) \geq 0$ pour tous $x_i \geq \mu$ ? |

Il faut remarquer en premier lieu que ce problème est indécidable : le 10eme problème de Hilbert peut s'y réduire. En pratique, on recherche des algorithmes de semi-décision de ce problème, on n'a pas besoin de la complétude.

Les algorithmes déjà mentionnés dans la section 2.1 [17, 65, 131] proposent d'approximer le problème en testant la positivité pour toutes les valeurs réelles supérieures ou égales à $\mu$ de leurs indéterminées, problème qui devient décidable (Tarski 1930) mais néanmoins d'une grande complexité algorithmique. Comme la complétude est déjà abandonnée en passant aux réels, on continue à utiliser sur les polynômes réels des méthodes partielles de test de positivité. En 1998, Hoon Hong et Dalibor Jakus [73] ont établi des comparaisons entre ces méthodes, et proposé une nouvelle. La conclusion de leur travail est que toutes ces méthodes sont aussi puissantes ou moins puissantes (c.-à-d. n'établissent pas la positivité de plus de polynômes) que la méthode triviale suivante.

**Définition 2.28** *Un polynôme $P$ est dit* $\mu$-absolument positif *si si le polynôme*

$$Q(X_1, \ldots, X_n) = P(X_1 + \mu, \ldots, X_n + \mu)$$

*n'a aucun coefficient négatif.*

**Lemme 2.29** *Si $P$ is $\mu$-absolument positif, alors il est positif ou nul sur toutes les valeurs possibles de ses variables dans $D_\mu$.*

*Preuve.* Si $P$ a $n$ indéterminées, soit $k_1, \ldots, k_n$ des entiers arbitraires au moins égaux à $\mu$, alors

$$P(k_1, \ldots, k_n) = Q(k_1 - \mu, \ldots, k_n - \mu)$$

est positif ou nul puisque c'est une expression sans aucune soustraction sur des entiers positifs ou nuls.

Ce test de positivité est vraiment simple à implanter. Néanmoins, il ne faut pas oublier que la normalisation du polynôme translaté $Q$ est potentiellement coûteuse.

**Exemple 2.30** *(suite) Dans l'example 2.27, la dernière vérification demandait de vérifier que*

$$P(x, y, z) = 2xyz + 2xy - 2xz - 1$$

*est positif ou nul pour tout $x, y, z \geq 1$. On calcule*

$$\begin{aligned} P(x+1, y+1, z+1) &= 2(x+1)(y+1)(z+1) + 2(x+1)(y+1) \\ &\quad + 2(x+1)(y+1) - 2(x+1)(z+1) - 1 \\ &= 2xyz + 2xy + 2yz + 2xz + 2x + 2y + 2z + 2 \\ &\quad + 2xy + 2x + 2y + 2 - 2xz - 2x - 2z - 2 - 1 \\ &= 2xyz + 4xy + 2yz + 2x + 4y + 1 \end{aligned}$$

*qui est un polynôme sans coefficient négatif, donc $P$ est* 1*-absolument positif.*

Plus précisément, Hong et Jakus ont montré que la complexité algorithmique de ce test est la même que celle de la méthode de Giesl qui calcule les dérivées successives. Quoi qu'il en soit, puisque cette méthode est au moins aussi puissante que les précédentes, sa simplicité en fait la meilleure méthode.

Mais ce n'est pas tout : grâce à cette méthode, nous avons apporté deux nouveautés, pas difficiles mais très utiles en pratique. Premièrement, nous proposons une façon de contourner la complexité de calcul de la translation, en effectuant plutôt un changement d'interprétations, pour se ramener à $\mu = 0$, le test de positivité devenant alors complètement trivial. Deuxièmement, le fait d'avoir montré que l'on pouvait toujours se ramener à $\mu = 0$ va grandement simplifier la recherche automatique d'interprétations polynomiales adéquates pour la terminaison d'un système de réécriture donné.

**Proposition 2.31** *Soit $(\geq_\phi, >_\phi)$ un ordre défini par des $\mu$-interprétations $[\![f]\!]$. Alors cet ordre peut être défini aussi par des $0$-interprétations polynomiales suivantes :*

$$[\![f]\!]_0(x_1, \ldots, x_n) = [\![f]\!](x_1 + \mu, \ldots, x_n + \mu) - \mu$$

**Exemple 2.32** *Toujours sur l'exemple de Ben Cherifa et Lescanne, on peut calculer une fois pour toutes les translations des interprétations de $f$ et $\times$ : nous avons les nouvelles interprétations :*

$$\begin{aligned} [\![f]\!]_0(x) &= [\![f]\!](x+1) - 1 = 2(x+1) - 1 = 2x + 1 \\ [\![\times]\!]_0(x, y) &= [\![\times]\!](x+1, y+1) - 1 \\ &= (x+1)(y+1) + (x+1) - 1 \\ &= xy + 2x + y + 1 \end{aligned}$$

Pour finir, on peut encore optimiser les choses en remarquant que dans le cas où $\mu = 0$, on a un test très simple de croissance stricte ou faible des interprétations.

**Proposition 2.33** *Une $0$-interprétation polynomiale $P(x_1, \ldots, x_n)$ à coefficients positifs ou nuls est toujours croissante au sens large en chacune de ses variables, et est strictement croissante en sa $i$-ème variable si et seulement si le coefficient de $x_i$ est strictement positif.*

**Exemple 2.34** *Sur l'exemple du morphisme de monoïde, les nouvelles interprétations pour $\mu = 0$ engendrent un ordre strictement monotone, puisque le coefficient de $x$ dans $[\![f]\!]_0$ est 2 et les coefficients de $x$ et $y$ dans $[\![\times]\!]_0$ sont $2$ and $1$ respectivement.*

### 2.3.3 Recherche de polynômes adéquats

Pour un système de réécriture donné, si l'on veut non pas tester si une interprétation polynomiale donnée convient pour sa terminaison, mais si on veut rechercher une telle interprétation, il nous faut procéder de la façon suivante :

1. Introduire des interprétations *paramétriques* pour les symboles, c'est-à-dire où les coefficients sont des variables à déterminer ; pour cela il nous faut choisir une borne sur le degré des polynômes cherchés, comme proposé par Steinbach [132].

2. Un critère de terminaison étant choisi, énoncer les contraintes à satisfaire, sont la forme de condition de positivité de polynômes paramétriques, puis grâce à notre choix systématique de $\mu = 0$, des contraintes arithmétiques (en géneral non linéaires sur les variables cherchées.
3. Résoudre ces contraintes arithmétiques

Ce qui reste à faire est donc la résolution de contraintes égalitaires et inégalitaires sur des entiers, problème qui est encore indécidable.

Un choix possible, qui n'est malheureusement pas toujours satisfaisant en pratique, est de fixer une borne sur les coefficients cherchés, autrement dit chercher une solution sur le domaine $[0, B]$ pour un entier $B$ fixé a priori. On se retrouve alors avec une contrainte sur un domaine fini, et notre contribution a été de mettre en œuvre les méthodes connues pour ce type de contraintes. Néanmoins, pour traiter des exemples de grande taille, nous avons été amené à concevoir une méthode originale d'abstraction des variables[34].

## 2.4 Critères incrémentaux

Prouver la terminaison d'un système de réécriture de grande taille, dans le sens où il possède un grand nombre de règles, est une tâche particulièrement ardue. En effet, la terminaison n'est pas une propriété *modulaire* dans le sens où l'union de deux systèmes qui terminent ne termine pas forcément, même sous des hypothèses très fortes comme par exemple s'ils ne partagent aucun symbole. Cela est mis en évidence par un fameux contre-exemple dû à Yoshihito Toyama [137] : si l'on prend comme premier système la seule règle $f(a, b, x) \to f(x, x, x)$, et comme second système les deux règles de projections $g(x, y) \to x$ et $g(x, y) \to y$, leur union ne termine pas :

$$f(g(a, b), g(a, b), g(a, b)) \to f(a, b, g(a, a)) \to f(g(a, b), g(a, b), g(a, b)) \to \cdots$$

alors que chacun d'eux termine.

Afin d'obtenir des résultats positifs de modularité, deux catégories de restrictions ont été proposées dans la littérature. La première catégorie restreint les règles autorisées selon des critères syntaxiques : linéarité à gauche, règles non-effrondrantes (c'est-à-dire dont le membre droit n'est pas une variable), membres gauches basés sur des constructeurs (c'est-à-dire de la forme $f(t_1, \ldots, t_n)$ où les $t_i$ ne contient que des variables et des constructeurs) ; ou bien sémantiques : confluence locale, etc [45, 54, 86, 111]. En pratique, ces restrictions sont très fortes, c'est pourquoi nous n'avons pas voulu suivre cette voie.

La seconde catégorie de restriction considère des formes renforcées de la terminaison. Le résultat fondateur de cette approche est celui de Kurihara et Ohuchi [87] en 1990, qui montre que la terminaison *simple* est modulaire pour l'union disjointe de systèmes de réécriture. La terminaison simple d'un système $R$ est définie comme la terminaison du système $R \cup \Pi_R$ où $\Pi_R$ est l'ensemble des règles de projection

$$f(x_1, \ldots, x_n) \to x_i$$

pour l'ensemble des symboles $f$ de $R$. Une union de systèmes $R_1 \cup R_2$ est dite *disjointe* si $R_1$ et $R_2$ n'ont aucun symbole en commun.

Ce résultat a été étendu aux unions à constructeurs partagés [68, 88], pour des systèmes qui sont à branchement fini (un terme donné n'a jamais une infinité de réductions possibles), ce qui n'est pas une restriction en pratique puisque les systèmes à nombre fini de règles sont à branchement fini. Enfin, ce résultat a encore été étendu aux unions dites *composables* [85, 119] : toute règle dont le symbole principal est partagé apparaît à l'identique dans les deux systèmes.

La notion d'union composable paraît complexe, alors qu'en fait c'est la notion naturelle qui intervient quand on construit un système de manière incrémentale, en ajoutant successivement chaque symbole et les règles qui le définissent. C'est pourquoi avec Xavier Urbain, nous avons introduit la notion de *module de réécriture* et d'extension :

**Définition 2.35** *Si $R_1$ est un système de réécriture sur une signature $F_1$, un module étendant $R_1$ est un couple $(F_2, R_2)$ où $F_2$ est une signature disjointe de $F_1$ et $R_2$ est un ensemble de règles sur $F_1 \cup F_2$ telle que pour chaque règle $l \to r$ de $R_2$, le symbole de tête de $l$ est dans $F_2$. L'union de $R_1$ et $R_2$ est alors un nouveau système de réécriture sur $F_1 \cup F_2$ appelé l'extension de $(F_1, R_1)$ par $(F_2, R_2)$.*

*On dit que deux modules $(F_2, R_2)$ et $(F_3, R_3)$ étendent $(F_1, R_1)$* indépendemment *si $F_2 \cap F_3 = \emptyset$.*

On relie alors facilement cette notion naturelle à la notion d'union composable :

**Proposition 2.36** *Si $(F_2, R_2)$ et $(F_3, R_3)$ sont deux modules étendant indépendemment $(F_1, R_1)$, alors l'union de $R_1 \cup R_2$ et $R_1 \cup R_3$ est une union de composables, et réciproquement toute union de composables peut-être vue comme l'extension de leur partie commune par deux modules indépendants.*

C'est pourquoi à partir de maintenant nous appelons *union hiérarchique* une union de composables.

Ces résultats positifs sur la terminaison simple sont évidemment importants, néanmoins la terminaison simple est une notion forte de terminaison, et en pratique il est fréquent de rencontrer des systèmes qui terminent mais pas simplement. Ceci arrive même quasiment systématiquement quand on utilise une récursion non structurelle, comme par exemple

$$s(x)/s(y) \to (x - y)/s(y)$$

où le terme $x - y$ est sensé être plus petit que $s(x)$, ce qui contredit la règle de projection vers $y$. On obtient ainsi une séquence infinie contredisant la terminaison simple :

$$s(0)/s(0) \to (0 - s(0))/s(0) \to_\Pi s(0)/s(0) \to \cdots$$

De plus, la technique des paires de dépendance permet de prouver la terminaison de systèmes qui ne terminent pas simplement. Le besoin se fait donc sentir de considérer une notion de terminaison moins forte que la terminaison simple.

Une telle notion a été proposée par Bernhard Gramlich en 1991 [68] : la *terminaison sous effrondement non-déterministe*, qui a été ensuite appelée, en anglais, *Collapse-Extended termination* par Enno Ohlebusch [118]. Nous utilisons l'abréviation termination CE. La terminaison CE d'un système $R$ est définie comme la terminaison du système

$R \cup \pi$ où $\pi$ est le système

$$\begin{aligned} \pi(x,y) &\rightarrow x \\ \pi(x,y) &\rightarrow y \end{aligned}$$

où $\pi$ est un symbole n'apparaissant pas dans $R$. Ainsi, le système $f(a,b,x) \rightarrow f(x,x,x)$ termine mais ne termine pas CE, par contre on ne connait pas d'autres exemples excepté des variations de celui-ci, aussi dans les cas pratiques, la terminaison CE est satisfaisante. Comme pour la terminaison simple, il a été montré que la terminaison CE est une notion modulaire pour les unions disjointes [68, 119], à constructeurs partagés et branchement fini [68], et pour les unions hiérarchiques [89].

Les preuves de ces résultats sont toutes relativement complexes (plusieurs dizaines de pages avec tous les détails), et ont tous été établis avant l'invention du critère des paires de dépendance. Or Xavier Urbain a remarqué que les idées utilisées dans ces preuves sont proches du critère de dépendance, en particulier l'idée de considérer, si un système est supposé non terminant, un contre-exemple minimal où l'on ne réduit pas de terme dont l'un des sous-termes est non fortement normalisable. Ceci a permis, en 2001, de proposer une nouvelle preuve du résultat de modularité de la terminaison CE pour les unions hiérarchiques [142, 143, 144], à la fois beaucoup plus courte et plus élégante, avec de plus un souci d'en extraire une méthode nouvelle de preuve automatique de terminaison, incrémentale. Cette méthode a été implantée dans CiME, avec un grand succès, ainsi un système de plus de 400 règles, proposé par T. Arts pour encoder des processus communicants (avec l'outil $\mu$-CRL [69]), est prouvé terminant automatiquement par CiME en une fraction de seconde.

Mieux encore, avec Xavier Urbain j'ai pu obtenir pour la première fois des résultats de modularité pour la terminaison AC : les anciennes preuves de Kurihara et Ohuchi étaient tellement complexes que leur extension au cas AC était inenvisageable, mais par contre la preuve simple et élégante a pu s'étendre au cas AC sans difficulté majeure [107]. Nos résultats principaux sont d'une part un premier résultat théorique :

**Proposition 2.37** *La terminaison CE modulo AC est modulaire pour l'union hiérarchique de systèmes à branchement fini.*

et d'autre part un résultat pour la pratique de la preuve automatique :

**Proposition 2.38** *Soit $(F_2, R_2)$ et $(F_3, R_3)$ deux modules étendant indépendemment $(F_1, R_1)$. Si*

1. *$R_1 \cup R_2$ termine CE modulo AC*
2. *il existe un ordre faiblement monotone $\pi$-extensible $(\succeq, \succ)$, compatible avec AC, tel que :*
   - *$R_1 \cup R_3 \subseteq \succeq$,*
   - *$DP(R_3) \subseteq \succ$*

*alors $R_1 \cup R_2 \cup R_3$ termine CE modulo AC.*

$$
R_1 \left\{
\begin{array}{rcl@{\qquad}rcl}
\mathsf{ef}(x)[y]_{\mathtt{t}} & \to & \mathsf{ef}(x[y]_{\mathtt{t}}) & \mathsf{Pe}(x)[y]_{\mathtt{f}} & \to & \mathsf{Pe}(x[y]_{\mathtt{t}}) \\
(f \vee g)[s]_{\mathtt{f}} & \to & (f[s]_{\mathtt{f}}) \vee (g[s]_{\mathtt{f}}) & \neg(f)[s]_{\mathtt{f}} & \to & \neg(f[s]_{\mathtt{f}}) \\
(f \wedge g)[s]_{\mathtt{f}} & \to & (f[s]_{\mathtt{f}}) \wedge (g[s]_{\mathtt{f}}) & x[i]_{\mathtt{t}}d & \to & x \\
(f \Rightarrow g)[s]_{\mathtt{f}} & \to & (f[s]_{\mathtt{f}}) \Rightarrow (g[s]_{\mathtt{f}}) & 1[x \cdot s]_{\mathtt{t}} & \to & x \\
\exists(f)[s]_{\mathtt{f}} & \to & \exists(f[1 \cdot (s \circ \uparrow)]_{\mathtt{f}}) & f[i]_{\mathtt{f}}d & \to & f \\
\forall(f)[s]_{\mathtt{f}} & \to & \forall(f[1 \cdot (s \circ \uparrow)]_{\mathtt{f}}) & (f[s]_{\mathtt{f}})[t]_{\mathtt{f}} & \to & f[s \circ t]_{\mathtt{f}} \\
(x[s]_{\mathtt{t}})[t]_{\mathtt{t}} & \to & x[s \circ t]_{\mathtt{t}} & id \circ s & \to & s \\
\uparrow \circ (x \cdot s) & \to & s & (s \circ t) \circ u & \to & s \circ (t \circ u) \\
(x \cdot s) \circ t & \to & (x[t]_{\mathtt{t}}) \cdot (s \circ t) & s \circ id & \to & s \\
1 \cdot \uparrow & \to & id & (1[s]_{\mathtt{t}}) \cdot (\uparrow \circ s) & \to & s
\end{array}
\right.
$$

FIG. 2.2 – Calcul des séquents modulo, partie 1

Grâce à ce résultat, pour montrer la terminaison de l'ensemble connaissant celle de $R_1 \cup R_2$, il suffit de considérer les paires de dépendances de $R_3$ et les règles de $R_1$ et $R_3$ (et surtout pas celle de $R_2$).

Avant de montrer l'efficacité de ce résultat sur un exemple, signalons que ces idées ont été depuis reprises et étendues par Giesl [136].

L'exemple suivant est dû à Eric Deplagne, qui a proposé un calcul des séquents modulo[40, 41]. Il a donné ce système en conjecturant sa terminaison mais sans pourvoir la prouver. Avec CiME, et le critère de terminaison incrémentale modulo AC, nous avons pu prouver sa terminaison entièrement automatiquement.

Ce système est formé de deux parties. La première, $R_1$, donné sur la figure 2.2, définit des substitutions explicites pour les quantificateurs, et la seconde partie, donné sur la figure 2.3, décrit une congruence sur un calcul de séquent. Les symboles $\{_{\mathtt{s}}\_\}$ and $\{_{\mathtt{f}}\_\}$ représentent respectivement les singletons de séquents et les singletons de formules. Les opérations d'union de ces ensembles sont AC et dénotées respectivement par '$\bullet$' et ','.

Le système $R_1 \cup R_2$ contient 53 règles et deux symboles AC. L'analyse hiérarchique automatique décompose ce système en 19 modules (9 triviaux, c'est-à-dire avec des constructeurs mais pas de règle), et c'est cette décomposition modulaire qui permet le succès de la preuve automatique.

## 2.5 Terminaison pour d'autres paradigmes

À partir de 1999, je me suis intéressé à la preuve de terminaison d'algorithme exprimés dans d'autres formalismes que la réécriture. Le premier formalisme auquel je me suis intéressé est celui des programmes logiques. La raison de cet intérêt est issu de résultats plus anciens dus en particulier à Enno Ohlebusch, qui montrent qu'il est possible d'associer à un programme logique un système de réécriture, tel que la terminaison du système assure la terminaison du programme logique de départ. Le problème était que les systèmes obtenus étaient d'une forme spécifique, qui s'avérait être difficile à traiter par les outils de

$$
R_2 \left\{ \begin{array}{rcl}
a, \bigtriangledown & \rightarrow & a \\
a, a & \rightarrow & a \\
a \bullet \Diamond & \rightarrow & a \\
a \bullet a & \rightarrow & a \\
\neg(\neg(f)) & \rightarrow & f \\
f \wedge f & \rightarrow & f \\
f \vee f & \rightarrow & f \\
f \Rightarrow g & \rightarrow & \neg(f) \vee g \\
\exists(f) & \rightarrow & \neg(\forall(\neg(f))) \\
(a, \{\mathtt{f}\neg(f)\}) \vdash b & \rightarrow & a \vdash (\{\mathtt{f}f\}, b) \\
\{\mathtt{f}\neg(f)\} \vdash b & \rightarrow & \bigtriangledown \vdash (b, \{\mathtt{f}f\}) \\
a \vdash (\{\mathtt{f}\neg(f)\}, b) & \rightarrow & (a, \{\mathtt{f}f\}) \vdash b \\
a \vdash \{\mathtt{f}\neg(f)\} & \rightarrow & (a, \{\mathtt{f}f\}) \vdash \bigtriangledown \\
(a, \{\mathtt{f}f \wedge g\}) \vdash b & \rightarrow & (a, \{\mathtt{f}f\}, \{\mathtt{f}g\}) \vdash b \\
\{\mathtt{f}f \wedge g\} \vdash b & \rightarrow & (\{\mathtt{f}f\}, \{\mathtt{f}g\}) \vdash b \\
a \vdash (\{\mathtt{f}f \vee g\}, b) & \rightarrow & a \vdash (\{\mathtt{f}f\}, \{\mathtt{f}g\}, b) \\
a \vdash \{\mathtt{f}f \vee g\} & \rightarrow & a \vdash (\{\mathtt{f}f\}, \{\mathtt{f}g\}) \\
\{\mathtt{s}a \vdash (\{\mathtt{f}f \wedge g\}, b)\} & \rightarrow & \{\mathtt{s}a \vdash (\{\mathtt{f}f\}, b)\} \bullet \{\mathtt{s}a \vdash (\{\mathtt{f}g\}, b)\} \\
\{\mathtt{s}a \vdash \{\mathtt{f}f \wedge g\}\} & \rightarrow & \{\mathtt{s}a \vdash \{\mathtt{f}f\}\} \bullet \{\mathtt{s}a \vdash \{\mathtt{f}g\}\} \\
\{\mathtt{s}(a, \{\mathtt{f}f \vee g\}) \vdash b\} & \rightarrow & \{\mathtt{s}(a, \{\mathtt{f}f\}) \vdash b\} \bullet \{\mathtt{s}(a, \{\mathtt{f}g\}) \vdash b\} \\
\{\mathtt{s}\{\mathtt{f}f \vee g\} \vdash b\} & \rightarrow & \{\mathtt{s}\{\mathtt{f}f\} \vdash b\} \bullet \{\mathtt{s}\{\mathtt{f}g\} \vdash b\} \\
\{\mathtt{s}(a, \{\mathtt{f}f\}) \vdash (\{\mathtt{f}f\}, b)\} & \rightarrow & \Diamond \\
\{\mathtt{s}(a, \{\mathtt{f}f\}) \vdash \{\mathtt{f}f\}\} & \rightarrow & \Diamond \\
\{\mathtt{s}\{\mathtt{f}f\} \vdash (b, \{\mathtt{f}f\})\} & \rightarrow & \Diamond \\
\{\mathtt{s}\{\mathtt{f}f\} \vdash \{\mathtt{f}f\}\} & \rightarrow & \Diamond \\
\{\mathtt{s}a \vdash b\} \bullet \{\mathtt{s}(a, f) \vdash (g, b)\} & \rightarrow & \{\mathtt{s}a \vdash b\} \\
\{\mathtt{s}a \vdash b\} \bullet \{\mathtt{s}(a, f) \vdash b\} & \rightarrow & \{\mathtt{s}a \vdash b\} \\
\{\mathtt{s}a \vdash b\} \bullet \{\mathtt{s}a \vdash (b, f)\} & \rightarrow & \{\mathtt{s}a \vdash b\} \\
\{\mathtt{s}a \vdash \bigtriangledown\} \bullet \{\mathtt{s}(a, f) \vdash b\} & \rightarrow & \{\mathtt{s}a \vdash \bigtriangledown\} \\
\{\mathtt{s}a \vdash (b, f)\} \bullet \{\mathtt{s}\bigtriangledown \vdash b\} & \rightarrow & \{\mathtt{s}\bigtriangledown \vdash b\} \\
\{\mathtt{s}a \vdash b\} \bullet \{\mathtt{s}\bigtriangledown \vdash b\} & \rightarrow & \{\mathtt{s}\bigtriangledown \vdash b\} \\
\{\mathtt{s}a \vdash b\} \bullet \{\mathtt{s}a \vdash \bigtriangledown\} & \rightarrow & \{\mathtt{s}a \vdash \bigtriangledown\} \\
\{\mathtt{s}a \vdash b\} \bullet \{\mathtt{s}\bigtriangledown \vdash \bigtriangledown\} & \rightarrow & \{\mathtt{s}\bigtriangledown \vdash \bigtriangledown\}
\end{array} \right.
$$

FIG. 2.3 – Calcul des séquents modulo, partie 2

terminaison automatique de l'époque. Or le renouveau apporté par la technique des paires de dépendances, puis le regain d'intérêt pour les ordres par interprétations polynomiales expliqué dans la section 2.3, ont changé la situation : les premiers tests ont montré que la classe des systèmes obtenus par transformation des programmes logiques se traite bien par la combinaison du critère des paires de dépendances et les ordres par interprétations poly-

nomiales. Ce travail a donné lieu à une publication commune avec Ohlebusch et Manuel Claves de l'Université de Bielefeld, Allemagne [122]. Une raison informelle de ce succès est que les systèmes obtenus par cette transformation comportent les symboles définis mutuellement récursivement, et d'autre part avec une arité relativement grande.

Après avoir traité le cas des programmes logiques, je me suis intéressé, en collaboration avec José Meseguer (Université de l'Illinois à Urbana-Champaign, USA), aux programmes dits *équationnels avec contraintes d'appartenance*, formalisme sous-jacent au système Maude [30]. Ce travail s'est poursuivi avec d'autres collègues étrangers : Salvador Lucas (Université de Valence, Espagne) et Francisco Durán (Université de Malaga, Espagne) et a conduit à plusieurs résultats successifs [53, 99].

Le point commun entre les programmes logiques et les programmes Maude est le passage intermédiaire par la réécriture dite *conditionnelle*, où l'application d'une instance de règle est soumise à une condition à vérifier sur cette instance. Quelques critères de terminaison étaient connus pour cette variante de la réécriture, notamment dus à Ohlebusch [121]. Pour traiter en particulier les programmes Maude de la section 2.5.3 qui suit, avec Salvador Lucas et José Meseguer [99] nous avons étendu ces critères au cas de règle avec stratégie d'application dite *context-sensitive* : la réécriture n'est pas permise sous certains contexte, comme par exemple avec

$$\begin{aligned} if(true, x, y) &\rightarrow x \\ if(false, x, y) &\rightarrow y \end{aligned}$$

on interdit de réécrire dans le 2ème et le 3ème argument du $if$, pour modéliser le comportement habituel des programmes.

### 2.5.1 Terminaison de la réécriture conditionnelle

Une règle de réécriture conditionnelle est une règle de la forme

$$l \rightarrow r \mid s_1 = t_1, \ldots, s_n = t_n$$

où les $s_1 \rightarrow t_i$ sont appelées les conditions de la règle. On abrège cette notation en $l \rightarrow r \mid c$ où $c$ est la condition de la règle. Une telle règle est de type :

1. si $Var(r) \cup Var(c) \subset Var(l)$ ;
2. si $Var(r) \subset Var(l)$ ;
3. si $Var(r) \subset Var(l) \cup Var(c)$ ;
4. sinon

Un $n$-CTRS est un ensemble de règles de type inférieur ou égal à $n$. L'interprétation des conditions peut se faire de plusieurs façons, nous considèrons des CTRS dits *orientés*, où l'égalité des conditions est interprétée comme la réduction elle-même. On note alors les règles sous la forme

$$l \rightarrow r \mid s_1 \rightarrow t_1, \ldots, s_n \rightarrow t_n$$

(Reflexivité) $$\frac{}{t \to^* t}$$

(Transitivité) $$\frac{t \to^1 t' \qquad t' \to^* t''}{t \to^* t''}$$

(Congruence) $$\frac{u_i \to^1 u_i'}{f(u_1, \ldots, u_i, \ldots, u_n) \to^1 f(u_1, \ldots, u_i', \ldots, u_n)}$$
où $f \in \mathcal{F}$ et $1 \leq i \leq ar(f)$

(Remplacement) $$\frac{s_1\sigma \to^* t_1\sigma \quad \ldots \quad s_n\sigma \to^* t_n}{t\sigma \to^1 t'\sigma}$$
où $\quad t \to t' \mid s_1 \to t_1 \cdots s_n \to t_n \in R$

FIG. 2.4 – Règles d'inférence pour la réécriture conditionnelle

Un 3-CTRS est dit déterministe si

$$\text{Var}(s_i) \subseteq \text{Var}(l) \cup \bigcup_{j=1}^{i-1} \text{Var}(t_j)$$

Cette condition assure que l'application d'une règle n'introduit jamais de variables non instanciées, ce qui correspond à la façon naturelle de calculer. Cette forme de règle sera la forme la plus générale que l'on autorisera en pratique.

La relation de réduction par un CTRS orienté $R$ est définie inductivement comme l'union des relations $R_n$ définies par $R_0 = \emptyset$ et

$$R_{n+1} = \{l\sigma \to r\sigma \mid (l \to r \mid s_1 \to t_1, \ldots, s_n \to t_n) \in R_n \text{ et } s_i\sigma \to^*_{R_n} t_i\sigma\}$$

La difficulté principale vient de cette définition qui n'est pas opérationnelle : pour décider si un terme se réécrit en un pas, il faut tester les conditions de la règle, ce qui demande de tester si les $s_i$ se réduisent en un nombre quelconque de pas en les $t_i$. Un interpréteur de CTRS pourra donc se mettre à boucler sans appliquer une seule règle, mais simplement en cherchant à tester les conditions. C'est le cas par exemple pour la règle

$$a \to b \mid a \to c$$

Pour pallier ce problème, nous avons introduit une notion différente de terminaison : la termination *opérationnelle*. L'idée est de définir la relation de réduction de manière axiomatique, par les règles de la figure 2.4.

La terminaison opérationnelle est alors définie comme le fait que l'on ne peut pas engendrer d'arbres infinis de dérivations.

$$(\mu\text{-Congruence}) \quad \frac{u_i \to^1 u_i'}{f(u_1,\ldots,u_i,\ldots,u_n) \to^1 f(u_1,\ldots,u_i',\ldots,u_n)} \quad \text{où } f \in \mathcal{F},\ 1 \le i \le ar(f) \text{ et } i \in \mu(f)$$

FIG. 2.5 – Règle de congruence pour la réécriture context-sensitive

**Terminaison par transformation**

Ohlebusch a proposé une transformation, qui construit un système de réécriture inconditionnel à partir d'un système conditionnel : chaque règle conditionnelle $l \to r \mid c$ avec $n$ conditions dans $c$ est transformée en $n+1$ règles inconditionnelles par l'opérateur $U$ défini par induction sur $n$ par :

$$\begin{aligned} U(l \to r) &= \{l \to r\} \\ U(l \to r \mid s \to t, c) &= \{l \to u(s,\vec{x})\} \cup U(u(t,\vec{x}) \to r \mid c) \end{aligned}$$

où $u$ est un nouveau symbole et $\vec{x} = \text{Var}(l) \cap (\text{Var}(t) \cup \text{Var}(c) \cup \text{Var}(r))$

La propriété essentielle de cette transformation est la suivante.

**Proposition 2.39 (Ohlebusch [120])** *Pour tout CTRS déterministe $R$, si $U(R)$ termine alors $R$ termine.*

En fait, Ohlebusch prouve plus précisément que le CTRS $R$ est *quasi-decreasing*, une propriété qu'il n'est pas utile de définir ici, mais qui est plus forte que la terminaison. Nous avons montré [99] que la terminaison opérationnelle d'un CTRS était en fait équivalente à la propriété de quasi-decreasingness. Cela semble dire que l'on n'a rien gagné, mais en fait l'avantage de l'approche axiomatique est que l'on peut facilement changer de règles d'inférence pour définir la relation de réduction. C'est ainsi que nous avons considéré les stratégies context-sensitive : à chaque symbole $f$ de la signature on associe un ensemble $\mu(f)$ d'entiers entre 1 et l'arité de $f$, qui indique quelles sont les positions où l'on a le droit de réécrire sous $f$. Ainsi, sur l'exemple du $if$ précédent, on posera $\mu(if) = \{1\}$, pour indiquer que l'on ne peut réduire que le premier argument. La sémantique est précisément définie en changeant la règle de congruence de la figure 2.4 par la règle de la figure 2.5.

En accompagnement de cette définition, on introduit une variante de la transformation de Ohlebusch : à un $\mu$-CTRS $R$ on associe un $\mu$-TRS $U_\mu(R)$ de la même façon que pour la transformation de Ohlebusch, en conservant les mêmes $\mu(f)$ pour les symboles $f$ déjà présents dans $R$, et en ajoutant $\mu(u) = \{1\}$ pour les symboles introduits $u$. Nous avons alors obtenu le résultat suivant :

**Proposition 2.40 (Duran, Lucas, Marché, Meseguer, Urbain [53])** *Pour tout $\mu$-CTRS déterministe $R$, si $U_\mu(R)$ termine alors $R$ termine opérationnellement.*

### 2.5.2 Terminaison des programmes logiques

La terminaison d'un programme logique doit toujours être considérée selon un *mode* donné de ce programme.

**Définition 2.41** *Un* mode *pour un programme logique est la donnée, pour chaque prédicat $p$ de ce programme, d'arité $n$, d'une fonction de $\{1,\ldots,n\}$ dans $\{in, out\}$ (paramètres d'entrée, resp. de sortie). Pour simplifier, on note $p(\vec{s},\vec{t})$ pour signifier que les $\vec{s}$ sont les entrées et $\vec{t}$ les sorties.*

*Un programme logique est* bien modé *pour un mode donné si pour chaque clause*

$$p(t_0, s_{n+1}) \texttt{ :- } p_1(s_1,t_1),\ldots,p_n(s_n,t_n)$$

*on a pour tout $i \in [1, n+1]$*

$$\mathit{Var}(s_i) \subseteq \bigcup_{j=0}^{i-1} \mathit{Var}(t_j)$$

Autrement dit, chaque variable apparaissant en entrée d'un prédicat du corps d'une clause apparait comme variable d'entrée de la tête de clause ou de sortie d'un atome venant avant dans le corps.

L'exemple ultra-classique de programme logique est celui de la concaténation de listes :

$$\begin{array}{rcl} app([],X,X). & & \\ app([X|Y],U,[X|Z]) & \texttt{ :- } & app(Y,U,Z). \end{array}$$

Pour ce programme, les modes (i,i,o), (i,i,i) mais aussi (o,o,i) sont acceptables. Le mode (i,o,o) ne l'est pas.

**Définition 2.42** *On dit qu'un programme logique bien modé termine si pour toute requête bien modée, l'exécution de Prolog sur une telle requête s'arrête en temps fini.*

Notre but est maintenant de déterminer un critère de terminaison pour ces programmes bien modés. On procède en transformant le programme logique bien modé en un CTRS orienté déterministe de la façon suivante (idée de transformation originalement due à Ganzinger [64]). À chaque atome $A = p(\vec{s},\vec{t})$ on associe la règle de réécriture

$$\rho(A) = p_{in}(\vec{s}) \to p_{out}(\vec{t})$$

À chaque clause

$$C = A \texttt{ :- } B_1,\ldots,B_m$$

on associe la règle conditionnelle

$$\rho(C) = \rho(A) \mid \rho(B_1),\ldots,\rho(B_m)$$

**Proposition 2.43 (Ohlebusch [121])** *Si le programme de départ est bien modé, alors le système de réécriture conditionnel obtenu est déterministe. Si le système obtenu termine, alors le programme logique de départ termine.*

Sur l'exemple de la concaténation des listes avec le mode (i,i,o), on obtient :

$$\begin{array}{rcl} app_{in}(nil, X) & \rightarrow & app_{out}(X) \\ app_{in}(cons(X,Y),U) & \rightarrow & app_{out}(cons(X,Z)) \mid app_{in}(Y,U) \rightarrow app_{out}(Z) \end{array}$$

Le TRS obtenu par transformation U est alors

$$\begin{array}{rcl} app_{in}(nil, X) & \rightarrow & app_{out}(X) \\ app_{in}(cons(X,Y),U) & \rightarrow & u(app_{in}(Y,U),X) \\ u(app_{out}(Z),X) & \rightarrow & app_{out}(cons(X,Z)) \end{array}$$

dont la preuve de terminaison est aisée.

Avec Ohlebusch et Claves, j'ai mis en œuvre ce procédé dans un prototype appelé TALP [122], qui effectue la transformation et ensuite passe le TRS obtenu à un prouveur de terminaison comme CiME. Avec CiME comme prouveur final, nous avons pu noter que la combinaison du critère des paires de dépendances et des ordres par interprétations polynomiales était particulièrement efficace. Cela est probablement dû au fait que les transformations produisent des symboles avec beaucoup d'arguments, mais avec des termes peu profond. La technique obtenue a été reconnue comme très efficace par rapport à des techniques purement adaptée à Prolog, en particulier parce que ces dernières demandent à l'utilisateur d'indiquer des mesures (des *normes* dans le vocabulaire des prologuistes) manuellement, alors qu'avec TALP et CiME les preuves se font automatiquement : c'est le résolveur de contraintes polynomiales qui s'occupe de « trouver » les normes adéquates.

### 2.5.3 Terminaison des programmes équationnels avec contraintes d'appartenance

Le système Maude [30, 109] est un environnement de spécification, de programmation et de preuve développé conjointement par le SRI et l'Université de l'Illinois à Urbana-Champaign. Le formalisme dans lequel ont écrit les spécifications et les programmes est celui des programmes équationnels avec contraintes d'appartenance. Si l'on fait abstraction du sucre syntaxique, on y définit des règles de réécriture de la forme

$$l \rightarrow r \mid c_1, \ldots, c_n$$

et des règles d'appartenance de la forme

$$t : s \mid c_1, \ldots, c_n$$

où $t$ est un terme et $s$ est une *sorte*. L'ensemble des sortes est un ensemble fini de noms introduits par l'utilisateur.

Dans les deux formes de règles, les conditions $c_i$ sont elles-mêmes de la forme $s_i \rightarrow t_i$ ou bien $t_i : s_i$. Ce formalisme permet d'encoder un typage dynamique, avec sous-sortes et surcharge d'opérateurs. Voici un exemple représentatif de programme, sur les listes infinies paresseuses :

```
fmod OvConsOS is
   sorts Nat NatList NatIList
   subsort NatList < NatIList
          (finite lists: subset of lazy lists)
   op 0 : -> Nat
   op s : Nat -> Nat
   op zeros : -> NatIList      zeros is [0,0,0,...]
   op nil : -> NatList
   op cons : Nat NatIList -> NatIList [strat (1 0)]
           (no reduction on 2nd argument)
   op cons : Nat NatList -> NatList [strat (1 0)]
           (overloading)
   op take : Nat NatIList -> NatList
           (take(n,[x_1, x_2, ...]) = [x_1, ..., x_n])
   op length : NatList -> Nat
   vars M N : Nat
   var  IL : NatIList
   var  L : NatList
   eq zeros = cons(0,zeros)
   eq take(0, IL) = nil
   eq take(s(M), cons(N, IL)) = cons(N, take(M, IL))
   eq length(nil) = 0
   eq length(cons(N, L)) = s(length(L))
endfm
```

La déclaration de sous-sorte (les listes finies forment une sous-sorte des listes quelconques) est formellement vu comme la règle d'appartenance

$$l : NatIList \mid l : NatList$$

Les déclarations de profil de nil et cons sont vues comme

$$\begin{array}{rcl} nil & : & NatList \\ cons(x,y) & : & NatIlist \mid x : Nat, y : NatIList \\ cons(x,y) & : & Natlist \mid x : Nat, y : NatList \end{array}$$

et donc la surcharge revient simplement à donner deux règles d'appartenance pour $cons(x, y)$.

La sémantique d'un programme avec appartenance est donnée par les règles de la figure 2.6. Elles définissent simultanément les relations $\rightarrow^1$ de reécriture en un pas, $\rightarrow^*$ de réécriture en un nombre quelconque de pas, :: relation d'appartenance directe et : relation d'appartenance indirecte. La distinction entre les deux relations d'appartenances se fait lors de l'interprétation des conditions d'appartenance : si $c_i$ est une condition d'appartenance $t_i : s_i$, l'interprétation de $c_i\sigma$ dans les règles (Remplacement) et (Appartenance) est $t\sigma : s$ en général, sauf si $t$ est réduit à une variable auquel cas c'est $t\sigma :: s$. La raison de ce choix est d'empecher d'appliquer la subject-reduction sur des variables.

$$\text{(Remplacement)} \qquad \frac{c_1\sigma \quad \dots \quad c_n\sigma}{t\sigma \to^1 t'\sigma} \quad \text{où} \quad t \to t' \mid c_1 \cdots c_n \in R$$

$$\text{(Appartenance)} \qquad \frac{c_1\sigma \quad \dots \quad c_n\sigma}{t\sigma :: s} \quad \text{où} \quad t : s \mid c_1 \cdots c_n \in R$$

$$\text{(Appartenance2)} \qquad \frac{t :: s}{t : s}$$

$$\text{(Subject reduction)} \qquad \frac{t \to^1 u \qquad u : s}{t : s}$$

FIG. 2.6 – Règles d'inférence pour la réécriture avec appartenance

Notre problématique est donc : comment peut-on vérifier la terminaison opérationnelle d'un programme avec appartenance ? Nous avons proposé [53] une transformation qui à partir d'un programme avec appartenance produit un $\mu$-CTRS, telle que si le $\mu$-CTRS obtenu termine opérationnellement, alors le programme original aussi.

Ce résultat a été implanté dans un outil MTT (`TODO`) satellite de Maude (les transformations sont en fait codées en Maude !) et qui fait appel aux outils externes MU-TERM [98] et CiME [32]. Les résultats expérimentaux ne sont pour l'instant pas très satisfaisant car il y a des exemples naturels dont la terminaison n'arrive pas a être établie. Nous sommes encore au stade de l'étude de ces échecs, afin de comprendre où se trouvent les limitations.

## 2.6 Conclusions et perspectives

Les avancées obtenues à partir de 1997 sur les techniques de preuve automatique de terminaison ont permis des progrès considérables pour les outils implantant ces techniques. Ainsi, de nombreux outils de ce type ont vu le jour durant cette période.

Cette prolifération d'outils m'a incité a organiser une compétition d'outils de terminaison. La première édition a eu lieu en mars 2004, et la deuxième édition, co-organisée avec Hans Zantema, a eu lieu en mars 2005. Les résultats de ces compétitions peuvent être consulté sur le Web : les transparents qui présentent les résultats sont aux adresses `http://www.lri.fr/~marche/termination-competition/2004/slides-1jun2004.ps` et `http://www.lri.fr/~marche/termination-competition/2005/TC.ppt`.
Les résultats complets sont aux adresses `http://www.lri.fr/~marche/termination-competition/2004/` et `http://www.lri.fr/~marche/termination-competition/2005/`.

La prochaine compétition est prévu pour avril 2006. Les résultats seront présentés lors du workshop sur la terminaison, affilié à la fédération de conférences Floc'2006 à Seattle

en août 2006, en même temps que d'autres compétitions d'outils automatiques de preuve (CASC, SMT).

Cette compétition joue maintenant un rôle moteur pour motiver l'exploration de nouvelles techniques, les problèmes résistants aux outils devenant de fait des défis. Ainsi, un des problèmes de la base de données est devenu l'un des problèmes ouverts de la liste des problèmes ouverts en réécriture(`http://www.lsv.ens-cachan.fr/rtaloop/` [48], problème numéro 104), et est maintenant résolu par Hofbauer and Waldmann (résultat non encore publié).

Il est remarquable que les avancées obtenues depuis 1997 ont permis pour la première fois de prouver par ordinateur la terminaison de systèmes que l'on ne savait pas prouver à la main, comme le montrent les exemples de la section 2.4

Mes contributions sur la preuve de terminaison pour d'autres formalismes de calcul que la réécriture [122, 53, 99] montrent que les méthodes par traduction sont viables, même s'il reste encore clairement du travail dans cette direction.

Dans le chapitre 4, je présenterai des perspectives de travaux de recherche sur la terminaison.

# Chapitre 3

# Preuve de programmes impératifs

Ce chapitre aborde le second de mes deux principaux axes de recherche : la preuve mécanisée de propriétés fonctionnelles de programmes impératifs.

## 3.1 Contexte

### 3.1.1 Historique

L'article de Alan Turing de 1949, mentionné dans l'introduction ainsi qu'au début du chapitre sur la terminaison [140], n'a pas pour objet essentiel la preuve de terminaison mais surtout la preuve qu'une certaine « routine » (c'est le vocabulaire de cet article) calcule bien ce que l'on attend. Le principe utilisé dans cet article est fondateur, puisqu'il s'agit d'annoter chaque point du programme par une formule logique qui exprime une propriété satisfaite par les valeurs des variables en ce point. On reconnaitra clairement le principe qui sera repris ensuite par Floyd en 1967 [62] et Hoare en 1969 [71].

La logique de Floyd-Hoare est une axiomatique portant sur des jugements d'une forme spéciale (dite *triplet de Hoare*) $\{P\}\ i\ \{Q\}$ qui est une assertion portant sur une instruction $i$ et des formules logiques $P$ et $Q$ portant sur les variables de $i$, affirmant que dans un état où les variables vérifient $P$, si l'on exécute $i$ alors on obtient un état où les variables vérifient $Q$. La formule $P$ est alors appelée *pré-condition* de $i$ et $Q$ est appelée sa *post-condition*. Malgré son ancienneté et la simplicité de son principe, la logique de Hoare est encore aujourd'hui à la base de nombreuses méthodes de preuves sur des programmes impératifs, grâce aux extensions qui lui ont été faites [5, 37].

La logique de Hoare est bien sûr adaptée à la preuve de programmes impératifs, mais pas du tout à la preuve de programmes fonctionnels. Pour les programmes fonctionnels, le formalisme qui se montre particulièrement adapté est la théorie des types [108, 76, 35, 77] en particulier les formalismes de lamba-calcul typé d'ordre supérieur avec types dépendants, car ceux-ci permettent de formuler la spécification d'un programme dans son type :

la formule

$$\forall X, P(X) \rightarrow \exists Y, Q(X, Y)$$

exprime le type d'un programme qui a pour données d'entrée $X$ vérifiant la pré-condition $P$, ayant $Y$ comme données de sortie vérifiant la post-condition $Q$. Avec un langage de types d'une telle expressivité, la preuve qu'un programme vérifie une telle spécification est équivalent à prouver qu'il a le type ci-dessus : c'est le fameux isomorphisme de Curry-Howard, qui identifie les types et les spécifications d'une part, et les termes et les programmes d'autre part.

Le système Coq, basé sur une théorie des types appelée *Calcul des constructions inductives* [112], se montre ainsi bien adapté à la preuve de programmes fonctionnels. Il existe même un mécanisme d'extraction [97], qui à partir d'une preuve d'une formule du type ci-dessus extrait un programme Caml ou Haskell, correct par construction. Afin de traiter en Coq des programmes de nature impérative, une méthode basée sur une interprétation fonctionnelle des programmes impératifs a été proposée en 1999 par Jean-Christophe Filliâtre [56]. Depuis, cette approche a donné lieu à un nouvel outil appelé WHY [58], qui permet de prouver des programmes impératifs écrits dans un langage spécifiquement adapté à la preuve de programme, qui génère des obligations de preuve (formules dont la validité entraine la correction du programme) et qui autorise d'utiliser aussi bien Coq que d'autres prouveurs, aussi bien des prouveurs interactifs (PVS, Isabelle/HOL) que automatiques (Simplify, etc.).

### 3.1.2 Situation en 2001

En 2001, je me suis intégré au projet européen IST VerifiCard, dont l'une des tâches était la preuve de programmes Java en utilisant Coq. L'objectif était d'utiliser la technologie WHY comme outil, et donc d'utiliser le langage de WHY comme langage intermédiaire pour la preuve de programmes Java. Puis à partir de 2003, nous avons également travaillé sur les programmes C.

### 3.1.3 Résultats obtenus

Les points suivants ont été abordés :

- Principe d'utilisation de l'approche Why pour des programmes avec structures de données complexes (tableaux, pointeurs, objets...) ;
- Modèles de la mémoire adaptés à la preuve de programme.
- Preuve de propriétés fonctionnelles de programmes directement sur le code source Java ou C
- Études de cas

Dans la suite de ce chapitre, nous allons décrire en détail la méthodologie WHY (section 3.2), puis comment on l'utilise pour des langages réels en général (section 3.3), puis spécifiquement sur les programmes JAVA (section 3.4) et C (section 3.5), enfin nous présentons des études de cas (section 3.6). Les problèmes ouverts et les enjeux futurs seront présentés dans la section 3.7 ainsi que dans le chapitre de conclusions et perspectives.

## 3.2 La méthode Why

### 3.2.1 Présentation de la méthode Why

La méthode Why, proposée par Jean-Christophe Filliâtre [57], peut être vue comme établissant un pont entre l'approche logique de Hoare pour les programmes impératifs et l'approche théorie des types/isomorphisme de Curry-Howard pour les programmes fonctionnels. L'idée de base de la méthode Why est issue directement du résultat bien connu de l'équivalence d'expressivité entre les formalismes impératifs et fonctionnels. Un programme impératif travaillant sur des variables $v_1, \ldots, v_n$, utilisant les valeurs initiales d'une partie $R$ (comme *reads*) de ces variables, et affectant une partie $W$ (comme *writes*) de ces variables, peut être codé par un programme fonctionnel ayant $R$ comme données d'entrée et $W$ comme données de sortie. Bien sûr, en général $R$ et $W$ ne sont pas disjoints, et une variable $v$ à la fois dans $R$ et dans $W$ sera à la fois une entrée $v_{in}$ et une sortie $v_{out}$ du programme fonctionnel équivalent.

Remarque technique : en fait pour effectuer cette traduction fonctionnelle, il faut prendre comme données d'entrée du programme fonctionnel toutes les variables $R \cup W$ : considérons par exemple le programme

```
if x > 0 then y := x + 1 else z := z + 2
```

on a $R = \{x, z\}$ et $W = \{y, z\}$, et la traduction fonctionnelle est

```
fun (x_in, y_in, z_in) ->
   if x_in > 0 then (x_in + 1, z_in) else (y_in, z_in + 2)
```

où l'on voit que l'on ne pourrait pas retourner les valeurs de sortie du couple $(y_{out}, z_{out})$ dans le cas où $x_{in} \leq 0$ si l'on avait pas $y_{in}$ comme donnée d'entrée. Ce phénomène peut se comprendre d'une autre manière : les programmes fonctionnels n'acceptent jamais de « variable » dont la « valeur initiale » n'est pas encore définie, et donc seuls les programmes impératifs ayant toutes leurs variables initialisées peuvent être traduits en programmes fonctionnels. Dans la suite, l'ensemble $R$ des variables accédées contiendra donc toujours $W$.

La méthode WHY se propose alors de faire des preuves de programmes impératifs non pas avec des règles de déduction à la Hoare, mais en prouvant ces propriétés sur leurs programmes fonctionnels équivalents avec une méthode basée sur la théorie des types. Cela demande bien sûr d'étendre la traduction de programmes à la traduction de programmes annotés, c'est-à-dire de traduire les triplets de Hoare, et c'est là que l'utilisation d'un langage comme le calcul des constructions, qui utilise le même langage pour exprimer les termes du calcul et les assertions logiques, prend tout son sens : un triplet $\{P\}\ i\ \{Q\}$ avec les variables accédées $R$ et les variables affectées $W$ sera traduit en un programme dont le type est

$$\forall R, P(R) \rightarrow \exists W, Q(R, W)$$

c'est-à-dire un programme fonctionnel ayant comme arguments les variables de $R$ et une preuve de $P(R)$, qui retourne les valeurs $W$ des variables affectées et une preuve de $Q(R, W)$. Les techniques pour construire de tels programmes avec preuves est bien sûr le cœur même de l'approche WHY, et est décrite dans [56, 57].

### 3.2.2 Les conséquences d'un point de vue « utilisateur»

En résumé, la méthode WHY est une méthode de preuves de programmes impératifs, qui utilise une approche radicalement différente de la logique de Hoare et de ses nombreuses extensions, car elle procède par traduction vers un programme fonctionnel et utilise les méthodes pour ceux-ci. Cela a des conséquences visibles dans l'utilisation de cette méthode.

Une première conséquence concerne la terminaison des programmes. La logique de Hoare classique ne fait que de la correction dite partielle, elle ne prouve jamais la terminaison, par exemple des boucles *while*. Au contraire, la traduction fonctionnelle d'une boucle *while* est une fonction récursive, et les méthodes de preuves sur des programmes fonctionnels peuvent et même parfois requièrent une preuve de terminaison de cette récursion. Ainsi, là ou une boucle est généralement annotée par un invariant en logique de Hoare, avec Why une boucle sera et même devra aussi être annotée par un *variant*, une mesure qui décroît lors de chaque exécution du corps de la boucle.

Une deuxième conséquence concerne l'utilisation de variables dites *auxiliaires* ou *fantômes* (*ghost* en anglais), pour pouvoir faire référence aux valeurs des variables prises à des « anciens » points de programme. Avec la méthode WHY, du fait de la traduction fonctionnelle qui est derrière, qui représente les valeurs successives d'une variable du programme impératif par une séquence de noms affectés de façon unique, les variables auxiliaires peuvent être avantageusement remplacées par une syntaxe $v@label$ désignant la valeur de $v$ au point de programme référencé par $label$. En particulier les constructions qui permettent, dans un triplet de Hoare $\{P\}\ i\ \{Q\}$ de faire référence dans $Q$ aux valeurs des variables avant l'exécution de $i$, n'ont pas besoin d'être interprétées de manière ad-hoc mais simplement traduite par la référence au bon nom dans la traduction fonctionnelle. C'est pourquoi les constructions `\old` et `\varAtLabel` de JML que l'on verra en section 3.4.1 seront traitées de manière très simple et naturelle avec l'approche WHY, alors qu'elles nécessitent des interprétations complexes avec une approche classique à la Hoare.

D'une manière générale, un certain nombre d'extension de la logique de Hoare qui ont été proposées sont traitées directement par la méthode WHY :

- Les effets de bord dans les expressions.
- L'appel de sous-programmes.
- Les exceptions.

Par contre, les limitations communes entre l'approche Why et l'approche classique à la Hoare sont

- On a toujours un ensemble fini fixé de variables, et il n'y pas *d'alias* possible entre elles. Ceci est à la fois une limitation et une hypothèse nécessaire à la correction même des approches. Par exemple la pseudo procédure PASCAL

```
incr2(inout x:int,inout y:int);
 begin
   x := x + 1; y := y + 1
 end
 { x = \old(x) + 1 and y = \old(y) + 1 }
```

  a une post-condition qui n'est pas valide si l'on s'autorise à l'appeler avec la même variable pour $x$ et $y$.

– Les entiers des programmes sont des entiers relatifs mathématiques, et donc le non-débordement des opérations sur les entiers doit être traité autrement.
– Il n'y a pas de structures de données complexes, là encore cela doit être traité autrement.

Ces trois limitations devront donc être levées si l'on souhaite traiter des programmes écrits dans des langages comme C ou Java, avec des entiers bornés, des structures des données complexes à base d'enregistrements, de tableaux, de pointeurs, d'objets ; qui plus est pouvant avoir du partage donc des alias de pointeurs.

### 3.2.3 Utilisation concrète de WHY

La levée des limitations précédentes peut être effectuée non pas en inventant des extensions à la méthode Why, mais en utilisant de manière appropriée la possibilité de travailler sur des types abstraits, dont la réalisation est définie dans la logique. En effet, WHY a une approche modulaire, dans le sens où de même qu'il accepte des programmes en paramètre avec uniquement leurs spécifications, il accepte des types abstraits, et des prédicats et fonctions logiques abstraites sur ces types. Autrement dit, les types de données complexes doivent être modélisés.

L'utilisation de WHY pour traiter des programmes impératifs dans des langages comme C ou JAVA se décompose alors en quatre étapes.

1. Déterminer une représentation des structures de données complexes, c'est-à-dire dans le cas de C et JAVA, du tas mémoire, par un ensemble fini $V$ de variables WHY.
2. Définir un calcul qui pour chaque instruction du langage source détermine les variables $V$ qui sont accédées et celles qui sont affectées. Cette étape est indispensable pour traduire les spécifications des sous-programmes et spécifications Why.
3. Construire un schéma de traduction des constructions du langage source annoté en des programmes WHY annotés.
4. Produire une modélisation logique des types abstraits représentant les structures de données, pour servir d'axiomatique pour les preuves des obligations de preuves.

### 3.2.4 Exemple d'un programme avec un tableau : le drapeau hollandais

C'est un exemple que j'ai conçu pour le cours de DEA Programmation, Sémantique Preuves et Langages en 2003-2004. Il n'a été publié nulle part ailleurs. Il s'agit d'un exemple célèbre : un programme de tri linéaire d'un tableau quand celui-ci ne contient que trois valeurs possibles (comme les couleurs du drapeau hollandais !), dû à Dijkstra [50]. Nous montrons maintenant comment spécifier et coder cet algorithme en WHY, puis comment on le prouve.

La première étape consiste à introduire la notion de couleur. Pour cela, on introduit un type abstrait `color`, muni de trois constantes :

```
parameter BLUE, WHITE, RED : color
```

puis on souhaite spécifier que ces trois couleurs sont les seules possibles, ce qu'on exprime en logique du premier ordre par un prédicat muni d'un axiome approprié :

```
logic iscolor : color -> prop
axiom color_elim :
  forall c:color. iscolor(c) -> c=BLUE or c=WHITE or c=RED
```

Il s'agit là d'une façon de coder en logique du premier ordre un type de donnée fini.

Le langage logique de WHY inclut un prédicat d'égalité, aussi sans rien déclarer de plus, nous avons le droit d'exprimer l'égalité de deux couleurs. Par contre si l'on souhaite, dans le corps du programme, faire un test d'égalité, il nous faut *déclarer* la fonction associée, ce qui se fait de la manière suivante :

```
parameter eq_color : c1:color -> c2:color ->
  { }  bool  { if result then c1=c2 else c1<>c2 }
```

Cette clause signifie que l'on déclare un nouveau sous-programme `eq_color`, qui prend en argument deux couleurs $c_1$ et $c_2$, qui n'a pas de pré-condition (`{ }`), retourne un booléen, et la valeur retournée (`result`) vérifie la post-condition `if result then c1=c2 else c1<>c2` signifiant que le booléen calculé représente bien l'égalité au sens logique de $c_1$ et $c_2$, autrement dit ce sous-programme est bien un test d'égalité. Il s'agit là d'un programme WHY déclaré mais non réalisé, donc cette déclaration est de même nature qu'un axiome : on suppose que l'on est capable de tester l'égalité de deux couleurs (en particulier cela implique la décidabilité de l'égalité des couleurs).

L'étape suivante consiste à introduire une axiomatisation des tableaux. Noter que l'outil WHY propose dans sa « bibliothèque standard » une telle axiomatisation, mais pour notre propos nous définissons une autre variante. Pour traiter les tableaux en logique de Hoare, la méthode classique consiste à les représenter dans la logique comme des tableaux *fonctionnels*, c.-à-d. où la mise à jour d'une case de tableau retourne un nouveau tableau. Nous reprenons cette même idée pour notre modélisation en WHY, et donc nous introduisons un type abstrait `colorarray` muni des opérations logiques suivantes :

```
(* longueur d'un tableau *)
logic length : colorarray -> int

(* acc(t,i) represente t[i] *)
logic acc : colorarray, int -> color

(* mise à jour t[i] := c *)
logic update : colorarray, int, color -> colorarray
```

Ces déclarations doivent être accompagnées d'axiomes exprimant les propriétés attendues de ces opérations : ceux-ci sont enumérés sur la figure 3.1.

Afin d'utiliser les opérations des tableaux dans les programmes WHY, il nous faut déclarer les fonctions WHY suivantes :

```
(* calcul de la longueur d'un tableau *)
parameter length_ : t:colorarray ->
  { }  int  { result = length(t) }
```

```
(* les longueurs de tableaux sont positives ou nulles *)
axiom length_pos: forall t:colorarray. 0 <= length(t)

(* une mise à jour d'un tableau ne change pas sa longueur *)
axiom length_up:
  forall t:colorarray.
    forall i:int. forall v:color.
       length(update(t,i,v)) = length (t)

(* axiomes spécifiant le résultat de l'accès à une case de
 * tableau après une mise à jour : le premier axiome traite
 * le cas où on accède à la case fraîchement mise à jour et
 * le deuxième axiome traite les autres cas
 *)
axiom acc_up_eq :
  forall t:colorarray.
    forall i:int. forall v:color.
       acc(update(t,i,v),i) = v

axiom acc_up_neq :
  forall t:colorarray.
    forall i:int. forall j:int. forall v:color.
       i <> j -> acc(update(t,i,v),j) = acc(t,j)
```

FIG. 3.1 – Une axiomatique WHY des tableaux fonctionnels

```
(* accès à une case d'un tableau *)
parameter acc_ : t:colorarray -> i:int ->
  { 0 <= i < length(t) } color  { result=acc(t,i) }

(* mise à jour d'une case d'un tableau *)
parameter update_ :
  t:colorarray ref -> i:int -> v:color ->
  { 0 <= i < length(t) }
  unit reads t writes t
  { t = update(t@,i,v) }
```

La fonction WHY `length_` déclare l'existence d'un programme qui calcule la longueur d'un tableau, la fonction `acc_` retourne effectivement une case donnée d'un tableau, sous la précondition que l'indice donné est bien dans les bornes du tableau, et enfin, sous la même précondition, la fonction `update_` modifie une case de tableau par effet de bord : la post-condition exprime que la nouvelle valeur de $t$ est le résultat de la mise à jour fonctionnelle

```
let swap = fun (t : colorarray ref) (i:int) (j:int) ->
  { 0 <= i < length(t) and 0 <= j < length(t) }
  let ti = (acc_ !t i) in
  let tj = (acc_ !t j) in
  begin
  (update_ t i tj);
  (update_ t j ti)
  end
  { length(t) = length(t@) and
    acc(t,i) = acc(t@,j) and
    acc(t,j) = acc(t@,i) and
    forall k:int. i <> k and j <> k ->
                  acc(t,k) = acc(t@,k) }
```

FIG. 3.2 – Code WHY de la fonction `swap`

de l'ancienne valeur (on rappelle que la notation $t@$ dénote la valeur de $t$ avant l'exécution de la fonction.

Les trois programmes WHY déclarés ci-dessus pour les tableaux sont très représentatifs de la démarche que l'on utilise pour traiter les types de données complexes, c'est là leur intérêt.

Nous sommes maintenant prêts à procéder à la spécification et la vérification de programmes WHY travaillant sur des tableaux. On commence par écrire un petit programme qui échange deux éléments d'un tableau. Celui-ci est donné sur la figure 3.2. La pré-condition exprime simplement que les deux indices $i$ et $j$ sont bien dans les bornes du tableau. La post-condition exprime quatre points :

1. la longueur du tableau en paramètre est inchangée ;
2. la nouvelle valeur de t[i] est l'ancienne valeur de t[j]
3. la nouvelle valeur de t[j] est l'ancienne valeur de t[i]
4. toutes les cases d'indice différents de i et j ont une valeur inchangée.

Vérifier que le corps du programme (`let ti ... end`) satisfait cette spécification est très simple.

Pour écrire le programme de tri de Dijkstra, nous introduisons un nouveau prédicat `monochrome` :

```
(* monochrome(t,i,j,c) exprime que tous les éléments du
 * tableau t entre les indices i et  j (i inclus et
 * j non-inclus) sont de la couleur c
 *)
predicate monochrome(t: colorarray, i:int, j:int, c:color) =
  forall k:int. i <= k < j -> acc(t,k)=c
```

```
let flag = fun (t : colorarray ref) ->
  { forall k:int. 0 <= k < length(t) -> iscolor(acc(t,k)) }
  begin
  let b = ref 0 in
  let i = ref 0 in
  let r = ref (length_ !t) in
  while !i < !r do
   { invariant
       (forall k:int. 0 <= k < length(t) ->
                      iscolor(acc(t,k)))
       and 0 <= b and b <= i and i <= r and r <= length(t)
       and monochrome(t, 0, b, BLUE)
       and monochrome(t, b, i, WHITE)
       and monochrome(t, r, length(t), RED)
     variant r - i }
    let c = (acc_ !t !i) in
    if (eq_color c BLUE)
    then
      begin (swap t !b !i); b := !b + 1; i := !i + 1 end
    else
     if (eq_color c WHITE)
     then i := !i + 1
     else
       begin r := !r - 1; (swap t !r !i) end
   done
  end
  { exists r:int. exists b:int.
           monochrome(t, 0, b, BLUE)
       and monochrome(t, b, r, WHITE)
       and monochrome(t, r, length(t), RED) }
```

FIG. 3.3 – Code WHY de la fonction `flag`

Nous pouvons alors coder et spécifier le programme de Dijkstra en WHY : c'est celui donné sur la figure 3.3. La pré-condition exprime que les éléments du tableau `t` sont bien des couleurs. La post-condition exprime que le tableau forme bien un drapeau hollandais : une zone bleue (de 0 à $b-1$), suivie d'une zone blanche (de $b$ à $r-1$), suivie enfin d'une zone rouge (de $r$ à la fin du tableau).

Notons que les onze obligations de preuves engendrées pour ces deux programmes (`swap` et `flag`) sont prouvées entièrement automatiquement par le prouveur Simplify. D'autre part, une preuve manuelle en COQ se fait facilement, avec 116 lignes de tactiques en tout. En conclusion, ce programme classique du drapeau hollandais, représentatif de la

façon dont on peut traiter les programmes sur des structures de données complexes, est un programme facile à prouver formellement avec la technologie WHY.

## 3.3 Programmes Java et C : généralités

L'approche précédente pour les tableaux simples peut être étendue à toute structure de données complexes, ce qui permet de traiter des « vrais » langages comme Java ou C : pour ceux-ci, la structure de la mémoire est modélisée avec les opérations logiques abstraites et des axiomes. L'outil Krakatoa [105, 103] suit ce principe, et permet de traduire automatiquement un programme Java en un programme Why avec une telle modélisation. L'outil CADUCEUS [61, 59] fait de même avec les programmes C.

### 3.3.1 Propriétés typiques recherchées

Il y a deux classes de propriétés qu'il faut chercher à prouver. La première est l'absence d'erreur à l'exécution : pas de division par zéro, absence de déréférenciation de pointeurs non alloués, pas d'accès en dehors des tableaux. Note dans le cas de Java : ces « erreurs » ne provoquent pas l'arrêt du programme mais lèvent des exceptions qui en principe peuvent être capturé par le programme. Or ces cas sont rarissimes, et donc dans notre approche on cherchera vraiment à prouver que les exceptions ne sont jamais levées.

La deuxième classe est celle des post-conditions ajoutées explicitement par le programmeur.

### 3.3.2 Modélisation fine du tas mémoire

Historiquement, la modélisation des structures avec pointeurs, avec alias possibles, est marquée par une approche proposée par Burstall [24] : les champs de structure peuvent être vus chacun comme un tableau indexé par les adresses. Grâce à cette idée, on sait statiquement que si un champ d'une structure est modifié, les autres champs de toutes les structures sont inchangés, et par contre les alias sont parfaitement modélisés. Cette idée sera reprise par de nombreux auteurs, et nous feront de même pour les programmes Java et C. Un article marquant sur ce sujet est celui de Bornat[22] en 2000, où il reprend cette idée pour prouver formellement des programmes manipulant des listes chaînées et des graphes. Dans la suite on parlera de modèle à la Burstall-Bornat.

## 3.4 Preuves de programmes JAVA

### 3.4.1 Programmes JAVA annotés en JML

Le langage JML (*Java Modeling Language*) est un langage pour spécifier le comportement des classes et des interfaces Java. La conception de ce langage est un effort collaboratif, dirigé par l'équipe de Gary Leavens de l'Université d'État de l'Iowa. La page Web de JML `http://www.cs.iastate.edu/~leavens/JML/index.shtml` donne les détails des différents partenaires.

Ce langage est destiné à être utilisé par des outils ayant différents objectifs. L'objectif principal est celui de la vérification des assertions introduites à l'exécution : cette tâche est assurée par l'outil JMLRAC (*runtime assertion checker*).

Plusieurs équipes dans le monde ont souhaité utiliser JML comme langage de spécification pour une approche de preuve de programme : le groupe SOS de Nijmegen (Bart Jacobs, Erik Poll) aux Pays-bas [79] développe l'outil LOOP, qui utilise PVS comme prouveur ; l'outil Jack [23] developpé par GemPlus puis maintenu par le projet Everest de l'INRIA Sophia ; l'outil Jive [110] de Arnd Poetzsch-Heffter à Kaiserlautern ; le système KeY [4] developpé par des équipes de recherche à Karlsruhe et Koblenz en Allemagne ainsi qu'à Gothenburg en Suède.

Nous avons souhaité poursuivre le même but, la preuve de programme Java annotés en JML, avec un outil Krakatoa, qui procède en générant un programme Why équivalent, selon l'approche décrite à la section 3.2.3. Il n'est pas question ici de donner tous les détails, par ailleurs donnés dans les articles publiés [105, 104], mais de donner les points importants.

Les figures 3.4 et 3.5 illustrent l'utilisation de JML sur l'exemple du drapeau Hollandais de la section précédente. Sur cet exemple, les obligations de preuve sont résolues automatiquement avec le prouveur Simplify. La preuve manuelle avec Coq est du même ordre de complexité que la preuve du programme WHY de la section précédente. On peut donc dire que notre interprétation WHY des programmes JAVA n'introduit pas de complexité supplémentaire : c'est là l'avantage du plongement dit *shallow embedding* par rapport au plongement profond *deep embedding* qui consisterait à encoder les programmes comme une structure de données concrète.

### 3.4.2 Modèle WHY des objets JAVA

Notre modèle des objets JAVA [104] utilise la représentation de Burstall-Bornat pour les variables d'instances : chaque champ non statique des classes donne lieu à une variable Why de type `memory`, qui est une table indexée par les adresses. Nous avons étendu ce modèle pour gérer les tableaux. Pour connaître dynamiquement la classe de chaque objet et les tailles de chaque tableau, une table supplémentaire `alloc` est introduite pour associer à chaque adresse la classe de l'objet qui s'y trouve ou bien la taille du tableau qui s'y trouve. La modélisation de la mémoire est schématiquement représentée sur la figure 3.6.

Le point important de ce modèle est que la mémoire est découpée en zones où les alias d'adresses ne sont possibles qu'à l'intérieur d'une zone et pas entre zones.

Les opérations d'accès et de mise à jour de la mémoire peuvent alors être axiomatisées : un ensemble d'axiomes satisfaits par ces opérations est décrit en WHY.

### 3.4.3 Traduction de JAVA/JML vers WHY

L'intérêt d'utiliser WHY comme langage intermédiaire est de bien séparer les problèmes d'interprétations du flot de contrôle de celui des données. La représentation des données a été choisie ci-dessus, et pour le flot de contrôle, nous utilisons la boucle while de WHY pour interpréter les boucles while, for et do de Java, et nous utilisons le mécanisme d'exception de WHY pour interpréter les exceptions Java, mais aussi le break, le continue et le return. Les

```
public class Flag {

    public static final int BLUE = 1, WHITE = 2, RED = 3;

    /*@ private normal_behavior
      @   ensures \result <==>
      @        (i == BLUE || i == WHITE || i == RED);
      @*/
    private static /*@ pure @*/ boolean isColor(int i) {
          return BLUE <= i && i <= RED;
    }

    public int t[];

    /*@ public invariant
      @  t != null && (\forall int k;
      @           0 <= k && k < t.length; isColor(t[k]));
      @*/

    /*@ private normal_behavior
      @   requires 0 <= i && i <= j && j <= t.length ;
      @   ensures \result <==>
      @        (\forall int k; i <= k && k < j; t[k] == c);
      @*/
    private /*@ pure @*/ boolean isMonochrome(int i, int j, int c) {
        /*@ loop_invariant k >= i &&
            (\forall int l; i <= l && l < k ==> t[l]==c);
               decreases j - k;
         */
        for (int k = i; k < j; k++){
            if (t[k] != c) {return false;}
        }
        return true;
    }
```

FIG. 3.4 – Le drapeau Hollandais de Dijkstra en Java, partie 1

```
/*@ private normal_behavior
      @   requires 0 <= i && i < t.length && 0 <= j && j < t.length;
      @   modifiable t[i],t[j];
      @   ensures
      @     t[i] == \old(t[j]) && t[j] == \old(t[i]);
      @*/
    private void swap(int i, int j) {
        int z = t[i];
        t[i] = t[j];
        t[j] = z;
    }

    /*@ public normal_behavior
      @   modifiable t[*];
      @   ensures
      @     (\exists int b,r; isMonochrome(0,b,BLUE) &&
      @                       isMonochrome(b,r,WHITE) &&
      @                       isMonochrome(r,t.length,RED));
      @*/
    public void flag() {
        int b = 0;
        int i = 0;
        int r = t.length;
        /*@ loop_invariant
          @   (\forall int k; 0 <= k && k < t.length; isColor(t[k])) &&
          @   0 <= b && b <= i && i <= r && r <= t.length &&
          @   isMonochrome(0,b,BLUE) &&
          @   isMonochrome(b,i,WHITE) &&
          @   isMonochrome(r,t.length,RED);
          @ decreases r - i;
          @*/
        while (i < r) {
            switch (t[i]) {
            case BLUE: swap(b++, i++); break;
            case WHITE: i++; break;
            case RED: swap(--r, i); break;
            }
        }
    }
}
```

FIG. 3.5 – Le drapeau Hollandais de Dijkstra en Java, partie 2

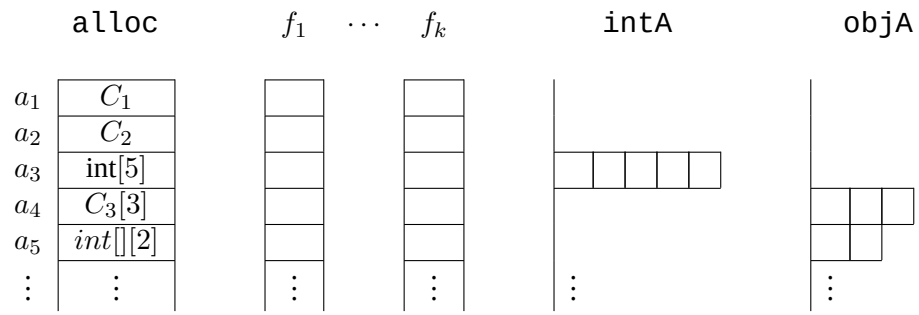



FIG. 3.6 – Modèle à la Burstall-Bornat du tas Java

schémas de traduction sont décrits dans [105]. La jonction entre la traduction du contrôle et celui des données se fait quand on traduit les opérations d'accès et de mise à jour d'un champ d'objet : on traduit $e.f$ en

let tmp = $\tilde{e}$ in { tmp $\neq$ null } (`acc` $\tilde{f}$ tmp) { }

où $\tilde{f}$ est la variable WHY associée à $f$, et $\tilde{e}$ est, récursivement, la traduction de $e$. La fonction WHY `acc` est la fonction d'accès à un champ, très similaire à celle utilisé sur l'exemple du drapeau hollandais. C'est en introduisant la précondition dans le schéma de traduction que l'on force la génération d'une obligation de preuve assurant que l'on ne déréférence pas le pointeur `null`.

### 3.4.4 Correction de la méthode

Une question importante est évidemment celle de la correction de cette approche : comment peut-on garantir l'adéquation de la sémantique de JAVA et celle du programme traduit en WHY ? En fait on ne peut pas garantir cette adéquation, et même de toute façon nous ne prétendons pas supporter toutes les constructions Java. Nous prétendons simplement que sur le fragment considéré, les schémas de traduction sont suffisamment simples pour être compris et admis comme corrects. C'est là même un avantage de la méthode passant par WHY : on peut se concentrer sur l'essentiel, à savoir décrire la sémantique de JAVA dans WHY, et laisser WHY s'occuper du problème de la génération d'obligations de preuve.

Une autre source possible d'incorrection dans notre approche est l'axiomatique utilisée pour décrire les opérations sur la mémoire : nous devons nous assurer que celle-ci est cohérente. Ceci a été fait en en produisant un modèle Coq calculatoire [104], sans axiome.

### 3.4.5 Problèmes des invariants de classe et de l'héritage

Nous concluons cette partie sur les programmes JAVA en mentionnant le point le plus important qui n'est pas actuellement supporté par Krakatoa.

La sémantique des invariants de classes en JML, comme indiqué dans le document de référence est : « à l'appel et au retour d'une méthode, tous les invariants de tous les objets

*accessibles* sont supposés valides ». Cette sémantique est difficile à mettre en œuvre car l'ensemble des objets accessibles n'est pas calculable statiquement : par exemple pour une structure chaînée, un arbre, un graphe, le nombre d'objets accessibles dépend de chaque exécution. Une solution possible consiste à exprimer l'ensemble des objets accessibles sous forme d'une formule, de façon analogue à notre façon d'exprimer les cellules mémoires spécifiées comme inchangées dans une clause *assignable*. Cette solution n'a pas été mise en œuvre par peur d'une trop grande complexité d'utilisation lors des preuves.

De manière très pragmatique, l'implantation actuelle de Krakatoa limite simplement l'ensemble des objets supposés satisfaire leur invariant à l'appel ou au retour d'une méthode : on ne prend en compte que les objets qui sont directement paramètres de la méthode. Ainsi sur l'exemple

```
class A {
  B x;
  static m(A a,B b[]) {
     ...
  }
}
```

Krakatoa considère que l'objet `a` doit satisfaire l'invariant de la classe `A`, mais n'exige rien sur `a.x`, ni les éléments de tableau `b[i]`, qui pourtant selon la sémantique de JML doivent aussi satisfaire l'invariant de la classe `B`.

Ce manque de respect de la sémantique de JML implique une incorrection de Krakatoa vis-à-vis de cette sémantique : ainsi le programme

```
class A {
      int x; // invariant x != 0;
}

class B {
      A i;
      void m() {
          this.i.x = 0;
      }
}
```

est considéré comme correct par Krakatoa, alors qu'il ne l'est pas selon la sémantique JML. Néanmoins il faut remarquer que Krakatoa est correct vis-à-vis de sa propre sémantique qui réclame la validité des invariants uniquement des objets directement en argument de méthode. Mais bien sûr cela pose des problèmes de complétude, ainsi le programme

```
class A {
      int x; // invariant x != 0;
}

class B {
```

```
      int m(A t[]) {
          return 100 / t[0].x;
      }
}
```

ne peut pas être prouvé exempt de division par zéro, puisque les éléments de `t` ne sont pas supposés vérifier leur invariant.

Un problème plus subtil avec les invariants de classe est leur interaction avec l'héritage. Considérons le programme

```
abstract class A {
    /*@ normal_behavior
      @  assignable \nothing;
      @  ensures \result != 0;
      @*/
    abstract int m1();
}

class C {
    static int m2(A a) {
        return 100/a.m1();
    }
}
```

on peut prouver que la méthode `m2` ne peut pas faire de division par zéro, grâce à la post-condition de la méthode `m1`. Mais si l'on étend la classe `A` par

```
class B extends A {
    int x; //@ invariant 0 < x;

    int m1() { return x; }

    int m3() {
        x = x-1;
        int z = C.m2(this);
        x = x+1;
        return z;
    }
}
```

alors, grâce à l'invariant, on peut effectivement prouver que la méthode `m1` respecte son contrat : elle retourne un entier non nul ; mais par contre l'appel à `m2` risque de provoquer une division par zéro. Lorsque l'on a traité la méthode `m2`, on réclame que son paramêtre `a` vérifie l'invariant de la classe `A`, mais pas de la classe `B`. Autrement dit, en présence d'héritage la vérification modulaire n'est plus possible, on a besoin de savoir que les objets vérifient l'invariant de leur classe dynamique, et pas seulement de leur classe statique.

Une solution à ces problèmes avec la sémantique des invariants de classe est proposée par Barnett, DeLine, Fähndrich, Leino et Schulte [14]. Ils proposent un changement radical dans la sémantique des invariants : les objets doivent pouvoir momentanément violer leur invariant, de façon contrôlée par l'ajout d'annotations explicites. Cette souplesse permet de recouvrer une approche correcte dans tous les cas, avec l'inconvénient de réclamer un travail supplémentaire d'annotations de la part de l'utilisateur.

Cette solution est malheureusement incompatible avec la sémantique actuelle de JML. La décision d'adopter cette approche pour Krakatoa, et donc de devenir incompatible avec JML, est encore en cours d'étude.

## 3.5 Programmes C

Dans cette section, nous nous intéressons spécifiquement à ce qui concerne les programmes C. L'aspect central qui n'apparaît pas en Java est l'arithmétique de pointeurs.

Contrairement au cas de Java, les outils de preuve formelle existants qui traitent directement les programmes C sont très rares (on peut citer l'outil non public Caveat du CEA Saclay), et il n'y a pas d'équivalent de JML comme langage commun de spécification. Nous avons dû concevoir notre propre langage, tout en nous inspirant de JML.

Avant d'entrer dans le détail de la modélisation de l'arithmétique de pointeurs, nous allons traiter deux exemples. Ce sont des exemples que j'ai conçu pour illustrer la méthode, ils n'ont été publiés nulle part, mais présentés à la journée QSL (Qualité et Sûreté du Logiciel) à Nancy, le 10 février 2005 (`http://qsl.loria.fr/Externe/10022005.php`)

### 3.5.1 Exemple : memcpy et strcpy

Le premier exemple est celui de la fonction `memcpy` consistant à recopier une zone de mémoire d'une longueur donnée. Le code source annoté de cette fonction est donné sur la figure 3.7. La clause `ensures` est la précondition, elle affirme que les cases 0 à $n-1$ sont allouées pour les tableaux, puis que ceux-ci ont des *adresses de base* différentes. Ceci signifie intuitivement que ce sont des pointeurs sur deux zones mémoires allouées avec deux appels distincts à une fonction de type `malloc`. Ceci sera détaillé à la section 3.5.2. La clause *ensures* est la post-condition, qui exprime que les cases du tableau `dest` sont devenues égales aux case de `src`.

Pour la vérification de la fonction `memcpy`, CADUCEUS produit des obligations de preuve qui expriment que :

- le code de `memcpy` ne contient pas de déréférenciation de pointeur non alloué : en l'occurrence il s'agit de montrer que l'affectation `dest[i]= src[i]` est valide ;
- la post-condition de `memcpy` est établie.

Ces obligations sont prouvées entièrement par le prouveur SIMPLIFY, et en Coq elles sont prouvées en seulement 8 lignes of tactiques, ce qui est très peu. Autrement dit, ce code C se prouve très facilement.

Le deuxième exemple est celui de la fonction `strcpy` pour copier des chaînes de caractères C, qui sont des tableaux de caractères terminés par 0. Un code pour cette fonction

```
typedef int size_t;

/*@ requires
  @   \valid_range(src,0,n-1) &&
  @   \valid_range(dest,0,n-1) &&
  @   \base_addr(src) != \base_addr(dest)
  @ ensures
  @   \forall int k; 0 <= k < n => dest[k] == src[k]
  @*/
char *memcpy(char *dest, const char *src, size_t n)
  int i = n;
  /*@ invariant
    @   i <= n &&
    @   \forall int k; i <= k < n => dest[k] == src[k]
    @ variant i */
  while (i-- > 0) dest[i] = src[i];
  return dest;
```

FIG. 3.7 – Code C annoté de la fonction `memcpy`

particulièrement concis, que l'on trouve par exemple dans le livre référence sur le C de Kernighan et Ritchie [82], est :

```
void strcpy(char *dest, const char *src) {
  while (*dest++ = *src++);
}
```

Les difficultés posées par ce programme, par rapport à celui de `memcpy`, sont :
- l'utilisation de l'arithmétique de pointeur ;
- l'absence de borne explicite pour la boucle, qui est nécessaire pour assurer aussi bien la terminaison de la fonction que l'absence de déréférenciation de pointeur invalide.

La spécification informelle de `strcpy`, que l'on trouve dans le manuel UNIX, dit que :
- le pointeur `src` pointe sur un tableau de caractères terminé par 0 ;
- le pointeur `dest` pointe sur une zone mémoire avec suffisamment d'espace alloué.

Il nous faut donc spécifier ceci formellement, dans le langage d'annotations de CADUCEUS. Pour exprimer que le tableau `src` est terminé proprement par un zéro, on introduit le prédicat suivant :

```
/*@ predicate is_c_string(char *src, int n) {
  @   n >= 0 && \validrange(src,0,n) && src[n] == 0 &&
  @   (\forall int i; 0 <= i < n => src[i] != 0) }
  @*/
```

```
/*@ requires
  @   \baseaddr(src) != \baseaddr(dest) &&
  @   (\exists int n;
  @      is_c_string(src,n) && \validrange(dest,0,n))
  @ ensures
  @   \forall int n ;  is_c_string(src,n) =>
  @      \forall int k; 0 <= k <= n => dest[k] == src[k]
  @*/
void strcpy(char *dest, const char *src)
```

FIG. 3.8 – Spécification de la fonction `strcpy`

```
void strcpy(char *dest, const char *src) {
  /*@ invariant
    @   \old(dest) <= dest && \old(src) <= src &&
    @   dest - \old(dest) == src - \old(src) &&
    @   (\forall char *p;  \old(dest) <= p < dest =>
    @          *p == *(\old(src) + (p-\old(dest)))) &&
    @   (\forall int n ;  is_c_string(\old(src),n) =>
    @      dest <= \old(dest)+n && src <= \old(src) + n)
    @ loop_assigns dest[*]
    @*/
  while (*dest++ = *src++);
}
```

FIG. 3.9 – Code annoté de la fonction `strcpy`

qui exprime que `src` est une chaîne de caractère C bien formée de taille $n$ : toutes les cases entre $0$ et $n$ sont allouées, la case $n$ vaut 0 et les autres ne sont pas nulles.

On peut alors spécifier la fonction `strcpy` de la façon donnée sur la figure 3.8. La différence avec `memcpy` est qu'ici on suppose l'existence d'un $n$ tel que `src` soit une chaîne de taille $n$.

La preuve de `strcpy` passe naturellement par l'écriture d'un invariant de boucle adapté. L'obtention d'un invariant suffisant ne vient pas naturellement, en fait il nous a fallu plusieurs itérations avec CADUCEUS et le prouveur COQ pour arriver à un invariant suffisant, qui est donné sur la figure 3.9.

La vérification du code annoté de `strcpy` avec le prouveur SIMPLIFY a été faite seulement partiellement :

- invariant de boucle initialement vrai : prouvé ;
- validité des déréférenciations `*src` et `*dest` : prouvée ;
- préservation de l'invariant de boucle : non prouvée ;

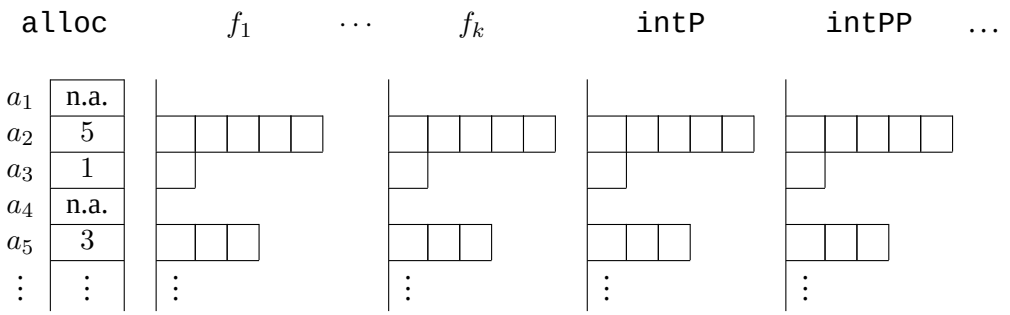



FIG. 3.10 – Représentation à la Burstall-Bornat du tas mémoire de C

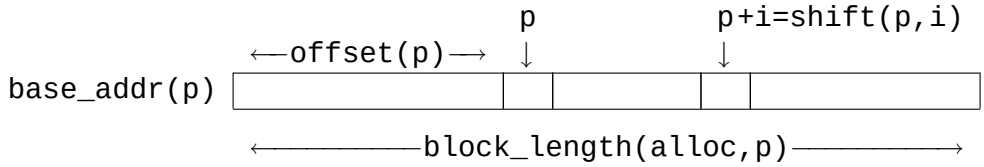
– post-condition établie : non prouvée.

Avec le prouveur Coq, par contre, tout a été prouvé. Cela a nécessité deux lemmes à propos du prédicat `is_c_string` (pour dire que si une chaîne est à la fois de taille $n$ et de taille $m$, alors $n = m$, et pour dire que si l'on change une case d'une chaîne de taille $n$ elle reste une chaîne de taille $n$), et au final 303 lignes de tactiques. La réalisation de ces preuves nécessite une bonne compréhension de la modélisation de la mémoire et de l'arithmétique de pointeurs.

### 3.5.2 Modélisation de l'arithmétique de pointeur

Le modèle de Burstall-Bornat est étendu pour supporter les tableaux de C et l'arithmétique de pointeur en général. En effet, les tableaux de C sont du sucre syntaxique pour l'arithmétique de pointeur : `t[i]` est une abréviation de `*(t+i)`.

Un bloc mémoire de C est vu selon ce schéma :




Chaque champ de structure est une table d'association, qui à chaque adresse associe non pas une cellule mémoire comme dans le cas du modèle Burstall-Bornat initial où de notre modèle pour Java, mais un bloc mémoire. Ceci est schématisé sur la figure 3.10

Les pointeurs sont vus comme des couples (adresse de base, décalage). L'axiomatisation au premier ordre est ainsi :

```
logic base_addr: pointer -> addr
logic offset: pointer -> int

axiom pointer_pair_1 :
  forall p1:pointer. forall p2:pointer.
```

```
  (base_addr(p1) = base_addr(p2) and
    offset(p1) = offset(p2)) -> p1 = p2

axiom pointer_pair_2 :
  forall p1:pointer. forall p2:pointer.
  p1 = p2 -> (base_addr(p1) = base_addr(p2)
                and offset(p1) = offset(p2))
```

La validité de la déréférenciation de pointeur correspond à deux choses :
- l'adresse de base de ce pointeur correspond à un bloc alloué
- la taille de ce bloc est suffisante pour que l'offset de ce pointeur soit à l'intérieur de ce bloc.

Pour exprimer cela, on introduit une variable Why supplémentaire représentant la table d'allocation courante : à chaque adresse de base on associe la taille du bloc alloué à cette adresse, ou bien 0 pour signifier qu'elle n'est pas allouée. Le type logique de cette table est noté `alloc_table`. L'axiomatisation du premier ordre est alors :

```
logic block_length: alloc_table, pointer -> int

axiom base_addr_block_length :
  forall a:alloc_table. forall p1:pointer.
  forall p2:pointer.
    base_addr(p1) = base_addr(p2) ->
      block_length(a,p1) = block_length(a,p2)

predicate valid(a:alloc_table, p:pointer) =
  0 <= offset(p) < block_length(a,p)
```

Des prédicats de validité supplémentaires s'introduisent facilement, comme celui qui exprime la validité d'un intervalle d'indices :

```
predicate valid_range(a:alloc_table, p:pointer,
                      i:int, j:int) =
  0 <= offset(p)+i <= offset(p)+j < block_length(a,p)
```

L'addition d'un pointeur et d'un entier est ensuite introduite, avec le nom logique `shift` pour bien la distinguer de l'addition des entiers.

```
logic shift: pointer, int -> pointer

axiom offset_shift : forall p:pointer. forall i:int.
  offset(shift(p,i)) = offset(p)+i

axiom shift_zero : forall p:pointer. shift(p,0) = p

axiom shift_shift : forall p:pointer. forall i:int.
```

```
  forall j:int. shift(shift(p,i),j) = shift(p,i+j)

axiom base_addr_shift : forall p:pointer. forall i:int.
  base_addr(shift(p,i)) = base_addr(p)

axiom block_length_shift : forall a:alloc_table.
  forall p:pointer. forall i:int.
  block_length(a,shift(p,i)) = block_length(a,p)
```

En plus de l'addition d'un pointeur et d'un entier, l'arithmétique permet la différence de pointeurs : le résultat est un entier ; et les comparaisons de pointeurs. Dans les deux cas, le résultat n'est défini que pour des pointeurs pointant sur le même bloc.

```
logic sub_pointer: pointer, pointer -> int
axiom sub_pointer_def :
  forall p1:pointer. forall p2:pointer.
  base_addr(p1) = base_addr(p2) ->
  sub_pointer(p1,p2) = offset(p1)-offset(p2)

predicate lt_pointer(p1:pointer,p2:pointer) =
  base_addr(p1) = base_addr(p2)
  and offset(p1) < offset(p2)
```

et de façon analogue `le_pointer`, `gt_pointer` et `ge_pointer`.

Cette théorie de l'arithmétique de pointeurs est originale, et en particulier nous n'avons pas du tout à notre disposition de technique automatique ou au moins systématique de raisonner avec. Sur les exemples que nous avons traités, les preuves en Coq demandent une application manuelle des axiomes précédents, et beaucoup d'étapes de réécriture manuelles. Avec l'outil automatique Simplify, nous avons dû être très attentifs avec les axiomes donnés, car parfois il suffit de donner un nouvel axiome, redondant avec les précédents, dans l'idée d'aider le prouveur, pour qu'en fait l'outil se mette à boucler indéfiniment.

Une question restée ouverte est donc comment s'y prendre pour raisonner systématiquement sur cette théorie. Et aussi d'un point de vue théorique, est-ce que cette théorie est décidable ? La réponse à cette question n'est certainement pas simple, en particulier parce que cette théorie contient l'arithmétique linéaire sur les entiers.

### 3.5.3 Invariants de structure

Dans notre approche actuelle des programmes C, il est possible de poser un invariant sur chaque variable globale, indépendemment les unes des autres. Il y a un besoin évident d'attacher un invariant à un type, en particulier de structure, devant être satisfait par chaque variable de ce type. Pour faire cela, on se heurte aux mêmes problèmes que pour les invariants de classe de Java, détaillés à la section 3.4.5. Ceci est actuellement en cours d'étude.

## 3.6 Études de cas

J'ai eu l'occasion de mettre en œuvre les outils Krakatoa et Caduceus sur des études de cas de taille conséquente. L'objet de cette section est de présenter chacune d'elles.

De manière générale, il faut noter que les exemples traités sont d'une complexité telle que la preuve entièrement automatique, par un outil comme Simplify, n'a jamais abouti. Les preuves de cette section ont toujours été faites à l'aide de Coq.

### 3.6.1 Programmes JavaCard

Les programmes JavaCard sont, au niveau du source, très proches des programmes Java. C'est pourquoi notre outil Krakatoa peut s'utiliser sur les deux. Il y a des différences plus importantes au niveau de la machine virtuelle. Une différence importante pour nous, dans la mesure où cela influe sur les programmes sources, est l'absence de tableau de dimension 2 ou plus. Cela permet un typage dynamique grandement simplifié dans la JCVM, puisque le langage des types n'est plus récursif. Cette limitation est contournée par les programmeurs JavaCard en simulant un tableau de dimension 2 par un tableau de dimension 1 de type Object[], et en castant ses éléments vers un tableau. Autrement dit c'est le programmeur qui introduit les opérations de typage dynamique nécessaires. Pour la preuve, ce genre de choses nécessite de rendre explicite ce codage par des invariants de classe.

Une autre différence importante est l'API, spécifique à JavaCard. D'où la nécessité pour l'outil KRAKATOA de permettre le choix de la bibliothèque de packages à importer : il faut préciser que l'on désire utiliser cette API, et il faut aussi fournir une version annotée en JML de cette API. Une telle version était proposée par le groupe LOOP, que nous avons utilisée mais en la modifiant pour l'adapter à nos besoins (les annotations n'étaient pas complètes de toute facon).

La première étude de cas que j'ai traité était une applette JavaCard proposée par la société Schlumberger (maintenant Axalto) dans le cadre du projet VerifiCard. En fait, au cours cette étude de cas, l'outil Krakatoa a été développé et amélioré (ainsi que l'outil WHY) pour résoudre des problèmes d'efficacité de l'outil. Quant à l'étude elle-même, elle a été poursuivie en même temps avec d'autres outils en collaboration avec Bart Jacobs (Loop) et Nicole Rauch (ESC/Java et Jive) et nous avons réussi à traiter une bonne partie de l'applette et même trouvé quelques problèmes de correction. Les détails sont décrits dans un rapport [26] et un article de conférence [80].

J'ai encadré un stage d'un étudiant, Vikrant Chaudhary, sur une autre étude de cas : une applette JavaCard proposée comme challenge académique par Trusted Logic. Nous avons sur cet exemple étudié des propriétés de haut-niveau, sur la vérification des droits d'accès, en traduisant des machines à états vers JML. Nous avons lors de cette étude identifié aussi des problèmes dans l'applette.

### 3.6.2 Pointeurs et aliasing

> *The Schorr-Waite algorithm is the first moutain that any formalism for pointer aliasing should climb.*
>
> — RICHARD BORNAT ([22], page 121)

Notre approche est adaptée aux programmes qui comportent des alias de pointeurs. Pour démontrer la validité de cette affirmation, et conformément à l'affirmation ci-dessus, nous avons souhaité refaire les exemples représentatifs traités par Richard Bornat. Ainsi, la fonction C de renversement en place d'une liste chaînées était donné comme exemple dans mon article avec Jean-Christophe Filliâtre en 2004 [61]. La preuve de l'algorithme de parcours de graphe de Schorr et Waite, qui requiert une preuve longue et difficile, a fait l'objet d'un article avec Thierry Hubert en 2005 [75]. Enfin, la preuve du fusion de deux listes triées a été traité par Julien Roussel, dont j'ai encadré le stage de 5eme année de l'EPITA en 2005.

Julien Roussel a également tenté de prouver des versions Java de ces mêmes algorithmes, mais il s'est heurté à des manques au niveau du langage JML : la construction `\old` n'est pas autorisée dans les invariants de boucle (ce manque a été signalé sur la mailing-list de JML et on peut espérer qu'il sera comblé dans un proche avenir), et les *variables de spécification* (c.-à-d. dont la portée couvre à la fois la pré et la post-condition) ne sont pas supportées par les outils JML, bien que décrites dans le manuel de référence.

### 3.6.3 Autres Algorithmes

D'autres preuves d'algorithmes ont été étudiés : recherche du plus court chemin de Dijkstra (stage de Cyril Roux en 2005) et l'algorithme d'Ukkonen (stage de Nicolas Guenot en 2005). Ces preuves n'ont pas été terminées, bien qu'aucun manque clair n'ait été identifié, il s'agit là seulement d'un manque de temps. Il faut noter que sur ces exemples, ce qui est compliqué est la modélisation logique qui doit être construite rien que pour spécifier leur comportement attendu des programmes considérés. La complexité de ces algorithmes est apparement trop grande pour que de jeunes stagiaires (Licence et Master 1 en l'occurrence) arrivent au bout dans le temps imparti.

### 3.6.4 Programmes C embarqués

Caduceus est actuellement en cours d'expérimentation sur des programmes C critiques : à Axalto et à Dassault aviation.

## 3.7 Conclusions et perspectives

La méthodologie Why de Jean-Christophe Filliâtre, basé sur une interprétation Coq des programmes impératifs, permet une grande souplesse. D'une part, avec l'outil Why d'autres prouveurs que Coq peuvent être utilisés pour établir la validité des obligations de preuve.

D'autre part, la méthode de génération des obligations sépare clairement ce qui est du ressort du flot de contrôle (conditionnelles, boucles, exceptions) et ce qui ne dépend que des données. Cette séparation des tâches a permis en un temps très réduit de mettre en place des outils pour les programmes Java et C, car il suffit dans chacun des cas de construire un modèle des données complexes de ces langages sous forme de variables globales modifiables de Why, et d'opérations logiques sur ces variables.

Grâce à cette approche, avec Jean-Christophe Filliâtre, Christine Paulin et Xavier Urbain, nous avons pu très rapidement proposer des outils intéressant les industriels. Mon objectif scientifique général est maintenant d'intégrer à notre approche les apports d'autres techniques de preuves de programme reconnues, en particulier les analyses statiques avancées de code. Je détaillerai des perspectives de recherche au chapitre 4.

# Chapitre 4

# Conclusions et perspectives

Dans ce chapitre de conclusions, je présente quelques perspectives de recherche, sur chacun de mes deux grands axes de recherche, avant de discuter de considérations plus générales.

## 4.1 Perspectives sur les preuves de terminaison

Si les avancées effectuées depuis 1997 ont fait beaucoup progresser les outils, comme cela est attesté par les taux de succès obtenus lors de la compétition sur la terminaison, il reste des classes de problèmes restant hors de portée.

D'abord, comme expliqué à la section 2.5.3 dans le cas du système Maude, la terminaison pour d'autres paradigmes par transformation vers la réécriture n'est pas encore satisfaisante. Or il y a intérêt important à doter un système comme Maude d'un outil de terminaison. L'analyse des exemples qui échouent montrent des pistes de recherche :

- La modularité des programmes Maude n'est pas préservée par les transformations, ce qui rend ardu la preuve de terminaison du programme transformé. Ceci révèle donc un aspect négatif de cette transformation, une variante de cette transformation est donc à trouver.
- Les sortes des programmes Maude sont parfois essentielles pour garantir la terminaison. Or la transformation vers la réécriture arrive dans un monde non typé, on y perd donc des informations structurelles. Pour la terminaison de la réécriture d'ordre supérieur, si l'utilisation de système de types est largement étudiée [3, 19], pour la terminaison de systèmes de réécriture du premier ordre multi-sortés ou ordo-sortés, n'est pas le cas.

La compétition d'outils de terminaison peut jouer un rôle moteur dans la résolution de ces problèmes. En effet, les exemples transformés produisent des défis présentés à tous les outils, et on peut espérer que la mise en commun des efforts produira plus rapidement des avancées.

Ensuite, la compétition de terminaison révèle d'autres lacunes des techniques actuelles. Le succès ou l'échec peut varier énormément en fonction des stratégies utilisées par un outil, on retrouve là un phénomène similaire à ce que l'on connait par exemple pour les

preuves de théorème du premier ordre de la compétition CASC. Les problèmes de choix de stratégie sont encore trop souvent vus comme des problèmes techniques, qu'une étude théorique peut parfois rendre obsolètes du jour au lendemain.

Une perspective importante pour la preuve automatique de terminaison est la génération de traces vérifiables. En effet, on sait que les outils, du fait même de la complexité des algorithmes qu'ils mettent en œuvre, contiennent des erreurs de programmation (je peux citer pas moins de quatre de ces outils dont un bug a été révélé à un moment ou à un autre, en produisant parfois une réponse positive sur un système qui ne terminait pas). Le projet CoLoR (`http://color.loria.fr/`) pour la production de traces vérifiables en Coq [70], ainsi que le travail de DEA de Thierry Hubert que j'ai encadré [74] apporte déjà des avancées en ce sens, mais on est encore loin de l'objectif qui serait d'exiger pour chaque outil un langage commun d'expression des traces. Cette problématique de production de trace existe aussi depuis longtemps pour les preuves de théorèmes, avec des avancées lentes [115] (voir par exemple la thèse de Pierre Corbineau [36] que j'ai partiellement encadrée) donc il est probable que la génération de trace de preuve de terminaison reste longtemps un défi.

Enfin, la preuve de terminaison de programmes fonctionnels est encore très peu abordée. Cela aurait des applications pour des programmes CAML, mais aussi des programmes Coq et des programmes Why. Une méthode par transformation vers la réécriture est encore une fois envisageable, par exemple j'ai modestement contribué à la certification de la librairie Set de Caml [60] en trouvant, à l'aide d'une transformation manuelle et de CiME, une preuve de terminaison de la fonction de comparaison des ensembles, qui était particulièrement ardue.

## 4.2 Perspectives sur la preuve de programmes impératifs

La méthode d'utilisation de l'outil Why que nous avons mise en œuvre pour la preuve de programmes Java et C offre une souplesse qui permettra à l'avenir d'expérimenter rapidement de nouvelles techniques.

Une première piste que nous allons explorer dans un proche avenir est celui du support de l'arithmétique entière et flottante des ordinateurs. En effet, un des reproches actuels à nos outils Krakatoa et Caduceus est que les entiers des programmes sont interprétés comme des entiers mathématiques, non bornés, et de même les flottants sont interprétés comme des réels mathématiques. La solution que nous prévoyons utilise la souplesse de l'approche WHY : en effet, il suffit lors de la traduction vers WHY de traduire les opérations par des fonctions WHY spécialisées. À titre d'exemple, si l'on souhaite vérifier qu'un programme sur des entiers 32 bits ne fait jamais de dépassement de capacité, on pourra traduire l'addition avec la fonction WHY :

```
parameter bounded_add : x:int -> y:int ->
  { -2^31 <= x+y < 2^31 } int { result = x+y }
```

signifiant que le résultat est égal à l'addition mathématique, sous la précondition que celui est dans les bonnes bornes. Ainsi, non seulement nous pouvons très rapidement ajouter le

support de ces arithmétiques bornées, mais nous pouvons aussi laisser l'utilisateur choisir, par exemple par le biais d'une option de configuration des outils, s'il désire vérifier ou non que il n'y a pas de dépassement de capacité, ou même s'il autorise le dépassement de capacité et souhaite travailler module $2^{32}$. Pour le cas des flottants, nous avons eu le renfort récent de Sylvie Boldo qui a développé une formalisation des flottants IEEE, sur laquelle nous allons nous baser (voir aussi le projet MPFR `http://www.mpfr.org/`).

Cette souplesse permettre de mettre en œuvre rapidement d'autres extensions, par exemple la variante d'interprétation des invariants de classe proposée à la section 3.4.5. Je suis également en train d'étudier, avec Sylvain Boulmé du LSR/IMAG à Grenoble, une version de cette technique pour les invariants des structures C. Avec Thierry Hubert, étudiant en thèse sous ma direction, nous étudions actuellement une modélisation fine de la mémoire, qui procède à une analyse statique sur le programme source, calculant une séparation optimale des pointeurs, de manière à produire une représentation du type de celle de la figure 3.10, mais avec le plus grand nombre de variables WHY différentes.

Le travail d'analyse statique précité est motivé par une étude de cas mené avec Dassault Aviation, sur le code C embarqué de contrôle-commande d'un avion. L'analyse de séparation en question est un besoin identifié lors de cette étude. Mais ce n'est pas le seul : il y a de nombreux besoin d'ajouts d'analyses statiques pour permettre d'engendrer automatiquement des propriétés des programmes sources. En ce sens, mes recherches devront à plus ou moins long terme se combiner avec les techniques d'analyse statique et d'interprétation abstraite qui ont fait le succès d'outils comme Polyspace, Caveat [27] ou Astrée [18].

Pour l'instant nous n'avons pas beaucoup travaillé sur l'automatisation de l'établissement des obligations de preuves, mais nous souhaitons progresser en ce sens, en faisant coopérer les prouveurs. C'est le sens de la thèse de Pierre Corbineau, récemment soutenue, que j'ai co-encadré.

Un défi important pour l'avenir est de fournir des méthodes pour éviter de devoir annoter le code, mais plutôt d'engendrer les annotations de code à partir de spécifications de haut niveau qui aurait été écrite en amont du processus de développement logiciel, lors de la spécification globale. Cela peut être des spécifications à l'aide de machines à états par exemple, du type de celles de la notation UML ou des machines abstraites à la Gurevich [25].

## 4.3 Et à long terme ?

Un groupe d'éminents chercheurs en informatique ont proposé un ensemble de défis pour le 21eme siècle. Le « grand challenge 6 » (`http://www.fmnet.info/gc6/`, `http://vstte.ethz.ch/content.html`) concerne la vérification des systèmes critiques. C'est bien dans cette direction que mes recherches iront à l'avenir. Tony Hoare a écrit une proposition, appelée le *Verifying compiler*, qui a pour objectif d'intégrer systématiquement la vérification de propriétés au processus même de la compilation d'un programme source. C'est un objectif évidemment très ambitieux, mais je suis enclin à croire que l'on peut arriver à faire beaucoup de choses en ce sens.

Au moins deux raisons m'amène à croire que ce défi pourra être réussi. D'abord, il faut bien se rendre compte que des langages comme Objective Caml offrent déjà, grâce à leur

typage sûr, beaucoup plus de vérification à la compilation que ce qui se faisait il y a une trentaine d'année : parmi les premiers langages offrant un typage sûr, citons le langage CLU développé au MIT en 1974, et le langage Caml développé à l'INRIA en 1985. Faut-il rappeler qu'un programme Caml compilé sans erreur ne fait *jamais* d'erreur de segmentation à l'exécution ? C'est évidemment le paradigme de programmation de OCaml qui permet cela. Mais malheureusement les applications, en particulier les systèmes critiques, restent encore très souvent programmés en C en Java. Et justement, la deuxième raison que je voulais invoquer est l'essor des outils comme ESC/Java [49, 2] ou Spec#, qui savent opérer de nombreuses vérifications automatiques de correction des programmes, respectivement en Java et en C#. On peut signaler aussi le fait que le code source de la prochaine version du système d'exploitation Windows de Microsoft, est systématiquement passé à un outil d'analyse statique maison. Les auteurs de ces outils ont souhaité jusqu'à présent proposer des méthodes entièrement automatiques, ce qui n'est pas notre cas car nous pensons qu'il est bien de pouvoir « passer en manuel » quand la complexité de la propriété à établir l'exige. Mais de même que nous cherchons à introduire plus d'automatisation dans nos preuves, ces outils commencent à s'ouvrir à la preuve interactive : par exemple, la toute dernière version de ESC/Java2 possède une sortie pour PVS.

Des grands projets en cours confirment l'activité dans le domaine de la preuve de programme, on peut citer par exemple le projet Verisoft (`http://www.verisoft.de/index_en.html`) et le projet Concert (`http://www-sop.inria.fr/lemme/concert/`) de compilateurs certifiés [20].

Les outils existants sont à mon avis maintenant suffisamment matures pour commencer à être pratiqué dans l'enseignement, mais c'est encore très rarement le cas aujourd'hui. On pourrait espèrer qu'au moins un outil comme ESC/Java soit maintenant pratiqué dans les cours de Java. (Il faut signaler de tout de même que, à ma connaissance, Krakatoa a été utilisé à l'Université d'Evry, et Caduceus à l'Université de Valenciennes.)

Il est clair qu'il ne faut pas se limiter à traiter le Java et le C. Les langages ML et C++ sont des candidats où il y a un intérêt à développer des techniques de vérification automatiques ou semi-automatiques.

Un défi très important est celui du support des programmes concurrents. Même si des travaux anciens (Lamport, 1977 [93]) ont été produits, ce n'est que récemment que des méthodes plus appliquées sont étudiées, avec en particulier le mécanisme des *assume-guarantee*, bien adapté pour raisonner composants par composants.

La vérification formelle de programmes est donc promise à un bel essor, et notre approche par traduction fonctionnelle, originale par rapport aux autres, nous permet d'espérer y contribuer de manière significative.

# Bibliographie


[1] ESC/Java. `http://research.compaq.com/SRC/esc/`.

[2] ESC/Java2. `http://www.sos.cs.ru.nl/research/escjava`.

[3] A. ABEL. Termination checking with types. *RAIRO – Theoretical Informatics and Applications*, vol. 38, n° 4, pp. 277–319, 2004. Special Issue : Fixed Points in Computer Science (FICS'03).

[4] W. AHRENDT, T. BAAR, B. BECKERT, R. BUBEL, M. GIESE, R. HÄHNLE, W. MENZEL, W. MOSTOWSKI, A. ROTH, S. SCHLAGER, et P. H. SCHMITT. The KeY tool. *Software and System Modeling*, vol. 4, pp. 32–54, 2005.

[5] K. R. APT. Ten years of Hoare's logic : A survey. *ACM Transactions on Programming Languages and Systems*, vol. 3, n° 4, pp. 431–483, octobre 81.

[6] T. ARTS. *Automatically proving termination and innermost normalisation of term rewriting systems*. PhD thesis, Universiteit Utrecht, 1997.

[7] T. ARTS et J. GIESL. Automatically proving termination where simplification orderings fail. In M. BIDOIT et M. DAUCHET (éd.), *Theory and Practice of Software Development*, coll. *Lecture Notes in Computer Science*, vol. 1214, Lille, France, avril 1997. Springer.

[8] T. ARTS et J. GIESL. Modularity of termination using dependency pairs. In Nipkow [116], pp. 226–240.

[9] T. ARTS et J. GIESL. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, vol. 236, pp. 133–178, 2000.

[10] F. BAADER et T. NIPKOW. *Term Rewriting and All That*. Cambridge University Press, 1998.

[11] L. BACHMAIR. Associative-commutative reduction orderings. *Info Proc. Letters*, 1992.

[12] L. BACHMAIR et N. DERSHOWITZ. Commutation, transformation, and termination. In J. H. SIEKMANN (éd.), *Proc. 8th Int. Conf. on Automated Deduction, Oxford, England, LNCS 230*, pp. 5–20, juillet 1986.

[13] L. BACHMAIR et D. A. PLAISTED. Termination orderings for associative-commutative rewriting systems. *Journal of Symbolic Computation*, vol. 1, n° 4, pp. 329–349, décembre 1985.

[14] M. BARNETT, R. DELINE, M. FÄHNDRICH, K. R. M. LEINO, et W. SCHULTE. Verification of object-oriented programs with invariants. *Journal of Object Technology*, vol. 3, n° 6, pp. 27–56, juin 2004.

[15] F. BELLEGARDE et P. LESCANNE. Transformation orderings. In *Proc. CAAP 87, Pisa, LNCS 249*. Springer, mars 1987.

[16] A. BEN CHERIFA et P. LESCANNE. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, vol. 9, pp. 137–159, 1987.

[17] A. BEN CHERIFA et P. LESCANNE. An actual implementation of a procedure that mechanically proves termination of rewriting systems based on inequalities between polynomial interpretations. In Siekmann [129], pp. 42–51.

[18] B. BLANCHET, P. COUSOT, R. COUSOT, J. FERET, L. MAUBORGNE, A. MINÉ, D. MONNIAUX, et X. RIVAL. The Astrée static analyzer. `http://www.astree.ens.fr/`.

[19] F. BLANQUI. A type-based termination criterion for dependently-typed higher-order rewrite systems. In V. VAN OOSTROM (éd.), *15th International Conference on Rewriting Techniques and Applications*, coll. *Lecture Notes in Computer Science*, vol. 3091, pp. 24–39, Aachen, Germany, juin 2004. Springer.

[20] S. BLAZY et X. LEROY. Formal verification of a memory model for -like imperative languages. In K.-K. LAU et R. BANACH (éd.), *ICFEM*, coll. *Lecture Notes in Computer Science*, vol. 3785, pp. 280–299. Springer, 2005.

[21] G. BONFANTE, A. CICHON, J.-Y. MARION, et H. TOUZET. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming*, vol. 11, n° 1, pp. 33–53, mars 2001.

[22] R. BORNAT. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pp. 102–126, 2000.

[23] L. BURDY. JACK : Java Applet Correctness Kit. Gemplus Developer Conference, 2002.

[24] R. BURSTALL. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, vol. 7, pp. 23–50, 1972.

[25] E. BÖRGER. Ten years of gurevich's abstract state machines. *Journal of Universal Computer Science*, vol. 3, n° 4, pp. 230–232, 1997. `http://www.jucs.org/jucs_3_4/introduction`.

[26] N. CATAÑO, M. GAWKOWSKI, M. HUISMAN, B. JACOBS, C. MARCHÉ, C. PAULIN, E. POLL, N. RAUCH, et X. URBAIN. Logical techniques for applet verification. Deliverable 5.2, IST VerifiCard Project, 2003. `http://www.cs.kun.nl/VerifiCard/files/deliverables/deliverable_5_2.pdf`.

[27] CAVEAT project. `http://www-drt.cea.fr/Pages/List/lse/LSL/Caveat/index.html`.

[28] V. CHAUDHARY. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML : A case study. Technical report, IIT internship report, juillet 2004.

[29] A. CICHON et P. LESCANNE. Polynomial interpretations and the complexity of algorithms. In D. KAPUR (éd.), *11th International Conference on Automated Deduction*, coll. *Lecture Notes in Computer Science*, vol. 607, pp. 139–147, Saratoga Springs, NY, juin 1992. Springer.

[30] M. CLAVEL, F. DURÁN, S. EKER, P. LINCOLN, N. MARTÍ-OLIET, J. MESEGUER, et J. F. QUESADA. Maude : Specification and programming in rewriting logic. *Theoretical Computer Science*, vol. 285, n° 2, pp. 187–243, août 2002.

[31] E. CONTEJEAN et C. MARCHÉ. CiME : Completion Modulo $E$. In H. GANZINGER (éd.), *7th International Conference on Rewriting Techniques and Applications*, coll. *Lecture Notes in Computer Science*, vol. 1103, pp. 416–419, New Brunswick, NJ, USA, juillet 1996. Springer. System Description available at `http://cime.lri.fr/`.

[32] E. CONTEJEAN, C. MARCHÉ, B. MONATE, et X. URBAIN. Proving termination of rewriting with CiME. In Rubio [128], pp. 71–73. Technical Report DSIC II/15/03, Universidad Politécnica de Valencia, Spain.

[33] E. CONTEJEAN, C. MARCHÉ, et L. RABEHASAINA. Rewrite systems for natural, integral, and rational arithmetic. In H. COMON (éd.), *8th International Conference on Rewriting Techniques and Applications*, coll. *Lecture Notes in Computer Science*, vol. 1232, pp. 98–112, Barcelona, Spain, juin 1997. Springer.

[34] E. CONTEJEAN, C. MARCHÉ, A. P. TOMÁS, et X. URBAIN. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, vol. 34, n° 4, pp. 325–363, 2005.

[35] T. COQUAND et G. HUET. The calculus of constructions. *Information and Computation*, vol. 76, pp. 95–120, février 1988.

[36] P. CORBINEAU. *Démonstration Automatique en Théorie des Types*. Thèse de doctorat, Université Paris-Sud, septembre 2005.

[37] P. COUSOT. Methods and logics for proving programs. In J. VAN LEEUWEN (éd.), *Handbook of Theoretical Computer Science*, vol. B, pp. 841–993. North-Holland, 1990.

[38] J. DAVENPORT et J. HEINTZ. Real quantifier elimination is doubly exponential. Unpublished draft, 1987.

[39] M. DAVIS, H. PUTNAM, et J. ROBINSON. The decision problem for exponential Diophantine equations. *Annals of Mathematics*, vol. 74, pp. 425–436, 1961.

[40] É. DEPLAGNE. Sequent Calculus Viewed Modulo. In CATHERINE PILIÈRE (éd.), *Proceedings of the Fifth ESSLLI Student Session*, pp. 66–76, University of Birmingham, 2000.

[41] É. DEPLAGNE. *Système de preuve modulo récurrence*. Thèse de doctorat, Université Henry Poincaré-Nancy-I, novembre 2002.

[42] N. DERSHOWITZ. A note on simplification orderings. *Information Processing Letters*, vol. 9, n° 5, pp. 212–215, novembre 1979.

[43] N. DERSHOWITZ. Orderings for term rewriting systems. *Theoretical Computer Science*, vol. 17, n° 3, pp. 279–301, mars 1982.

[44] N. DERSHOWITZ. Termination of rewriting. *Journal of Symbolic Computation*, vol. 3, n° 1, pp. 69–115, février 1987.

[45] N. DERSHOWITZ. Hierarchical termination. In N. DERSHOWITZ et N. LINDENSTRAUSS (éd.), *Proceedings of the Fourth International Workshop on Conditional and Typed Rewriting Systems (Jerusalem, Israel, July 1994)*, vol. 968, pp. 89–105, Berlin, 1995. Springer-Verlag.

[46] N. DERSHOWITZ et J.-P. JOUANNAUD. Rewrite systems. In J. VAN LEEUWEN (éd.), *Handbook of Theoretical Computer Science*, vol. B, pp. 243–320. North-Holland, 1990.

[47] N. DERSHOWITZ et Z. MANNA. Proving termination with multiset orderings. *Communications of the ACM*, vol. 22, n° 8, pp. 465–476, août 1979.

[48] N. DERSHOWITZ et R. TREINEN. An on-line problem database. In Nipkow [116], pp. 332–342.

[49] D. L. DETLEFS, K. R. M. LEINO, G. NELSON, et J. B. SAXE. Extended static checking. Technical Report 159, Compaq Systems Research Center, décembre 1998. See also `http://research.compaq.com/SRC/esc/`.

[50] E. W. DIJKSTRA. *A discipline of programming*. Series in Automatic Computation. Prentice Hall Int., 1976.

[51] P. J. DOWNEY, R. SETHI, et R. E. TARJAN. Variations on the common subexpressions problem. *Journal of the ACM*, vol. 27, n° 4, pp. 771–785, 1980.

[52] F. DURAN, S. LUCAS, C. MARCHÉ, J. MESEGUER, et X. URBAIN. Termination of membership equational programs. Submitted.

[53] F. DURÁN, S. LUCAS, J. MESEGUER, C. MARCHÉ, et X. URBAIN. Proving termination of membership equational programs. In *ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation*, Verona, Italy, août 2004. ACM Press.

[54] M. FERNÁNDEZ et J.-P. JOUANNAUD. Modular termination of term rewriting systems revisited. In E. ASTESIANO, G. REGGIO, et A. TARLECKI (éd.), *Recent Trends in Data Type Specification*, Lecture notes in Computer Science, vol.906, pp. 255–272. Springer, 1995. Refereed selection of papers presented at ADT'94.

[55] M. C. FERREIRA, D. KESNER, et L. PUEL. Reducing AC-Termination to Termination. In L. BRIM, J. GRUSKA, et J. ZLATUSKA (éd.), *Mathematical Foundations of Computer Science*, coll. *Lecture Notes in Computer Science*, vol. 1450, pp. 239–247, Brno, Czech Republic, août 1998. Springer.

[56] J.-C. FILLIÂTRE. *Preuve de programmes impératifs en théorie des types*. PhD thesis, Université Paris-Sud, juillet 1999.

[57] J.-C. FILLIÂTRE. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, vol. 13, n° 4, pp. 709–745, juillet 2003.

[58] J.-C. FILLIÂTRE. Why : a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, mars 2003. `http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz`.

[59] J.-C. FILLIÂTRE, T. HUBERT, et C. MARCHÉ. The Caduceus tool for the verification of C programs. `http://why.lri.fr/caduceus/`.

[60] J.-C. FILLIÂTRE et P. LETOUZEY. Functors for Proofs and Programs. In *Proceedings of The European Symposium on Programming*, coll. *Lecture Notes in Computer Science*, vol. 2986, pp. 370–384, Barcelona, Spain, avril 2004.

[61] J.-C. FILLIÂTRE et C. MARCHÉ. Multi-prover verification of C programs. In J. DAVIES, W. SCHULTE, et M. BARNETT (éd.), *6th International Conference on Formal Engineering Methods*, coll. *Lecture Notes in Computer Science*, vol. 3308, pp. 15–29, Seattle, WA, USA, novembre 2004. Springer.

[62] R. W. FLOYD. Assigning meanings to programs. In J. T. SCHWARTZ (éd.), *Mathematical Aspects of Computer Science*, coll. *Proceedings of Symposia in Applied Mathematics*, vol. 19, pp. 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[63] R. FORGAARD et D. DETLEFS. Reve 2.4 : A program for generating and analyzing term rewriting systems. Massachusetts Institute, 1984.

[64] H. GANZINGER et U. WALDMANN. Termination proofs of well-moded logic programs via conditional rewrite systems. In *3rd International Workshop on Conditional Term Rewriting Systems*, coll. *Lecture Notes in Computer Science*, vol. 656, pp. 113–127, Berlin, 1993. Springer.

[65] J. GIESL. Generating polynomial orderings for termination proofs. In J. HSIANG (éd.), *6th International Conference on Rewriting Techniques and Applications*, coll. *Lecture Notes in Computer Science*, vol. 914, pp. 426–431, Kaiserslautern, Germany, avril 1995. Springer.

[66] J. GIESL et D. KAPUR. Dependency pairs for equational rewriting. In A. MIDDELDORP (éd.), *12th International Conference on Rewriting Techniques and Applications*, coll. *Lecture Notes in Computer Science*, vol. 2051, pp. 93–108, Utrecht, The Netherlands, mai 2001. Springer.

[67] I. GNAEDIG et P. LESCANNE. Proving termination of associative commutative rewriting systems by rewriting. In Siekmann [129], pp. 52–61.

[68] B. GRAMLICH. Generalized sufficient conditions for modular termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing*, vol. 5, pp. 131–158, 1994.

[69] J. GROOTE et A. PONSE. The syntax and semantics of $\mu$CRL. In A. PONSE, C. VERHOEF, et S. VAN VLIJMEN (éd.), *Algebra of Communicating Processes*, pp. 26–62, 1995.

[70] S. HINDERER. Certification des preuves de terminaison par interprétations polynomiales. Dea thesis, LORIA, 2004. `http://www.loria.fr/publications/2004/A04-R-489/A04-R-489.ps`.

[71] C. A. R. HOARE. An axiomatic basis for computer programming. *Communications of the ACM*, vol. 12, n° 10, pp. 576–580 and 583, octobre 1969.

[72] D. HOFBAUER et C. LAUTERMANN. Termination proofs and the length of derivations. In N. DERSHOWITZ (éd.), *Rewriting Techniques and Applications*, coll. *Lecture Notes in Computer Science*, vol. 355, pp. 167–177, Chapel Hill, U.S.A., avril 1989. Springer.

[73] H. HONG et D. JAKUŠ. Testing Positiveness of Polynomials. *Journal of Automated Reasoning*, vol. 21, n° 1, pp. 23–38, août 1998.

[74] T. HUBERT. Certification des preuves de terminaison en Coq. Rapport de DEA, Université Paris 7, septembre 2004. In French.

[75] T. HUBERT et C. MARCHÉ. A case study of C source code verification : the Schorr-Waite algorithm. In B. K. AICHERNIG et B. BECKERT (éd.), *3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, Koblenz, Germany, septembre 2005. IEEE Comp. Soc. Press.

[76] G. HUET. A uniform approach to type theory. Research Report 795, INRIA, février 1988.

[77] G. HUET. Type theory, specification languages and program verification, aug 1995. Working Material for the lectures of G.Huet.

[78] J.-M. HULLOT. Associative commutative pattern matching. In *Proc. 6th IJCAI (Vol. I), Tokyo*, pp. 406–412, août 1979.

[79] B. JACOBS. Loop project. `http://www.cs.kun.nl/~bart/LOOP`.

[80] B. JACOBS, C. MARCHÉ, et N. RAUCH. Formal verification of a commercial smart card applet with multiple tools. In *Algebraic Methodology and Software Technology*, coll. *Lecture Notes in Computer Science*, vol. 3116, Stirling, UK, juillet 2004. Springer.

[81] S. KAMIN et J.-J. LÉVY. Two generalizations of the recursive path ordering. Available as a report of the department of computer science, University of Illinois at Urbana-Champaign, 1980.

[82] D.-M. KERNIGHAN et B.-W. RITCHIE. *Le langage C ANSI (2ème édition)*. Masson, 1990.

[83] D. E. KNUTH. *The art of computer programming*, vol. 2. Addison-Wesley, 2nd edition, 1981.

[84] D. E. KNUTH et P. B. BENDIX. Simple word problems in universal algebras. In J. LEECH (éd.), *Computational Problems in Abstract Algebra*, pp. 263–297. Pergamon Press, 1970.

[85] M. R. K. KRISHNA RAO. Simple termination of hierarchical combinations of term rewriting systems. In *Theoretical Aspects of Computer Software*, coll. *Lecture Notes in Computer Science*, vol. 789, pp. 203–223. Springer, 1994.

[86] M. R. K. KRISHNA RAO. Modular proofs for completeness of hierarchical term rewriting systems. *Theoretical Computer Science*, vol. 151, pp. 487–512, 1995.

[87] M. KURIHARA et A. OHUCHI. Modularity of simple termination of term rewriting systems. *Journal of Information Processing Society, Japan*, vol. 34, pp. 632–642, 1990.

[88] M. KURIHARA et A. OHUCHI. Modularity of simple termination of term rewriting systems with shared constructors. *Theoretical Computer Science*, vol. 103, pp. 273–282, 1992.

[89] M. KURIHARA et A. OHUCHI. Decomposable termination of composable term rewriting systems. *IEICE*, vol. E78–D, n$^o$ 4, pp. 314–320, avril 1995.

[90] K. KUSAKARI, C. MARCHÉ, et X. URBAIN. Termination of associative-commutative rewriting using dependency pairs criteria. Research Report 1304, LRI, 2002. `http://www.lri.fr/~urbain/textes/rr1304.ps.gz`.

[91] K. KUSAKARI, M. NAKAMURA, et Y. TOYAMA. Argument filtering transformation. In G. NADATHUR (éd.), *Principles and Practice of Declarative Programming, International Conference PPDP'99*, coll. *Lecture Notes in Computer Science*, vol. 1702, pp. 47–61, Paris, 1999. Springer.

[92] K. KUSAKARI et Y. TOYAMA. On proving AC-termination by AC-dependency pairs. *IEICE Transactions on Information and Systems*, vol. E84-D, n$^o$ 5, pp. 604–612, 2001.

[93] L. LAMPORT. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering,*, vol. 2, pp. 125–143, 1977.

[94] D. S. LANKFORD. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Mathematics Department, Louisiana Tech. Univ., 1979. Available at `http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html`.

[95] P. LESCANNE. Computer experiments with the REVE term rewriting system generator. In *Proc. 10th ACM Symp. on Principles of Programming Languages, Austin, Texas*, 1983.

[96] P. LESCANNE. Termination of rewrite systems by elementary interpretations. In H. KIRCHNER et G. LEVI (éd.), *3th International Conference on Algebraic and Logic Programming*, coll. *Lecture Notes in Computer Science*, vol. 632, pp. 21–36, Volterra, Italy, septembre 1992. Springer.

[97] P. LETOUZEY. *Programmation fonctionnelle certifiée : l'extraction de programmes dans l'assistant Coq*. Thèse de doctorat, Université Paris-Sud, juillet 2004.

[98] S. LUCAS. Mu-term, a tool for proving termination of rewriting with replacement restrictions, 2003. Available at `http://www.dsic.upv.es/~slucas/csr/termination/muterm/`.

[99] S. LUCAS, C. MARCHÉ, et J. MESEGUER. Operational termination of conditional term rewriting systems. *Information Processing Letters*, vol. 95, pp. 446–453, 2005.

[100] Z. MANNA et S. NESS. On the termination of markov algorithms. In *Proc. third Hawaii International Conference on Systems Sciences*, pp. 789–792, Honolulu, HI, 1970. `http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html`.

[101] C. MARCHÉ. *Réécriture modulo une théorie présentée par un système convergent et décidabilité des problèmes du mot dans certaines classes de théories équationnelles*. Thèse de doctorat, Université Paris-Sud, Orsay, France, octobre 1993.

[102] C. MARCHÉ. Normalised rewriting and normalised completion. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pp. 394–403, Paris, France, juillet 1994. IEEE Comp. Soc. Press.

[103] C. MARCHÉ, C. PAULIN, et X. URBAIN. The KRAKATOA proof tool, 2002. `http://krakatoa.lri.fr/`.

[104] C. MARCHÉ et C. PAULIN-MOHRING. Reasoning about Java programs with aliasing and frame conditions. In J. HURD et T. MELHAM (éd.), *18th International Conference on Theorem Proving in Higher Order Logics*, coll. *Lecture Notes in Computer Science*, vol. 3603, pp. 179–194. Springer, août 2005.

[105] C. MARCHÉ, C. PAULIN-MOHRING, et X. URBAIN. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, vol. 58, n° 1–2, pp. 89–106, 2004. `http://krakatoa.lri.fr`.

[106] C. MARCHÉ et X. URBAIN. Termination of associative-commutative rewriting by dependency pairs. In Nipkow [116], pp. 241–255.

[107] C. MARCHÉ et X. URBAIN. Modular and incremental proofs of AC-termination. *Journal of Symbolic Computation*, vol. 38, pp. 873–897, 2004.

[108] P. MARTIN-LÖF. *Intuitionistic type theory*. Bibliopolis, 1984.

[109] The MAUDE System. `http://maude.cs.uiuc.edu/`.

[110] B. MEYER. *Object-Oriented Software Construction*. Prentice Hall PTR, March 2000.

[111] A. MIDDELDORP et Y. TOYAMA. Completeness of combinations of constructor systems. *Journal of Symbolic Computation*, vol. 15, pp. 331–348, 1993.

[112] C. P. MOHRING et B. WERNER. Synthesis of ml programs in the system coq. *Journal of Symbolic Computation*, vol. 15, pp. 607–640, 1993.

[113] F. L. MORRIS et C. B. JONES. An early program proof by alan turing. *IEEE Annals of the History of Computing*, vol. 6, n° 2, pp. 139–143, avril 1984.

[114] G. NELSON et D. C. OPPEN. Fast decision procedures based on congruence closure. *Journal of the ACM*, vol. 27, pp. 356–364, 1980.

[115] Q.-H. NGUYEN, C. KIRCHNER, et H. KIRCHNER. External rewriting for skeptical proof assistants. *Journal of Automated Reasoning*, vol. 29, n° 3-4, pp. 309–336, 2002.

[116] T. NIPKOW (éd.). *9th International Conference on Rewriting Techniques and Applications*, coll. *Lecture Notes in Computer Science*, vol. 1379, Tsukuba, Japan, avril 1998. Springer.

[117] E. OHLEBUSCH. *Advanced Topics in Term Rewriting*. Springer, 2002.

[118] E. OHLEBUSCH. On the modularity of termination of term rewriting systems. *Theoretical Computer Science*, vol. 136, pp. 333–360, 1994.

[119] E. OHLEBUSCH. Modular properties of composable term rewriting systems. *Journal of Symbolic Computation*, vol. 20, pp. 1–41, 1995.

[120] E. OHLEBUSCH. Transforming conditional rewrite systems with extra variables into unconditional systems. In *6th International Conference on Logic for Programming and Automated Reasoning*, coll. *Lecture Notes in Artificial Intelligence*, vol. 1705, pp. 111–130, Berlin, 1999.

[121] E. OHLEBUSCH. *Advanced Topics in Term Rewriting*. Springer, avril 2002.

[122] E. OHLEBUSCH, C. CLAVES, et C. MARCHÉ. TALP : A tool for the termination analysis of logic programs. In L. BACHMAIR (éd.), *11th International Conference on Rewriting Techniques and Applications*, coll. *Lecture Notes in Computer Science*, vol. 1833, pp. 270–273, Norwich, UK, juillet 2000. Springer. Available at `http://bibiserv.techfak.uni-bielefeld.de/talp/`.

[123] E. OHLEBUSCH, C. CLAVES, et C. MARCHÉ. The talp tool for termination analysis of logic programs. In Rubio [128]. `http://bibiserv.techfak.uni-bielefeld.de/talp/`.

[124] G. E. PETERSON et M. E. STICKEL. Complete sets of reductions for some equational theories. *Journal of the ACM*, vol. 28, n° 2, pp. 233–264, avril 1981.

[125] J. A. ROBINSON. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, vol. 12, n° 1, pp. 23–41, 1965.

[126] J. ROUYER. Preuves de terminaison de systèmes de réécriture fondées sur les interprétations polynomiales. Une méthode basée sur le théorème de Sturm. *R.A.I.R.O.*, vol. 25, n° 2, pp. 157–169, 1991.

[127] A. RUBIO. A fully syntactic AC-RPO. *Information and Computation*, vol. 178, n° 2, pp. 515–533, 2002.

[128] A. RUBIO (éd.). *Extended Abstracts of the 6th International Workshop on Termination, WST'03*, juin 2003. Technical Report DSIC II/15/03, Universidad Politécnica de Valencia, Spain.

[129] J. H. SIEKMANN (éd.). *8th International Conference on Automated Deduction*, coll. *Lecture Notes in Computer Science*, vol. 230, Oxford, England, 1986. Springer.

[130] J. STEINBACH. Improving associative path orderings. In *Proc. 10th Int. Conf. on Automated Deduction, Kaiserslautern, LNCS 449*. Springer, juillet 1990.

[131] J. STEINBACH. Proving polynomials positive. In R. SHYAMASUNDAR (éd.), *Foundations of Software Technology and Theoretical Computer Science*, coll. *Lecture Notes in Computer Science*, vol. 652, pp. 191–202, New Delhi, India, décembre 1992. Springer.

[132] J. STEINBACH. Generating polynomial orderings. *Information Processing Letters*, vol. 49, pp. 85–93, 1994.

[133] M. E. STICKEL. A unification algorithm for associative-commutative functions. *Journal of the ACM*, vol. 28, n° 3, pp. 423–434, 1981.

[134] A. TARSKI. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, Berkeley, CA, 1951.

[135] TERESE. *Term Rewriting Systems*, coll. *Cambridge Tracts in Theoretical Computer Science*, vol. 55. Cambridge University Press, 2003. `http://www.cs.vu.nl/~terese/`.

[136] R. THIEMANN, J. GIESL, et P. SCHNEIDER-KAMP. Improved modular termination proofs using dependency pairs. In *Second International Joint Conference on Automated Reasoning*, coll. *Lecture Notes in Artificial Intelligence*, vol. 3097, pp. 75–90, Cork, Ireland, 2004. Springer.

[137] Y. TOYAMA. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, vol. 25, pp. 141–143, avril 1987.

[138] A. M. TURING. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, vol. 42, n$^o$ 2, pp. 230–265, 1936.

[139] A. M. TURING. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, vol. 43, pp. 544–546, 1937.

[140] A. M. TURING. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculing Machines*, pp. 67–69, Cambridge, 1949. Mathematical Laboratory.

[141] X. URBAIN. Interprétations polynomiales et applications aux $\lambda$-calculs avec substitutions explicites. Rapport de stage de maîtrise, Université Paris-sud, 1996.

[142] X. URBAIN. *Approche incrémentale des preuves automatiques de terminaison*. Thèse de doctorat, Université Paris-Sud, Orsay, France, octobre 2001. `http://www.lri.fr/~urbain/textes/these.ps.gz`.

[143] X. URBAIN. Automated incremental termination proofs for hierarchically defined term rewriting systems. In R. GORÉ, A. LEITSCH, et T. NIPKOW (éd.), *First International Joint Conference on Automated Reasoning*, coll. *Lecture Notes in Artificial Intelligence*, vol. 2083, pp. 485–498, Siena, Italy, juin 2001. Springer.

[144] X. URBAIN. Modular and incremental automated termination proofs. *Journal of Automated Reasoning*, vol. 32, pp. 315–355, 2004.

# Table des figures


2.1 Addition et multiplication des entiers relatifs codés en ternaire . . . . . 19
2.2 Calcul des séquents modulo, partie 1 . . . . . 30
2.3 Calcul des séquents modulo, partie 2 . . . . . 31
2.4 Règles d'inférence pour la réécriture conditionnelle . . . . . 33
2.5 Règle de congruence pour la réécriture context-sensitive . . . . . 34
2.6 Règles d'inférence pour la réécriture avec appartenance . . . . . 38

3.1 Une axiomatique WHY des tableaux fonctionnels . . . . . 47
3.2 Code WHY de la fonction `swap` . . . . . 48
3.3 Code WHY de la fonction `flag` . . . . . 49
3.4 Le drapeau Hollandais de Dijkstra en Java, partie 1 . . . . . 52
3.5 Le drapeau Hollandais de Dijkstra en Java, partie 2 . . . . . 53
3.6 Modèle à la Burstall-Bornat du tas Java . . . . . 54
3.7 Code C annoté de la fonction `memcpy` . . . . . 58
3.8 Spécification de la fonction `strcpy` . . . . . 59
3.9 Code annoté de la fonction `strcpy` . . . . . 59
3.10 Représentation à la Burstall-Bornat du tas mémoire de C . . . . . 60

# Table des matières


**1 Introduction** **3**
- 1.1 Preuve formelle en mathématiques . . . 3
- 1.2 Preuve mécanisée par ordinateur . . . 4
- 1.3 Preuve de programmes . . . 5
- 1.4 La vérification de programmes aujourd'hui . . . 5
- 1.5 Résumé de mes contributions . . . 6
- 1.6 Organisation de ce mémoire . . . 7

**2 Preuves de terminaison** **9**
- 2.1 Contexte . . . 9
  - 2.1.1 Historique . . . 9
  - 2.1.2 La situation en 1997 . . . 11
  - 2.1.3 Résultats obtenus . . . 13
- 2.2 Généralités sur la terminaison et les ordres de réduction . . . 13
  - 2.2.1 Paires d'ordres de réduction . . . 14
  - 2.2.2 Critères des paires de dépendance . . . 15
  - 2.2.3 Critères des graphes de dépendance . . . 16
  - 2.2.4 Symboles associatifs et commutatifs . . . 17
- 2.3 Terminaison par interprétations polynomiales . . . 20
  - 2.3.1 Ordres définis par interprétation et leurs propriétés . . . 20
  - 2.3.2 Test de positivité des polynômes . . . 24
  - 2.3.3 Recherche de polynômes adéquats . . . 26
- 2.4 Critères incrémentaux . . . 27
- 2.5 Terminaison pour d'autres paradigmes . . . 30
  - 2.5.1 Terminaison de la réécriture conditionnelle . . . 32
  - 2.5.2 Terminaison des programmes logiques . . . 35
  - 2.5.3 Terminaison des programmes équationnels avec contraintes d'appartenance . . . 36
- 2.6 Conclusions et perspectives . . . 38

**3 Preuve de programmes impératifs** **41**
- 3.1 Contexte . . . 41
  - 3.1.1 Historique . . . 41

3.1.2 Situation en 2001 . . . 42
3.1.3 Résultats obtenus . . . 42
3.2 La méthode Why . . . 43
3.2.1 Présentation de la méthode Why . . . 43
3.2.2 Les conséquences d'un point de vue « utilisateur» . . . 44
3.2.3 Utilisation concrète de WHY . . . 45
3.2.4 Exemple d'un programme avec un tableau : le drapeau hollandais . 45
3.3 Programmes Java et C : généralités . . . 50
3.3.1 Propriétés typiques recherchées . . . 50
3.3.2 Modélisation fine du tas mémoire . . . 50
3.4 Preuves de programmes JAVA . . . 50
3.4.1 Programmes JAVA annotés en JML . . . 50
3.4.2 Modèle WHY des objets JAVA . . . 51
3.4.3 Traduction de JAVA/JML vers WHY . . . 51
3.4.4 Correction de la méthode . . . 54
3.4.5 Problèmes des invariants de classe et de l'héritage . . . 54
3.5 Programmes C . . . 57
3.5.1 Exemple : memcpy et strcpy . . . 57
3.5.2 Modélisation de l'arithmétique de pointeur . . . 60
3.5.3 Invariants de structure . . . 62
3.6 Études de cas . . . 63
3.6.1 Programmes JavaCard . . . 63
3.6.2 Pointeurs et aliasing . . . 64
3.6.3 Autres Algorithmes . . . 64
3.6.4 Programmes C embarqués . . . 64
3.7 Conclusions et perspectives . . . 64

**4 Conclusions et perspectives** **67**
4.1 Perspectives sur les preuves de terminaison . . . 67
4.2 Perspectives sur la preuve de programmes impératifs . . . 68
4.3 Et à long terme ? . . . 69

**Bibliographie** **71**

**Table des figures** **81**