Facettes de la preuve

Jeux de reflets
entre
démonstration automatique
et
preuve assistée

Évelyne Contejean

Chapitre 1

Liminaire

1.1 Comment la rhétorique s'est faite machine

Depuis l'antiquité les philosophes ont cherché à convaincre les autres hommes de la justesse de leurs idées ; dans le domaine du droit, comme dans celui des mathématiques, la notion de preuve est centrale, et ce n'est pas un hasard si le même mot, la même notion, apparaît dans deux champs de la connaissance qui semblent assez éloignés. Il s'agit de persuader, en ne laissant aucune place au doute. Dans le domaine des mathématiques, ce souci a conduit, dès Euclide, à préciser les objets du discours, les hypothèses que l'on fait sur eux, puis les propriétés que l'on peut en déduire, d'où la notion de démonstration.

La question de savoir ce qu'est une « bonne » démonstration est devenue d'une actualité brûlante à la fin du XIXe siècle : les mathématiciens ont cherché à dégager du bon sens commun les principes du raisonnement et à formaliser l'idée de preuve.

À partir de 1874, Georg Cantor travaille à sa théorie des ensembles et montre qu'il existe plusieurs sortes – en vérité, une infinité – d'infinis. Cette théorie unificatrice est encore aujourd'hui la base du travail des mathématiciens.

En 1879, « tout à coup [...] la logique des propositions sort dans un état d'achèvement quasi-absolu du cerveau génial de Gottlob Frege, le plus grand logicien de notre temps. » (Jan Lukasiewicz [1]).

Une quinzaine d'années plus tard, en 1893, Heinrich Weber axiomatise la notion de groupe abstrait, généralisant par là divers objets issus de la théorie de Galois.

Malheureusement, cet effort de rigueur semble mis à bas par la découverte de paradoxes. Le premier chronologiquement est celui de Cesare Burali-Forti et date de 1897. Il énonce que, comme on peut définir la borne supérieure d'un ensemble d'ordinaux, si l'ensemble de tous les ordinaux existe, on peut définir un ordinal supérieur strictement à tous les ordinaux, donc à lui-même, d'où une contradiction. Le paradoxe de Bertrand Russell (1903) est plus aisé à expliquer aux non mathématiciens : l'ensemble des ensembles qui ne s'appartiennent pas $\{x \mid x \notin x\}$ devrait à la fois appartenir et ne pas appartenir à lui-même. L'analogue profane est : « Si dans une ville, le barbier rase tous les hommes qui ne se rasent pas eux-mêmes, est-ce que le barbier se rase, ou est-ce qu'il ne se rase pas ? ». En fait, avec cette définition du barbier, celui-ci ne peut pas se raser, et ne peut pas ne pas se raser. . .

1. traduit par Jean Largeault [82].

Toutefois, la crise des fondements a été extrêmement féconde, puisqu'elle a donné naissance à plusieurs formalismes visant à fonder solidement le raisonnement mathématique en bloquant les paradoxes connus :

- Ernst Zermelo a donné une version révisée de la théorie naïve des ensembles qui ne permet plus d'écrire $x \in x$;
- Bertrand Russell et Alfred Whitehead ont proposé dans leurs Principia Mathematica [114] d'utiliser une hiérarchie sur les objets : la théorie des types simples.

Cette crise a ainsi nourri le programme de Hilbert qui visait à formaliser et à complètement mécaniser le raisonnement mathématique. Cette ambition se décèle dès 1900, quand dans sa fameuse liste de 23 problèmes pour les mathématiciens du XXe siècle, David Hilbert inclut la consistance de l'arithmétique – deuxième problème – et l'existence d'un algorithme pour résoudre les équations diophantiennes – dixième problème. Si certains des 23 problèmes ont reçu des réponses positives, pour le deuxième et le dixième, des réponses négatives ont été données respectivement par Kurt Gödel avec ses deux théorèmes d'incomplétude (1931) [64] et par Yuri Matijasevic (1970) [89].

La question de la mécanisation a de nouveau été agitée par Alan Turing en 1936, avec la description de machines – théoriques – très simples. Ces machines visent à définir précisément ce qu'est une méthode effectivement calculable dans le cadre du problème de la décision dans les théories axiomatiques. Une nouvelle fois, le programme de Hilbert est en échec, car si les machines de Turing ont eu une longue postérité scientifique, il n'existe pas de méthode algorithmique pour savoir si elles s'arrêtent [106, 107].

Encore jeune étudiante en troisième année de mathématiques, j'avais été choquée d'entendre dire par l'une de mes professeures qu'une preuve mathématique était ultimement considérée comme vraie quand elle était reconnue comme telle par l'ensemble des mathématiciens. Les mathématiques ne seraient qu'une construction humaine, sujettes à toutes les défiances, à toutes les erreurs, et ne reposeraient que sur un fragile consensus, parfois remis en question par la découverte de paradoxes.

De là date, je pense, mon goût pour les fondements des mathématiques, la logique, et les preuves mécanisées, sur papier et, depuis le développement de leur puissance de calcul, sur ordinateurs.

La crise de la fin du XIXe siècle a causé un traumatisme dans la communauté mathématique et, pour certains pratiquants contemporains des mathématiques, mieux vaut oublier tout cela : « La crise des fondements a été résolue par Russell et Whitehead, pourquoi travailler encore sur la logique ? » m'a dit en 2007 un membre du jury de l'agrégation de mathématiques, quand j'ai expliqué que mon domaine de recherche était la démonstration automatique, lointaine parente de la logique mathématique.

Ces questionnements m'ont incitée à lire l'introduction du premier livre de Bourbaki [17] sur la théorie des ensembles (1939) :

« En principe, la vérification d'un texte formalisé est chose aisée, puisqu'elle ne demande qu'une attention en quelque sorte mécanique ; [...] du moins n'est-on pas exposé, dans un calcul formalisé, aux fautes de raisonnement que risquent trop souvent d'entraîner, dans un texte mathématique écrit en langage courant, l'usage abusif de l'intuition, l'association d'idées, et les autres sources des pétitions de principes auxquelles est exposé le mathématicien. »

Bien sûr, l'utilisation d'un langage formel est un idéal fort peu pratique à manipuler

par un être humain et, une fois que l'on en est convenu, « l'emploi de toutes les ressources de la rhétorique devient [...] légitime, pourvu bien entendu que l'ossature logique du texte demeure intacte. »

Cependant, depuis les théorèmes d'incomplétude de Gödel, la question de la cohérence du tout reste un horizon inaccessible et repose sur une conviction intime :

« Si nous croyons que notre mathématique est destinée à survivre, qu'on ne verra jamais les parties essentielles de ce majestueux édifice s'écrouler du fait d'une contradiction soudain manifestée, nous ne prétendons pas que cette opinion repose sur autre chose que l'expérience renforcée par le raisonnement. C'est peu, diront certains. Mais voilà vingt-cinq siècles que les mathématiciens ont l'habitude de corriger leurs erreurs et d'en voir leur science enrichie, non appauvrie ; cela leur donne le droit d'envisager l'avenir avec sérénité. »

Et de fait, aucun nouveau paradoxe n'est venu pour troubler cette quiétude – pour l'instant du moins.

La question qui se pose maintenant est : pourquoi s'intéresser encore à cette vieille histoire ? Simplement parce que, comme lors de la recherche du coupable dans un roman policier, nous avons l'*occasion* et le *mobile* pour faire des preuves mécanisées : l'occasion, grâce à l'invention et au développement des ordinateurs, et le mobile, parce que certains programmes sensibles doivent être vérifiés et que cette vérification revient *in fine* à prouver des théorèmes mathématiques.

1.2 Preuves mécanisées

Le programme de Hilbert a été mis en échec à plusieurs reprises et il est aujourd'hui clair qu'il est totalement illusoire d'espérer construire une machine qui répondrait « vrai » ou « faux » à tout énoncé mathématique. Cependant le rêve de Hilbert peut devenir partiellement réalité :

– D'une part, comme le note Bourbaki (*confer supra*), la *vérification* d'une preuve est possible, à condition que toutes ses étapes soient suffisamment détaillées ;
– D'autre part, si l'arithmétique – et *a fortiori* l'ensemble des mathématiques – est indécidable, pour *certains* énoncés simples, appartenant à des fragments restreints, la recherche d'une preuve est complètement automatisable.

Ces deux remarques donnent lieu à deux approches, qui peuvent parfois se compléter au sein d'une même preuve.

1.2.1 Preuve assistée

Principes

Un préalable à l'écriture d'un programme pour vérifier des preuves sur un ordinateur est la conception d'un langage de description de ces preuves, c'est-à-dire en vérité d'une logique. Cette dualité a déjà été explicitée par Gottfried Leibniz au XVIIe siècle : la langue formelle et universelle dont il a rêvé, la *characteristica universalis* devait être traitée par un calcul, le *calculus ratiocinator*, qui aurait pu déterminer la véracité de toutes les phrases écrites dans cette langue. L'ambition de Leibniz ne se bornait pas aux mathématiques, il visait à formaliser l'ensemble de la pensée, y compris la philosophie.

Depuis le XXe siècle, la plupart des logiques des systèmes existants sont des logiques d'ordre supérieur fondées sur diverses variantes du λ-calcul typé. Certaines font usage de la correspondance de Curry-Howard, qui permet d'identifier une formule – un théorème que l'on souhaite prouver – à un type, et une preuve à un terme de ce type. Ces logiques sont très puissantes et permettent d'exprimer tous les théorèmes des mathématiciens – à défaut de les prouver.

Précurseurs

Les premiers systèmes dans le domaine de la vérification de preuves formelles sont communément datés par le moment de la conception de leur langage de preuves mais, souvent, l'implantation n'a été réalisée que plusieurs années plus tard.

Les débuts du projet AUTOMATH (AUTOmatisation des MATHématiques) [91] remontent à 1967, quand le mathématicien Nicolaas Govert de Bruijn a décidé de concevoir un langage formel assez puissant pour exprimer toutes les mathématiques et assez détaillé pour que les preuves puissent être vérifiées par un ordinateur. Ce projet a été mené à bien : un vérificateur de preuves a effectivement été programmé et utilisé sur des portions significatives de mathématiques par les étudiants de de Bruijn. Cependant, bien qu'ayant introduit des concepts novateurs pour l'époque et influencé largement ses successeurs, AUTOMATH n'a pas vraiment connu de succès dans les années 1970 et est resté assez peu utilisé. La principale raison en est, à mon avis, que ce système laisse toute la charge de l'écriture infiniment détaillée de la preuve à son utilisateur et ne traite que de la phase de vérification.

Les systèmes suivants seront des *assistants* à la preuve car, en plus de la vérification proprement dite, ils aident l'utilisateur à construire sa preuve grâce à des *tactiques* qui transforment le théorème à prouver – le but initial – en sous-buts. Par exemple, si le théorème à prouver est

$$\forall x : \mathbb{N}, pair(x) \vee impair(x)$$

et que l'utilisateur applique la tactique de preuve par récurrence, le système lui demandera de démontrer les deux sous-buts :

$$pair(0) \vee impair(0)$$
$$\forall x : \mathbb{N}, (pair(x) \vee impair(x)) \Rightarrow (pair(x+1) \vee impair(x+1))$$

Ces sous-buts peuvent à leur tour être prouvés en utilisant des tactiques. L'utilisateur n'est pas véritablement guidé, car c'est lui qui fixe l'ossature logique de la preuve, mais il voit la preuve se développer sous ses yeux et ne travaille pas à l'aveugle.

Le premier système à pouvoir véritablement être qualifié d'assistant est LCF (Logic for Computable Functions) réalisé en 1972 par Robin Milner et fondé sur un langage logique proposé par Dana Scott dans un manuscrit de 1969 [103]. Pour permettre à ses utilisateurs de programmer des tactiques, Milner a introduit le métalangage ML, dont les différents dialectes (Standard ML et Caml) auront une vie propre comme langages de programmation fonctionnels. LCF a connu plusieurs déclinaisons, Stanford LCF, Edimburg LCF, Cambridge LCF, au gré des déplacements géographiques de ses développeurs.

Le cas du prouveur Nqthm de Robert S. Boyer et J Strother Moore est un peu particulier puisqu'il a été conçu à l'origine en 1971 comme un démonstrateur entièrement

automatique [18] fondé sur la logique dite de Lisp, à savoir du premier ordre sans quantificateurs, avec égalité et un principe d'induction sur les entiers naturels. Cependant en 1974, l'utilisation des égalités déjà prouvées dans son mécanisme de simplification l'a rendu très instable, l'entraînant souvent dans des boucles de réécriture infinies ; l'interaction avec l'utilisateur est devenue nécessaire, le faisant ainsi basculer du côté des assistants à la preuve. Néanmoins, cette interaction se limite à proposer des lemmes et à choisir dans quel ordre le faire.

Systèmes actuels

Avec le développement de la puissance des ordinateurs et le perfectionnement des interfaces, une large palette d'assistants à la preuve est aujourd'hui disponible pour les utilisateurs : ACL (A Computational Logic), Agda, Alf (Another Logical Framework), Coq, HOL, Isabelle, Lego, LF (Logical Framework), NuPrl, PVS (Prototype Verification System), etc.

Ces systèmes sont de plus en plus utilisés, notamment pour développer de larges pans des mathématiques, comme le théorème fondamental de l'algèbre dans la bibliothèque CoRN (Constructive Coq Repository at Nijmegen), le théorème des quatre couleurs, la classification des groupes finis (théorème de Feit-Thompson) par le groupe de Georges Gonthier, mais surtout pour formaliser et vérifier d'autres systèmes, par exemple un compilateur pour le langage C avec le projet CompCert de Xavier Leroy, un micro noyau de système d'exploitation avec le projet seL4 au NICTA, et une version légère du langage de programmation Java (Lightweight Java) grâce à l'outil Ott développé à Cambdridge (UK) et INRIA.

1.2.2 Démonstration automatique

Le parti pris de la démonstration automatique est radicalement différent de celui de la preuve assistée. Comme son nom l'indique, les preuves sont non seulement mécanisées mais aussi sans aucune intervention extérieure. En pratique, cela signifie que le système se charge de la *recherche* de la preuve – plus rarement de sa vérification. La contrepartie est que les logiques sous-jacentes sont beaucoup plus simples que celles des preuves assistées, décidables ou semi-décidables.

Un exemple paradigmatique est la logique dite de Presburger, ainsi nommée d'après Mojzesz Presburger qui a démontré en 1929 que la théorie des nombres naturels avec addition et égalité est décidable en donnant un algorithme déterminant, pour toute formule logique du premier ordre dans ce langage, si elle est valide ou non. Dans son chapitre sur l'histoire des débuts, dans le précis de démonstration automatique, Martin Davis [47] rapporte qu'il a implanté cet algorithme en 1954 sur l'ordinateur à tubes à vide Johnniac de l'institut pour les études avancées (Institute for Advanced Study) à Princeton, avec un succès mitigé : « Puisqu'il est maintenant connu que la procédure de Presburger a une complexité pire qu'exponentielle, il n'est pas surprenant que le programme n'ait pas très bien fonctionné. Son plus grand triomphe a été de prouver que la somme de deux nombres pairs est paire. »

En 1957, Allen Newell, John Shaw et Herbert Simon [93] se sont quant à eux appuyés

sur le premier chapitre des « Principia » pour implanter, également sur un Johnniac, LTM, leur « Logic Theory Machine », une procédure de décision pour le calcul propositionnel. LTM a ainsi démontré 38 des 52 premiers théorèmes des « Principia », en utilisant une stratégie heuristique, essayant d'imiter le raisonnement humain, c'est-à-dire en vérité incomplète. L'influence de LTM se mesure à l'aulne de la diffusion des techniques qui y ont été utilisées pour la première fois, telles que les chaînages avant et arrière, la génération de lemmes utiles et la recherche de substitutions appariant des formules.

À la même époque, Herbert Gelernter a implanté la « Geometry Machine » pour la géométrie élémentaire du plan qui, bien que fondée sur une procédure de décision théoriquement complète, a également une stratégie heuristique. Parmi les innovations introduites, il est à noter l'utilisation des symétries pour réduire la taille des preuves.

La démonstration automatique vise à mettre en œuvre un raisonnement systématique pour démontrer des théorèmes. Les théories décidables se prêtent bien à ce jeu mais elles ont un pouvoir d'expression limité. La logique du premier ordre est de ce point de vue plus séduisante, puisqu'en se donnant des axiomes raisonnables en hypothèses, on peut y exprimer tout le raisonnement mathématique. Malheureusement, il n'existe pas de procédure de décision générale pour cette logique. Cependant, en 1957, le logicien Abraham Robinson a proposé d'utiliser les fonctions de Skolem et le théorème de Herbrand comme fondements pour implanter des démonstrateurs automatiques généralistes. Ces idées ont été reprises par Paul Gilmore [63] en 1960, et par Dag Prawitz, Haåkan Prawitz et Neri Voghera à la même période [97]. Ces machines exploraient l'univers de Herbrand sans aucun guidage, elles ne pouvaient donc prouver que des théorèmes très simples dans un temps raisonnable.

Martin Davis et Hilary Putnam [49] ont ensuite proposé une stratégie d'exploration de l'univers de Herbrand pour guider les démonstrateurs et, avec George Logemann et Donald Loveland, ils l'ont affinée quelques temps après, donnant lieu à la procédure connue aujourd'hui sous le nom de DPLL [48].

En 1965, John A. Robinson proposa une unique règle d'inférence, complète pour la logique du premier ordre, la *résolution* [101], qu'il étendit en « hyper-résolution » (aujourd'hui paramodulation) pour traiter l'égalité de façon prédéfinie [100]. Cette approche a eu une très large postérité, et à donné naissance à toute la famille des démonstrateurs de la famille dite « TPTP » – Thousands of Problems for Theorem Provers [104], ainsi qu'à la compétition CASC associée.

En 1970, Donald Knuth et Peter Bendix ont proposé une procédure, la *complétion* [78], pour traiter efficacement le problème de savoir si une équation (universellement quantifiée) est la conséquence d'un ensemble d'autres équations, en utilisant un ordre bien fondé pour guider l'orientation des équations en règles de réécriture. Dix ans plus tard, Gérard Huet a montré que la complétion fournit une procédure de semi-décision [70].

Dans le cas clos, le problème est décidable, grâce à l'algorithme de clôture par congruence, proposé indépendamment par Peter Downey, Ravi Sethi et Robert Tarjan [55] et Greg Nelson et Derek Oppen [92]. Du côté décidable des logiques, afin de combiner les procédures de décision, il existe deux techniques, celle due encore une fois à Greg Nelson et Derek Oppen, qui fonctionne par partage des égalités déduites par les différentes procédures, et celle de Shostak, fondée sur des théories avec solveur, pour résoudre les équations, et canoniseur, pour simplifier et réduire les termes égaux à la même forme. Ces techniques sont le fondement d'une autre classe de démonstrateurs automatiques, les prou-

veurs « SMT » – satisfaisabilité modulo théories – qui ont eux aussi leur compétition dédiée et un format d'entrée partagé, SMT-LIB.

1.3 Organisation de ce mémoire

Pour décrire de façon synthétique l'ensemble des travaux que j'ai effectués depuis ma thèse, j'ai choisi de les rassembler tous sous le titre de « Facettes de la preuve » qui, non seulement, est fédérateur mais vise également à indiquer que j'ai envisagé dans ce domaine aussi bien les aspects théoriques que pratiques, automatiques, qu'interactifs.

Plus concrètement, j'ai décidé de centrer ce document sur mes travaux non publiés les plus récents, tout en survolant dans le chapitre 2 qui suit mes travaux plus anciens et/ou déjà publiés. Le chapitre 3 est ainsi consacré à la formalisation dans l'assistant à la preuve Coq des algèbres universelles, ainsi que de certaines propriétés de la réécriture de termes au premier ordre. Ce document se termine par les perspectives ouvertes par mes travaux à moyen terme.

Chapitre 2

Survol de mes contributions récentes

J'effectue ma recherche au sein de Vals-Toccata, une équipe du LRI qui est commune au CNRS, à INRIA et à l'université Paris-Sud. Les objectifs scientifiques de ce groupe sont de promouvoir et développer des méthodes et des outils qui peuvent être intégrés dans le cycle de développement du logiciel et rendent possible la production d'un code prouvé conforme à sa spécification. La construction de cette preuve de conformité est mécanisée, autant que faire se peut complètement automatique, et/ou formellement développée dans un assistant à la preuve.

Une de nos spécificités est une importante production de logiciels, qui permettent de valider l'approche choisie et de mener à bien des études de cas réalistes, et pas seulement de résoudre des exemples jouets – en témoignent d'ailleurs nombre de collaborations avec des partenaires industriels dans des domaines critiques.

Forts de notre expertise en langages et techniques de preuve avancées, nous attaquons plusieurs verrous technologiques. Le premier est la collaboration intelligente de preuves automatiques et interactives ; le second consiste à fournir des langages, méthodes et outils adaptés à la spécification des programmes, phase aussi importante et complexe que la phase de preuve.

Notre approche repose sur une succession de transformations des programmes schématisée par la figure 2.1. Les programmes peuvent être écrits dans différents langages, pour l'instant essentiellement des langages séquentiels « mainstream », tels que C ou Java, et sont spécifiés par des annotations qui décrivent leur comportement attendu, par exemple que le résultat d'un programme de tri est une permutation triée du tableau de départ.

Un programme annoté est transformé en une formule logique comme suit :

Φ1. Dans une première phase, le programme est compilé dans le langage intermédiaire de l'outil Why, WhyML ;

Φ2. Puis par un mécanisme de calcul de précondition qui prend en compte la sémantique et la gestion de la mémoire du langage de programmation initial, le programme est retranscrit dans une formule logique dont la validité est équivalente au fait qu'il satisfait sa spécification ;

Φ3. Enfin, comme la logique est relativement riche et parle notamment des théories suivantes :
   – Égalité (c'est-à-dire congruence) ;

– Arithmétique sur les rationnels, les flottants, les entiers (avec principe d'induction) ;
– Atructures de données, éventuellement polymorphes, en particulier pour modéliser la mémoire,

certaines parties sont axiomatisées et/ou traduites afin de rentrer dans le cadre de différents prouveurs, soit automatiques, soit interactifs.

Les formules logiques produites pendant la phase Φ2 peuvent être découpées afin de subir un traitement différencié lors de la phase Φ3, de façon à pouvoir utiliser les meilleurs démonstrateurs pour chaque sous-formule.

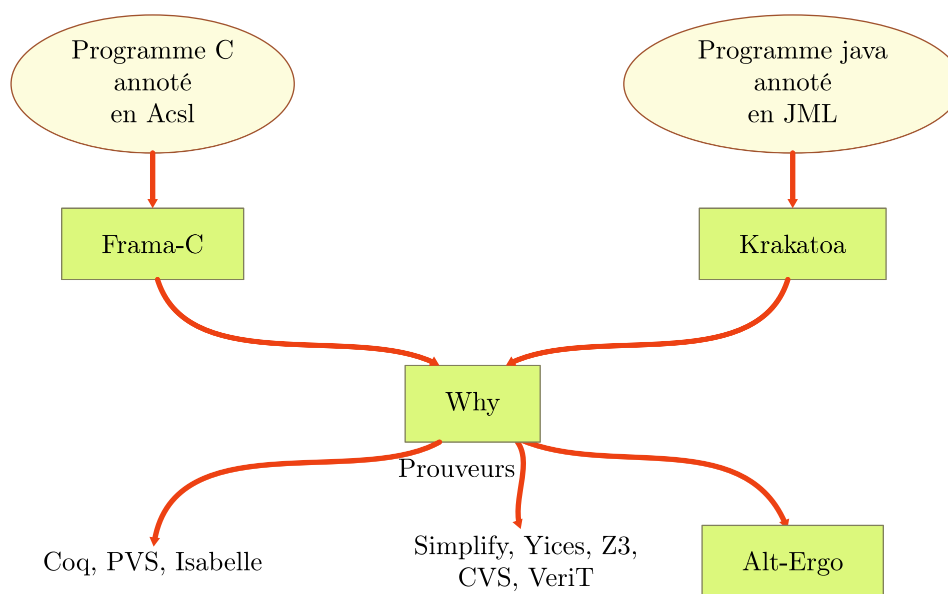


FIGURE 2.1 – Approche générale de Toccata pour la vérification de programmes.

Depuis plusieurs années, une partie de mes travaux (sections 2.1 et 2.2) se situent dans « les couches basses » du diagramme, à l'articulation des preuves automatiques et interactives ; depuis quelques mois, je vise également à appliquer l'ensemble de l'approche pour prouver des programmes écrits dans d'autres types de langages, les langages centrés sur les données (section 2.3).

Toccata, en tant qu'équipe INRIA, amorce une inflexion thématique, visant à produire des preuves de programme *très hautement* certifiées, en utilisant des méthodes et des outils eux-mêmes prouvés sûrs. Cela se traduira par une application réflexive de notre approche à elle-même. En particulier, il s'agira de proposer un démonstrateur automatique certifié ; un pas dans cette direction a d'ailleurs été fait avec la thèse de Stéphane Lescuyer [85].

2.1 Procédures de décision

Je collabore depuis 2006 avec Sylvain Conchon, également membre de Toccata. Nous avons choisi de nous intéresser à la combinaison de procédures de décision dans un cadre

polymorphe, car même si elle a évolué, l'approche de la preuve de programmes au sein de Toccata a toujours suivi le schéma de la figure 2.1 et, en 2006, à notre connaissance, aucun démonstrateur automatique ne pouvait s'attaquer directement aux formules produites après la phase $\Phi 2$ de la chaîne de transformations.

En effet, les propriétés à prouver sont données sous la forme de formules du premier ordre dans une logique multi-sortée polymorphe qui contient de façon native certaines théories provenant des types de données manipulés par les programmes, tels que les entiers, les tableaux, *etc*. Le polymorphisme apparaît naturellement pour modéliser la mémoire de la machine sur laquelle s'exécute le programme à vérifier, de façon à décrire de façon générique et uniforme l'allocation et l'accès en lecture et écriture pour ces données et à instancier dans un second temps cette description par le type des objets effectivement utilisés.

En outre, ces formules ont des formes particulières : elles se présentent comme $M \wedge H_1 \wedge \ldots \wedge H_n \rightarrow C$, où M est un ensemble d'hypothèses polymorphes décrivant le modèle mémoire utilisé, $H_1 \wedge \ldots \wedge H_n$ sont des hypothèses qui dépendent du programme à vérifier et C est la conjecture à démontrer. M est très gros, certaines des hypothèses $H_1 \wedge \ldots \wedge H_n$ sont inutiles, et C est en général assez simple.

Parmi les démonstrateurs automatiques disponibles à l'époque, et testés concurremment dans notre équipe – Simplify, Yices, Z3, Cvc-lite, Harvey, Rv-sat et Zenon –, aucun ne répondait parfaitement au cahier des charges suivant : la vérification de programme demande un démonstrateur

- Dans une logique multi-sortée polymorphe ;
- Qui intègre des procédures de décision, en particulier pour l'égalité et l'arithmétique linéaire ;
- Qui a des stratégies de recherche modulables en fonction des types de propriétés à vérifier ;
- Qui a un temps de réponse rapide (dans l'idéal, les programmes devraient être vérifiés automatiquement durant leur phase de compilation) ;
- Et enfin qui produit des résultats vérifiables par un assistant de preuves, par exemple Coq. En effet, vérifier un programme revient à augmenter la confiance que l'on a dans sa correction mais la frontière ne fait que se déplacer, il faut alors faire confiance au démonstrateur, ou bien se donner des garde-fous comme la production de traces.

Certains outils possèdent une ou plusieurs de ces caractéristiques mais aucun ne travaille avec une logique polymorphe.

Avec Sylvain Conchon, nous avons conçu le démonstrateur Alt-Ergo [12] de façon à répondre au cahier des charges décrit ci-dessus. Son architecture est schématisée à la figure 2.2 et le place dans la famille des démonstrateurs SMT.

Alt-Ergo autorise deux syntaxes d'entrée et, après une phase de typage, les formules sont découpées en une conjonction d'axiomes (universellement quantifiés) et un but (clos). La boucle principale explore la structure propositionnelle du but grâce à un SAT-solveur, et appelle ensuite le « cœur » du prouveur, c'est-à-dire la procédure de décision principale, pour résoudre les sous-buts. Si le but n'a pas pu être démontré, on passe alors à une phase d'instanciation des axiomes par les termes clos connus avant d'itérer le processus.

Notre but était alors, et est toujours, d'intégrer dans nos procédures de décision le plus possible des caractéristiques de la logique en sortie de la phase $\Phi 2$, afin de minimiser la phase $\Phi 3$. Notre préoccupation essentielle est d'éviter les erreurs et d'améliorer l'efficacité.

En effet un traitement prédéfini est en général plus rapide et plus puissant qu'un traitement générique à base de codages.

Les résultats théoriques que nous avons obtenus sont intégrés dans Alt-Ergo : la modularité de l'architecture nous permet d'expérimenter aisément et rapidement nos idées et d'affiner nos nouvelles procédures de décision en les (dé-)branchant à la place du cœur.

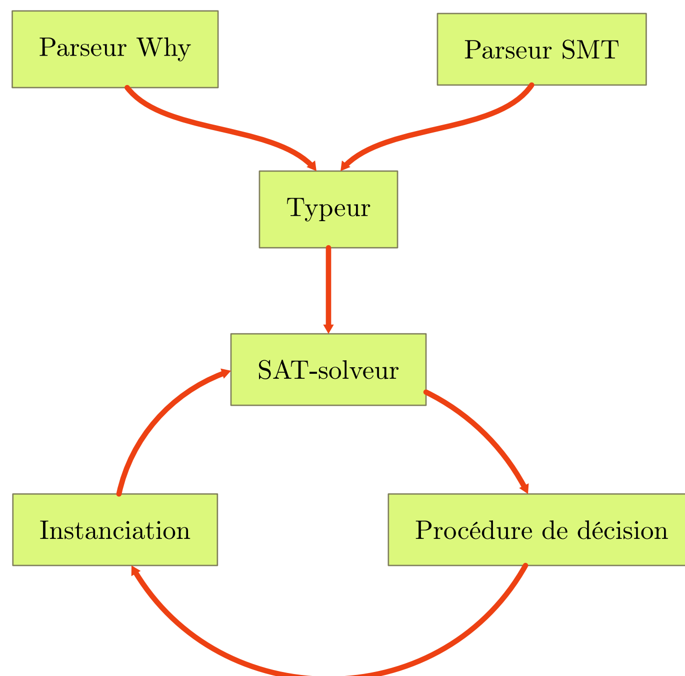


FIGURE 2.2 – L'architecture d'Alt-Ergo.

Nous avons co-encadré plusieurs stages de niveau M1/M2, et plus récemment deux thèses, celle de Stéphane Lescuyer [85] et celle de Mohamed Iguernelala [73].

Stéphane Lescuyer a travaillé sur le polymorphisme et la formalisation en Coq du cœur d'Alt-Ergo. Avec Mohamed Iguernelala, nous avons tenté d'améliorer la puissance et l'efficacité de nos procédures de décision essentiellement sur deux axes, les symboles associatifs-commutatifs et l'arithmétique.

2.1.1 Polymorphisme

En 2006, comme la plupart des démonstrateurs utilisés par la chaîne Why3 n'avaient pas une logique polymorphe, il fallait passer par une phase d'élimination. Trois possibilités étaient offertes : l'oubli pur et simple ; la « monomorphisation », c'est-à-dire une instanciation bornée des variables de types ; enfin un codage des sortes par des prédicats posés comme garde dans toutes les formules logiques.

Ces trois choix étaient également malheureux. L'oubli des sortes est notoirement incorrect, comme le montre la formule suivante

$$\forall x : \texttt{unit}, x = \texttt{tt}$$

qui en devenant $\forall x, x = \mathtt{tt}$, permet de prouver $0 = 1$ par transitivité avec tt !

La question de la complétude théorique de la technique de monomorphisation était ouverte mais l'implantation au sein de Why était bel et bien incomplète. Depuis Andrei Paskevich et François Bobot [15] ont répondu, de manière négative, à cette question.

Enfin, la technique alternative par codage des sortes par des prédicats, en changeant la structure des formules logiques, perturbait le mécanisme d'instanciation des démonstrateurs SMT, fondé sur un calcul de motifs déclencheurs – « triggers ».

Dans un premier temps, j'ai proposé une piste pour utiliser un autre codage avec des symboles de fonction, ce qui permet en outre d'éviter la phase de monomorphisation. Par exemple, la formule

$$\forall a : int,\ l : list\ int,\ \mathtt{head}(\mathtt{cons}(a, l)) = a$$

devient

$$\forall a,\ l,\ (sort(int, \mathtt{head}(sort(list(int), \mathtt{cons}(sort(int, a), sort(list(int), l))))) = sort(int, a)$$

Les sortes, comme $list(int)$, sont vues comme des termes, tandis que $sort$ est un nouveau symbole de fonction binaire dont le premier argument est la sorte de son deuxième argument.

Le polymorphisme est alors traité simplement, les variables de sortes deviennent des variables comme les autres. La formule

$$\forall a : \alpha,\ l : list\ \alpha,\ \mathtt{head}(\mathtt{cons}(a, l)) = a$$

devient

$$\forall \alpha,\ a,\ l,\ (sort(\alpha, \mathtt{head}(sort(list(\alpha), \mathtt{cons}(sort(\alpha, a), sort(list(\alpha), l))))) = sort(\alpha, a)$$

Un stage de niveau M2 a été proposé au MPRI et Stéphane Lescuyer a été co-encadré sur ce sujet par Sylvain Conchon et moi-même de mars à août 2006. Il a démontré que ce codage conservait la validité des formules en utilisant des techniques de la théorie des domaines et, de façon surprenante, il a été amené à introduire une logique trivaluée pour prendre en compte les termes mal formés, tels que $sort(list(\alpha), 3)$. Il a en outre implanté ce codage dans l'outil Why. Ce travail réalisé en 6 mois répondait à un besoin immédiat, les codages ou les oublis utilisés précédemment n'étant pas satisfaisants (incorrection, monomorphisation qui ne termine pas, etc.). Stéphane Lescuyer a approfondi ce travail avec Jean-François Couchot et ils ont publié un article à la conférence CADE en 2007 [44]. Finalement, Andrei Paskevich et François Bobot ont raffiné ce codage pour ne pas encoder certaines sortes prédéfinies – les sortes natives des démonstrateurs – de façon à ce que ces derniers puissent utiliser leurs procédures de décision associées [15].

Dans une perspective à plus long terme, il vaut mieux traiter le polymorphisme directement, sans passer par un codage. Nous avons plaidé dans ce sens auprès de la communauté SMT [14]. Notre principal argument est que l'ajout du polymorphisme n'entraîne que des modifications légères dans un solveur SMT, comme le montre le développement d'Alt-Ergo [20]. Nous avons partiellement été entendus, car la deuxième version du langage commun des démonstrateurs SMT, SMT-LIB2, comporte des théories polymorphes, même si la logique associée reste multi-sortée.

2.1.2 Combinaison

Le cœur d'Alt-Ergo est le module CC(X) [24, 26], clôture par congruence modulo une théorie X à la Shostak (avec canoniseur et solveur). L'algorithme implanté dans Alt-Ergo est décrit par un petit système de règles d'inférence à la fois proche du code OCaml et dont les preuves de correction et complétude sont plus faciles à prouver que sur du pseudo-code impératif. Nous donnons des conditions suffisantes à la correction et à la complétude. Ces conditions sont remplies par un certain nombre de théories utiles en pratique comme l'arithmétique linéaire, la théorie des vecteurs de bits et la théorie des constructeurs.

Nous avons étendu la version de base de CC(X) en AC(X) qui traite également de façon native les opérateurs qui vérifient les propriétés d'associativité et de commutativité (AC). Ces opérateurs sont nombreux en mathématiques et également dans les formules logiques produites par la vérification de programmes. Il est bien connu en démonstration automatique qu'un traitement intégré de ces propriétés (*cfr* réécriture/complétion modulo AC) est bien plus rapide et efficace qu'un traitement générique, où l'on considère comme des axiomes à instancier les formules

$$\begin{array}{rcl} \forall x, y, z. & f(f(x,y),z) = f(x, f(y,z)) & (A) \\ \forall x, y. & f(x,y) = f(y,x) & (C) \end{array}$$

Nous avons proposé un traitement prédéfini des propriétés AC en combinaison avec une théorie de Shostak fondé sur la complétion AC close, où la règle usuelle d'orientation des équations en règles de réécriture est remplacée par l'appel au solveur de la théorie de Shostak, et où la normalisation intègre la réécriture AC usuelle et le canoniseur de Shostak [21]. Cette approche est correcte, complète, et termine dans le cas clos [22, 23] et nous avons également montré qu'elle est plus efficace qu'un traitement axiomatique, à la fois en comparant les différentes versions de notre prouveur entre elles, mais aussi avec les prouveurs SMT à la pointe de la technologie. En outre nous avons proposé [23] une implantation efficace sans recours à des ordres compatibles AC compliqués. Ce travail, à la frontière des domaines de la démonstration automatique « à la TPTP » et des procédures de décision modulo théories a été apprécié par la communauté et nous avons reçu le « Best ETAPS theoretical paper award » de l'EATCS pour l'article [22] présenté à TACAS en 2011.

2.1.3 Vérification formelle

La thèse de Stéphane Lescuyer portait sur la formalisation en Coq de CC(X)et sa déclinaison en une tactique pour Coq. Nous avons ainsi un prouveur dans lequel on peut placer une confiance raisonnable.

Concernant la tactique pour Coq, Denis Cousineau, au cours d'un post-doc supervisé par Sylvain Conchon, l'a étendue et adaptée pour traiter également les prédicats $\leq$ et $<$ de l'arithmétique et utiliser Alt-Ergo comme un oracle afin d'instancier les quantificateurs universels des axiomes.

Ces travaux ont été réalisés dans le cadre du projet ANR Décert (procédures de DÉcision CERTifiées) dont le but était d'utiliser les résultats de démonstrateurs automatiques de la famille SMT dans des preuves formelles réalisées dans l'assistant Coq. Le projet s'appuyait sur deux approches :

- La formalisation d'un noyau SAT/SMT pour des preuves par réflexion entièrement réalisées par calcul au sein de Coq ;
- L'utilisation de traces produites par des prouveurs SMT externes sur chaque instance de problèmes, pour produire un certificat de correction, et l'interprétation de ces certificats dans Coq.

Du côté de Proval, nous avons travaillé autour d'Alt-Ergo, en suivant les deux approches. Les traces [25] nous ont par ailleurs été utiles pour améliorer l'efficacité du SAT-solveur, via le mécanisme classique de « back-jumping ». Les équipes INRIA Marelle et Typical [5, 6, 4] ont quant à elles étudié l'interprétation des traces du prouveur Z3, développé par Microsoft Research [50]. Les deux tactiques obtenues ont chacune leurs points forts et leurs points faibles. Marelle et Typical ont réussi à interpréter de façon remarquablement efficace les traces de Z3, y compris celles de très grande taille, obtenues à partir d'exemples engendrés automatiquement. Cependant le fait de ne pas avoir la main sur le prouveur SMT Z3 rend leur tactique tributaire des traces produites et donc des théories tracées par Z3. Comme l'optique de Proval en démonstration automatique est très influencée par la vérification déductive de programmes, où les théories sont riches, le traitement des quantificateurs essentiel, et où si la *recherche* des preuves peut-être longue, les preuves elles-mêmes sont en général ramassées, nous avons au contraire concentré nos efforts sur l'expressivité de la logique, et non pas sur l'efficacité.

2.1.4 Arithmétique

L'arithmétique est cruciale dans un solveur SMT dédié à la preuve de programme, car outre le fait que les programmes eux-mêmes manipulent des données scalaires, les preuves de terminaison fondées sur les variants de boucles font un grand usage des entiers. Si l'arithmétique linéaire rationnelle comporte des algorithmes bien connus relativement efficaces en pratique (tels que le simplexe), dans le cas entier, il est souvent nécessaire de procéder à une analyse par cas fastidieuse et inefficace.

Avec un petit groupe de personnes issues de la partie « parisienne » du projet ANR Décert, nous avons décrit une nouvelle procédure pour l'arithmétique linéaire entière (sans quantificateurs). Cet algorithme combine de façon originale les idées de relaxation au domaine rationnel et d'énumération dans les entiers : au vu des bornes sur les variables dans les solutions rationnelles, ou bien on conclut immédiatement à l'existence ou non de solutions entières, ou bien dans un dernier cas, il faut énumérer toutes les valeurs possibles d'une variable bornée avant d'itérer le processus ; les bornes sont supposées fournies par un oracle.

Nous avons montré comment construire un tel oracle en simulant en une seule fois toutes les étapes de l'algorithme de Fourier-Motzkin par une application de la méthode du simplexe. Ce travail a été présenté à la conférence IJCAR en 2012 [13].

2.1.5 Valorisation

Le travail que nous réalisons autour d'Alt-Ergo avec Sylvain Conchon et nos étudiants n'a pas seulement un impact académique, traduit par une implication dans plusieurs projets financés par l'Agence Nationale pour la Recherche :

- A3PAT (2005 - 2009)
- Décert (2009 - 2012)
- BWare (2012 - 2016)

et un soutien par INRIA (ADT Alt-Ergo, 2009 - 2011). Bien sûr c'est un terrain d'expérimentation pour nos recherches sur les procédures de décision, et il est largement utilisé au sein de la plate-forme Why pour faire des preuves de programme. Mais de par son positionnement unique :

- Code source ouvert ;
- Licence CeCILL ;
- Localisation en Europe (ses concurrents directs, CVC3 et Z3 sont développés aux États-Unis),

il a attiré l'attention des industriels français.

Nous avons noué des contacts directs avec Airbus : la suite CAVEAT du CEA historiquement utilisée par Airbus offre depuis quelques années Alt-Ergo comme alternative à son propre démonstrateur et l'intention d'Airbus est maintenant d'utiliser Alt-Ergo en production pour ses nouveaux appareils [1].

Dans le cadre du projet FUI Hi-Lite (2010 - 2013), nous avons collaboré avec la société Adacore/Altran/Praxis, qui distribue maintenant Alt-Ergo avec SPARK/Ada, sa suite d'outils pour le développement en Ada.

Cependant, la recherche universitaire est un métier, répondre aux demandes d'utilisateurs industriels en est un autre. Depuis quelques mois, Alt-Ergo a deux versions, l'une académique, l'autre distribuée et maintenue par OcamlPro, où travaille maintenant notre ancien étudiant Mohamed Iguernelala, avec qui nous continuons à collaborer. Nous pourrons ainsi avec Sylvain Conchon nous consacrer à faire avancer Alt-Ergo sur le plan de la recherche, en laissant la partie centrée sur le développement du côté d'OcamlPro.

2.2 Preuve automatique ou assistée ?

Un second volet de mon travail ne rentre pas directement dans le cadre des procédures de décision dédiées aux preuves de programme, puisqu'il porte sur la réécriture, thématique historique de l'équipe Démons du LRI qui s'est transformée en Proval, puis Toccata sous l'influence de son association à INRIA. Néanmoins sans mon expertise dans ce dernier domaine, en particulier la complétion modulo AC, certains des travaux décrits ci-dessus n'auraient sans doute pas été réalisés.

Je m'intéresse depuis quelques années à l'articulation des approches automatiques et interactives pour la mécanisation des preuves, en particulier pour la terminaison des systèmes de réécriture et les preuves d'égalité entre termes modulo une théorie équationnelle.

Le tableau qui suit résume de façon très schématique les caractéristiques les plus saillantes des approches duales, automatique et interactive :

1. C'est dans ce but que Sylvain Conchon a participé à la qualification d'Alt-Ergo pour la norme DO-178C en 2011.

	Preuves mécanisées	
	Automatiques	Interactives
Facilité d'utilisation	+ rien à faire	- preuve assistée pas à pas
Expressivité	- logique du 1er ordre	+ ordre supérieur
Confiance	- gros logiciel sujet à bugs	+ petit noyau sûr

La terminaison est fondamentale puisque c'est elle qui permet d'assurer qu'une fonction définie par réécriture est bien définie sur tout son domaine (pas de calcul qui boucle).

Les outils mis en jeu en pratique sont, du côté automatique, le logiciel C*i*ME, une boîte à outils pour la réécriture que j'ai contribué à développer, et du côté interactif, l'outil Coq, fondé sur le calcul des constructions inductives.

Pour l'instant, mes travaux ont conduit à combler partiellement le manque de confiance dans les preuves automatiques : certains des algorithmes fondamentaux de la réécriture mis en œuvre dans C*i*ME [35] ont été formalisés en Coq, et leurs propriétés classiques ont été formellement démontrées. La bibliothèque Coq, vue comme un compagnon certifié de C*i*ME se nomme Coccinelle [29] et comporte dans sa version actuelle environ 56 000 lignes de script Coq.

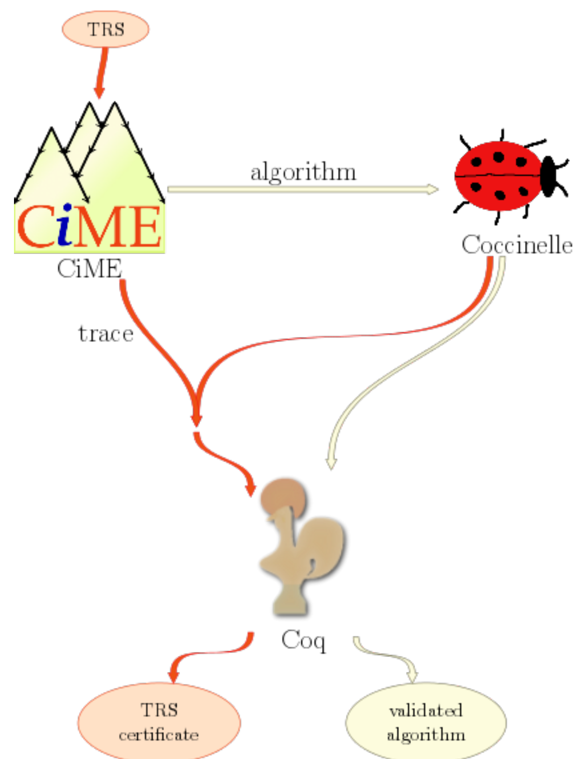


FIGURE 2.3 – Intrication entre preuves automatiques et assistées.

Les interactions entre C*i*ME, Coccinelle et Coq sont de deux ordres. Le parcours tracé

par les flèches claires dans la figure 2.3 désigne le cheminement suivi lors de la certification d'un algorithme de C*i*ME, modélisé puis prouvé formellement dans Coq. Cette preuve est générique, dans le sens où par construction, l'algorithme retourne un résultat conforme à sa spécification sur *toutes* les entrées.

Le réseau rouge explicite comment une propriété est vérifiée, pour *une entrée particulière* – une propriété pour un système de réécriture donné – par la production d'un certificat à partir d'une trace. Le certificat utilise deux composantes distinctes, l'une est générique, et démontrée dans Coccinelle une fois pour toutes, l'autre est un script Coq généré automatiquement à partir d'une trace produite par un outil externe, C*i*ME ou d'autres prouveurs « traçants » comme l'outil de terminaison AProVE [61]. Vérifier la trace (par exemple en compilant le script produit avec Coq) revient donc à vérifier qu'un théorème n'est appliqué que si ses prémisses sont satisfaites. Pour l'instant ce travail est toujours en cours, et l'essentiel de la partie sur Coccinelle n'a pas encore donné lieu à publication. Je le détaille dans le chapitre qui suit.

2.2.1 Égalité et filtrage

Le développement de Coccinelle [29, 30, 31] a été amorcé en 2004 par la modélisation et la certification de l'algorithme de filtrage modulo AC [28] que j'avais implanté dans C*i*ME. Cet algorithme, contrairement aux approches théoriques classiques dans la communauté de la réécriture, n'est pas une spécialisation de l'unification AC [16] fondée sur la résolution d'équations diophantiennes linéaires (sujet sur lequel j'ai également travaillé [27, 37, 2, 3]).

Dans les années 1990, peu d'implantations du filtrage AC existaient dans les outils de démonstration automatique, outre C*i*ME, à ma connaissance, seuls les outils daTac [111, 112], Elan [57] et Maude [90] proposaient cette fonctionnalité.

J'ai commencé par extraire une présentation formelle sous forme de règles d'inférence. Avant de prouver quoi que ce soit sur cet algorithme, j'ai fait un travail préparatoire pour montrer qu'il est équivalent de travailler sur des formes canoniques aplaties et triées (intuitivement $3+4+5+6$) et sur des termes bien formés ($(6+3)+(4+5)$). De façon surprenante, ce fait paraît si évident qu'il est rarement précisé et encore moins démontré (j'en ai trouvé un énoncé dans un article de Jean-Marie Hullot [72] mais le lecteur est renvoyé à un rapport difficile d'accès pour la preuve). La preuve formelle repose sur une présentation syntaxique de la théorie équationnelle AC [76, 77, 94] et est structurée autour de 7 lemmes, un pour chacun des axiomes de la présentation de AC. Ce travail préparatoire représente environ la moitié du travail total (5 000 lignes de script Coq sur 10 000). La partie sur l'algorithme lui-même est assez longue et fastidieuse ; cela m'a permis de découvrir un cas d'incomplétude dans l'algorithme original. Une fois cette erreur corrigée dans C*i*ME, le code C*i*ME et le code extrait de la preuve sont très similaires.

En 2005, avec Pierre Corbineau, dans le cadre sa thèse sur la démonstration automatique en théorie des types [43], nous avons réalisé une première étude sur la production de certificats d'égalité modulo une théorie équationnelle [32]. Ce travail, indépendant de Coccinelle, reposait uniquement sur C*i*ME et Coq et comportait deux aspects :

- La réflexion générique de la logique du premier ordre en Coq (environ 4500 lignes de Coq),

– La détermination des informations utiles à la production d'une trace, puis la production effective de cette trace sous forme d'un type inductif Coq – preuve réifiée.

Il a permis de mettre en évidence la nécessité d'ajouter la règle de coupure dans la logique. En effet, les preuves d'égalité produites par la complétion ordonnée contiennent naturellement des réécritures par des règles qui proviennent de paires critiques. Ces étapes peuvent être dépliées récursivement ou laissées telles quelles. La première solution est lourde et produit des traces énormes qui sont longues à vérifier. La seconde solution, plus élégante est celle que nous avons choisie, en regardant ces règles comme des lemmes intermédiaires, autrement dit des coupures. Nous avons conduit des expériences à l'aide d'un sous-ensemble pertinent de la base de problèmes TPTP (778 problèmes). Les résultats sont les suivants :

– 298 ne sont pas résolus par C*i*ME (timeout),
– 11 sont réfutés par C*i*ME (conjecture fausse),
– 239 ont des symboles associatifs-commutatifs (non traités par cette approche),
– 230 sont résolus par C*i*ME et vérifiés par Coq.

Cette approche souffre de quelques limitations. La première est qu'elle utilise des indices de de Bruijn pour modéliser la quantification universelle, ce qui a rendu la mise au point très délicate. Toutefois la principale est que l'égalité utilisée est l'égalité native de Coq, ce qui entraîne que les instanciations des règles de réécriture sont fondées sur le filtrage de Coq. Le traitement des symboles AC qui nécessite une égalité et un filtrage modulo est donc impossible dans ce cadre.

J'ai donc repris ce travail quelques années plus tard en modifiant la génération des traces pour qu'elles soient traitées conformément à l'approche de la figure 2.3. En effet dans Coccinelle, les termes sont un reflet quasi à l'identique des termes de C*i*ME et l'égalité modulo une théorie équationnelle est définie par un type inductif, et non plus par l'égalité native de Coq. Ce travail de reprise est une (petite) partie de l'article sur la génération de traces par C*i*ME [36] décrit ci-dessous. Pour traiter le cas des symboles AC, il reste maintenant à repenser aux traces modulo et, en particulier, à donner une bonne définition d'une position dans un terme AC aplati et canonisé. En revanche, tout le travail nécessaire a été fait du côté de Coccinelle (*confer supra*).

2.2.2 Terminaison

La terminaison des systèmes de réécriture est indécidable mais, depuis une quinzaine d'années, ce domaine est très actif.

En utilisant les critères modernes de paires de dépendance [7], la terminaison d'un système de réécriture se ramène à la terminaison d'une autre relation, qui à son tour se réduit à la terminaison de plusieurs sous-relations par des critères de graphes. Ces décompositions peuvent se poursuivre sur plusieurs niveaux. Ainsi se déploie une arborescence de règles d'inférence qui résume la structure de la preuve par l'application des différents théorèmes. Les feuilles de cet arbre correspondent à une preuve de terminaison par plongement dans un ordre bien fondé comme les interprétations polynomiales, les ordres récursifs sur les chemins – RPO –, ou les ordres de Knuth et Bendix – KBO.

$$\dfrac{\dfrac{< \dfrac{<_1 \text{ (poly. interp.)}}{\{\ldots \langle t_{1,i}, u_{1,i}\rangle \ldots\}} \quad \dfrac{< \dfrac{<_2 \text{ (RPO)}}{\{\ldots \langle t_{2,1,i}, u_{2,1,i}\rangle \ldots\}} \quad \dfrac{<_3 \text{ (poly. interp.)}}{\{\ldots \langle t_{2,2,i}, u_{2,2,i}\rangle \ldots\}} <}{\{\ldots \langle t_{2,i}, u_{2,i}\rangle \ldots\}}\ \text{Sub-Graph}}{R_{dp} = \{\ldots \langle t_i, u_i\rangle \ldots\}}\ \text{Graph}}{R_{init} = \{\ldots l_i \to r_i \ldots\}}\ \text{DP}$$

Cette structure d'arbre est par exemple mise en lumière par la notion de « processeurs » [62] utilisée dans l'outil AProVE.

La certification de ces preuves de terminaison complexes est devenue un enjeu lorsque certains outils automatiques ont répondu de façon contradictoire pendant les compétitions de terminaison. La communauté a ainsi produit un format commun de traces en XML, « Certification Problem Format » [45], afin de permettre les échanges entre les outils de terminaison et les outils de certification émergeants à l'époque. Ainsi AProVE a pû être connecté à C*i*ME [2]/Coccinelle pour produire des certificats.

De façon concomitante, une catégorie sur la terminaison certifiée a été intégrée dans la compétition de terminaison.

Une partie de la tâche des outils automatiques de terminaison consiste ainsi à *trouver* des ordres bien fondés dans lesquels plonger une relation R pour prouver sa terminaison. Pour RPO, il n'y a qu'un nombre fini de paramètres à faire jouer : le statut de chaque symbole et la précédence sur les symboles. Pour les ordres polynomiaux, le choix est beaucoup plus étendu : pour chaque symbole, il faut choisir le degré d du polynôme associé, puis une fois d choisi, il faut choisir les coefficients des monômes de degré inférieur à d dans $\mathbb{Z}$. Si ce dernier choix est retardé et les coefficients conservés sous forme de variables, les propriétés de décroissance de l'ordre sur la relation R se traduisent par des contraintes non linéaires dans les entiers, dont les variables sont les coefficients mentionnés ci-dessus. Il faut remarquer que même dans le cas où l'on se restreint à rechercher des polynômes linéaires, les contraintes à résoudre sont non linéaires à cause de l'imbrication des symboles de fonction dans les termes qui définissent la relation. Le problème est maintenant de *résoudre des contraintes non-linéaires sur les entiers*. Ce problème, le dixième problème de Hilbert, est indécidable. Cependant, on peut tenter une résolution dans les entiers bornés : si on réussit, on a une preuve de terminaison, dans le cas contraire, on ne peut en tirer aucune conclusion. Claude Marché, Ana-Paula Tomás, Xavier Urbain et moi-même avons étudié comment générer ces contraintes et comment les résoudre de façon efficace en généralisant les techniques de propagation connues en programmation par contraintes [41]. Ce travail a donné lieu à une implantation par Claude Marché qui a été utilisée dans d'autres outils, comme Mu-Term [86], Cariboo [59] ou Talp [95].

Avec Pierre Courtieu, Julien Forest Olivier Pons et Xavier Urbain (équipe CPR, laboratoire Cedric, CNAM & ENSIIE), qui forment avec moi le petit groupe qui travaille actuellement autour de C*i*ME, dans le cadre du projet ANR A3PAT, nous avons développé à la fois la *recherche* de preuves de terminaison, la production de *traces*, ainsi que celle de *certificats*.

L'approche générale [34] suit le réseau rouge (traces) de la figure 2.3.

2. C*i*ME est utilisé ici seulement comme un traducteur de CPF/XML vers un fichier pour Coq.

Coccinelle fournit la modélisation des différentes relations (théorie équationnelle, récriture, paires de dépendance), des ordres bien fondés (polynomiaux et RPO), ainsi que les preuves constructives des théorèmes utilisés. Nous avons pu proposer des critères affinés de terminaison [42, 33], obtenus en formalisant des théorèmes déjà connus et en affaiblissant les conditions d'application. Les preuves formelles permettent en effet de mettre en lumière que certaines hypothèses ne sont pas utilisées et peuvent donc être supprimées.

En parallèle, nous avons instrumenté C*i*ME pour en faire une version « traçante » [35, 36], qui produit des traces de deux sortes : d'une part des traces d'égalité modulo une théorie équationnelle et, d'autre part, des traces de terminaison, le tout dans le format CPF/XML ou directement sous la forme d'un script en Gallina pour l'assistant Coq.

De même que pour l'égalité, un bon ensemble de référence est fourni par TPTP, dans le cas de la terminaison, il est donné par la librairie TPDB. Sur les quelques 990 problèmes de TPDB, environ 300 sont certifiés dans le cadre actuel.

2.3 Vers des données certifiées

En 2010, l'équipe Toccata a été rejointe par Véronique Benzaken, spécialiste des langages centrés sur les données. Son expertise en bases de données et mes compétences en formalisation dans l'assistant à la preuve Coq nous ont permis de nous lancer depuis 2012 dans un travail de longue haleine qui vise à définir un cadre pour vérifier les (algorithmes sous-jacents des) systèmes de gestion de bases de données. Nous co-encadrons la thèse de Stefania Dumbrava sur ce sujet.

Les systèmes et applications actuels de gestion de données manipulent des volumes de données de plus en plus importants et il est crucial de s'assurer de leur disponibilité, de leur intégrité et de leur fiabilité. Il est naturel de penser que des outils de vérification analogues à ceux utilisés pour les programmes critiques écrits dans les langages classiques sont à même d'assurer les objectifs de sûreté, mais de façon surprenante, la recherche académique s'est peu attachée à étudier ce problème. L'obtention de garanties fortes requiert l'utilisation de méthodes formelles et d'outils matures et une approche très prometteuse consiste à utiliser des assistants de preuves.

2.3.1 Modèle relationnel de données

Les systèmes plus répandus traitent de bases de données relationnelles, fondées sur le modèle relationnel de données, utilisé dans différentes perspectives :

- Tout d'abord il *représente* des informations grâce à des *relations* ;
- Dans un souci de précision et d'exactitude, il permet de *raffiner* l'information représentée grâce à des *contraintes d'intégrité* ;
- Enfin, il donne des moyens d'*extraire* l'information grâce à des *langages de requêtes* fondés soit sur l'algèbre relationnelle, soit sur le calcul des tableaux (encore appelés requêtes conjunctives).

Il existe deux versions théoriquement équivalentes de ce modèle : la version nommée et la version par position. Dans le cadre nommé, les attributs sont considérés comme faisant partie intégrante de la base de données, et sont utilisés explicitement par les langages de

requêtes et les contraintes d'intégrité. Dans le cadre par position, les attributs spécifiques d'une relation sont ignorés, et les langages de requêtes ne disposent que de l'arité (c'est-à-dire le nombre des attributs) des relations. Les systèmes les plus répandus, tels Oracle, DB2, Postgresql ou encore Microsoft Access, reposent sur une version nommée du modèle relationnel.

2.3.2 Formalisations existantes

Les premiers travaux sur la modélisation de ces systèmes sont dûs à Gonzalia [66, 65] qui examine plusieurs façons de formaliser dans l'assistant Agda la version par position du modèle et étudie uniquement différentes définitions possibles des données et de l'algèbre relationnelle. Plus récemment, Malecha *et al.*, [87] ont proposé une formalisation dans le but de concevoir une implantation légère mais complétement vérifiée d'un système de bases de données relationnel. Ils montrent que leur implantation respecte sa spécification, toutes les preuves étant écrites et vérifiées dans Ynot, une extension de Coq [19]. Cependant, ils ont également choisi la version positionnelle du modèle, ont implanté un système *mono-utilisateur*, et n'ont traité ni les contraintes, ni les techniques sophistiquées d'optimisation.

2.3.3 Datacert

Notre but à long terme est de prouver que les systèmes *existants* sont conformes à leur spécification et de vérifier que les programmes qui font un usage intensif de requêtes sur des bases de données sont corrects ; il n'est pas d'implanter un système de gestion de bases de données en Coq.

Pour ce faire, la première étape – incontournable – est de formaliser le modèle relationnel de données. Comme les deux formalisations existantes sont éloignées des systèmes les plus utilisés en pratique, nous avons formalisé la version *nommée* du modèle, l'algèbre relationnelle et les requêtes conjonctives. Puisque ces dernières jouent un rôle central dans l'optimisation [3], nous avons fourni à la fois une spécification formelle et un algorithme certifié qui traduit les requêtes de l'algèbre relationnelle en requêtes conjonctives.

Nous avons étudié l'optimisation logique des requêtes dans les deux formalismes, et formellement prouvé les principaux « théorèmes des bases de données » : les équivalences algébriques, le théorème de l'homomorphisme, et la minimisation des requêtes conjonctives. Le théorème de minimisation permet de dériver un algorithme de minimisation certifié.

Enfin, nous avons spécifié les contraintes d'intégrité du modèle relationnel ; ces dernières sont centrales au niveau de la conception des bases des données, pour construire des structures de relations (appelées *schémas* dans le jargon des bases de données) qui possèdent de bonnes propriétés, ainsi qu'aux niveaux du compilateur et de l'optimiseur. Nous avons modélisé les dépendances, fonctionnelles et générales, qui sont considérées comme la classe de contraintes la plus importante par la communauté des bases de données. Pour ces deux classes, nous avons également traité le problème de l'implication, c'est-à-dire l'inférence de toutes les contraintes logiquement dérivables à partir d'un ensemble donné. Inférer *toutes* les contraintes est crucial, puisque s'il en manque, le compilateur ne peut pas optimiser autant que possible. Pour dépendances fonctionnelles, nous avons formalisé l'algorithme

3. elles admettent une forme optimisée exactement équivalente.

connu sous le nom d'« axiomes d'Armstrong » et prouvé formellement qu'il est correct et complet. Pour les dépendances générales, nous avons prouvé la correction la procédure qui déduit de nouvelles dépendances, le « chase ».

La présentation informelle de tous ces concepts se trouve dans les livres de référence sur les bases de données [1, 108, 98]. Datacert, la formalisation que nous en avons réalisée en Coq comporte 24 000 lignes essentiellement consacrées à la formalisation des bases de données (non comptées les parties auxiliaires sur les ensembles finis, les comparaisons décidables, les exemples de test, *etc*).

Nom de fichier	spec.	proof	comment.	contenu
Data.v	105	105	48	Attributs, n-uplets et relations
RelationalAlgebra.v	930	2086	135	Définition et sémantique de l'algèbre relationnelle
AlgebraOptimisation.v	109	1723	154	Équivalences algébriques structurelles
Tableaux.v	364	1497	175	Formalisation des requêtes conjonctives (tableaux), homomorphisme et minimisation
Translation.v	731	3274	248	Traduction de l'algèbre vers les requêtes conjonctives
Matching.v	380	886	156	Filtrage pour l'homomorphisme de tableaux et le chase
Unify.v	781	3550	59	Unification pour la traduction
Chase.v	645	3506	191	Procédure du chase
Armstrong.v	212	1268	146	Système d'inférence d'Armstrong
Tree.v	241	431	44	Définitions auxilliaires pour les ordres sur les expressions de l'algèbre
Total	4498	18326	1356	

Notre formalisation atteint un haut degré d'abstraction et de modularité, et c'est, à notre connaissance, la mécanisation la plus réaliste de la théorie classique des bases de données. Son interface est disponible [4] et les travaux qui la décrivent commencent à être publiés [10].

4. http://datacert.lri.fr/esop/html/Datacert.AdditionalMaterial.html

Chapitre 3

Algèbres de termes

Les mathématiciens manipulent quotidiennement des objets génériques comme les groupes, les anneaux, les corps, les espaces vectoriels. Ils ne sont cependant pas encore assez abstraits pour le logicien ou l'informaticien théorique, qui souhaitent parler de, et raisonner sur ces structures une fois pour toutes, qu'il y ait une addition et une multiplication, ou seulement une addition, ou un minimum/maximum, ou d'autres opérateurs plus attachés aux structures de données comme la concaténation de listes, l'accès dans un tableau ou l'ensemble des successeurs d'un nœud dans un graphe fini. Les algèbres universelles (ou encore algèbres de termes) donnent un formalisme commun pour exprimer toutes ces notions. Elles ont été introduites par Alfred Whitehead en 1898 [113], mais George Grätzer indique dans la préface de son livre [67] que les premiers résultats sur ce sujet ne remontent pas avant les travaux de Garrett Birkhoff [11], au début des années 1930. Toutefois, dans sa thèse [68], Jacques Herbrand utilise également ce concept (bien que sans le nommer algèbre universelle) pour élaborer son célèbre théorème logique.

De nos jours, les algèbres universelles sont une abstraction commode et largement répandue dans la communauté des informaticiens théoriques dite du « volume B [1] » [109].

Ce chapitre esquisse à grand traits des travaux que j'ai menés sur le traitement de l'égalité dans les algèbres de termes. Le lecteur pourra se reporter à l'ouvrage de Franz Baader et Tobias Nipkow [8] pour les définitions détaillées des notions utilisées ci après, et à l'usage des notations préconisé par Nachum Dershowitz et Jean-Pierre Jouannaud [53]. Toutefois, je présenterai certaines notions de base de façon assez précise afin de souligner dans le cours de ce chapitre les différences qui existent entre les définitions livresques, l'implantation dans un démonstrateur automatique et les formalisations dans un assistant à la preuve. Il est pour moi très clair que dans un développement formel, le plus important est le choix de modélisation des objets manipulés ; les preuves de théorèmes sont parfois longues et souvent fastidieuses, mais si les choix de départ sont bons, elles sont faisables. Après avoir rappelé les définitions et les résultats les plus classiques, plutôt que de faire un catalogue de résultats formellement prouvés en Coq, je vais indiquer et motiver mes choix de modélisation.

1. Je remercie Pierre Lescanne pour m'avoir fait remarquer que la division du « Handbook of Theoretical Science » entre volume A, algorithmique, et volume B, logique, recouvre une structuration forte de cette communauté.

3.1 Préliminaires et résultats classiques

3.1.1 Termes

Les termes sont définis à partir d'une signature (et de variables). C'est une notion purement syntaxique, c'est pourquoi il est question de *symboles* de fonction et de *symboles* de variables, et non pas simplement de fonctions et de variables. Pour rendre compte de la nature diverse des objets construits – entiers naturels, entiers relatifs, listes, *etc.* –, les symboles de fonction en portent l'information grâce à des *sortes*.

Définition 3.1 (Signature) *Une signature est un triplet $(\mathscr{S}, \mathscr{F}, \tau)$ où,*
- *$\mathscr{S}$ est un ensemble non vide de sortes ;*
- *$\mathscr{F}$ est un ensemble de symboles de fonctions ;*
- *τ est une fonction définie sur les éléments de $\mathscr{F}$ et à valeur dans les séquences non vides d'éléments de $\mathscr{S}$. Si $\tau(f) = (s_1, \ldots, s_n, s)$, on note*

$$f : s_1 \times s_2 \times \ldots \times s_n \to s$$

Le domaine de f est égal à $s_1 \times \ldots \times s_n$, son codomaine à s et son arité à n.

Exemple 3.2 *La signature utilisée pour les entiers de Peano est égale à $(\mathscr{S}_{Peano}, \mathscr{F}_{Peano}, \tau_{Peano})$:*

$$\mathscr{S}_{Peano} = \{P, bool\}$$
$$\mathscr{F}_{Peano} = \{Zero, Succ, Add, Mul, vrai, faux, PlusPetit\}$$

et τ_{Peano} est définie par :

$$\begin{array}{l} Zero : P \\ Succ : P \to P \\ Add : P \times P \to P \\ Mul : P \times P \to P \\ vrai : bool \\ faux : bool \\ PlusPetit : P \times P \to bool \end{array}$$

Définition 3.3 (Termes) *Soit $(\mathscr{S}, \mathscr{F}, \tau)$ une signature et $\mathscr{X} = \uplus_{s \in \mathscr{S}} \mathscr{X}_s$ un ensemble dont les éléments sont appelés symboles de variable, tel que les $\mathscr{X}_s$ sont deux à deux disjoints.*
- *si x appartient à $\mathscr{X}_s$, x est un terme de sorte s.*
- *si f appartient à $\mathscr{F}$, $f : s_1 \times s_2 \times \ldots \times s_n \to s$, $t_1, \ldots, t_n$ sont des termes respectivement de sortes $s_1, \ldots, s_n$, alors $f(t_1, \ldots, t_n)$ est un terme de sorte s.*

L'ensemble des termes bâtis sur $(\mathscr{S}, \mathscr{F}, \tau)$ et $\mathscr{X}$ est noté $\mathscr{T}(\mathscr{F}, \mathscr{X})$. Si $\mathscr{X}$ est réduit à l'ensemble vide, $\mathcal{T}(\mathscr{F}, \emptyset)$, noté $\mathscr{T}(\mathscr{F})$, est appelé l'algèbre des termes clos sur $\mathscr{F}$.

Exemple 3.4 *Le terme*

$$Mul(Add(Zero, Succ(Succ(Zero))), Succ(Zero))$$

bâti sur la signature $(\mathscr{S}_{Peano}, \mathscr{F}_{Peano}, \tau_{Peano})$ *est de sorte* P*, tandis que*

$$PlusPetit(Succ(Add(Succ(Zero), Succ(Zero))), Zero)$$

est de sorte bool.

Dans le reste de cette section, nous supposerons fixée une algèbre de termes $\mathscr{T}(\mathscr{F}, \mathscr{X})$.

Définition 3.5 (Sous-termes) *Étant donné t un terme de* $\mathscr{T}(\mathscr{F}, \mathscr{X})$*, l'ensemble des sous-termes de t, noté* $ST_{\trianglelefteq}(t)$ *est défini par*

- *si* $t = x$ *est une variable,* $ST_{\trianglelefteq}(x) = \{x\}$ *;*
- *si* $t = f(t_1, \ldots, t_n)$*,* $ST_{\trianglelefteq}(f(t_1, \ldots, t_n)) = \{f(t_1, \ldots, t_n)\} \cup \bigcup_i ST_{\trianglelefteq}(t_i)$

La notion de sous-terme induit deux relations, celle de sous-terme, et celle de sous-terme strict : si s est un sous-terme de t, on le note $t \trianglerighteq s$ *– ou* $s \trianglelefteq t$ *–, si s est un sous-terme de t, distinct de t, on le note* $t \triangleright s$ *– ou* $s \triangleleft t$*.*

Pour « naviguer » dans les sous-termes d'un terme, il est commode d'utiliser la notion de position :

Définition 3.6 (Position) *Les positions sont des mots – plus simplement dit, des séquences – construits sur les entiers positifs,* Λ *dénote le mot vide,* $\cdot$ *la concaténation (infixe) de deux mots et* $\leq_{pref}$ *l'ordre préfixe usuel sur les mots. Étant donné t un terme de* $\mathscr{T}(\mathscr{F}, \mathscr{X})$*, l'ensemble des positions de t, noté Pos(t) est défini par*

- *si* $t = x$ *est une variable,* $\mathcal{P}os(x) = \{\Lambda\}$ *;*
- *si* $t = f(t_1, \ldots, t_n)$*,* $\mathcal{P}os(f(t_1, \ldots, t_n)) = \{\Lambda\} \cup \bigcup_i \{i \cdot p \mid p \in \mathcal{P}os(t_i)\}$*.*

Soit t un terme, et p une position de ce terme : elle permet d'identifier un sous-terme unique de t, noté $t|_p$ *et défini par :*

- *si* $p = \Lambda$*,* $t|_\Lambda = t$ *;*
- *si* $p = i \cdot q$*, alors nécessairement* $t = f(t_1, \ldots, t_i, \ldots, t_n)$*, et* $t|_{i \cdot q}$ *est défini par* $t_i|_q$*.*

Exemple 3.7 *L'ensemble des positions du terme*

$$t = Mul(Add(Zero, Succ(Succ(Zero))), Succ(Zero))$$

est égal à

$$\{\Lambda,\ 1,\ 1{\cdot}1,\ 1{\cdot}2,\ 1{\cdot}2{\cdot}1,\ 1{\cdot}2{\cdot}1{\cdot}1,\ 2,\ 2{\cdot}1\}$$

et le sous terme de t à la position $1 \cdot 2$ *est* $Succ(Succ(Zero))$*.*

Notation 3.8 *Si s et t sont deux termes, et p une position de t,* $t[s]_p$ *est le terme défini par*

- *si* $p = \Lambda$*,* $t[s]_\Lambda = s$*,*
- *si* $p = i \cdot q$ *et* $t = f(t_1, \ldots, t_i, \ldots, t_n)$*,* $t[s]_{i \cdot q} = f(t_1, \ldots, t_i[s]_q, \ldots, t_n)$*.*

3.1.2 Substitutions

Conformément à la sémantique usuelle des variables, les symboles de variables ont pour finalité de représenter d'autres objets – ici des termes –, et d'être remplacés par eux. C'est ce que fait une substitution :

Définition 3.9 (Substitution) *Une substitution σ de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ est une application de $\mathcal{X}$ dans $\mathcal{T}(\mathcal{F}, \mathcal{X})$ qui préserve les sortes et qui est égale à l'identité sauf sur un nombre fini de symboles de variables, son* domaine*, noté $\mathcal{D}om(\sigma)$. Si $\mathcal{D}om(\sigma) = \{x_1, \dots, x_n\}$, σ est parfaitement définie dès lors que l'on connaît $t_1, \dots, t_n$, les valeurs prises sur $x_1, \dots, x_n$. On utilisera la notation*

$$\sigma = \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$$

Si tous les termes $t_1, \dots, t_n$ sont clos, la substitution elle-même sera dite close.

La notation usuelle dans la littérature pour $\sigma(x)$ est postfixée : $x\sigma$.

Une substitution σ s'étend de façon naturelle en une fonction *unique* $\mathcal{H}_\sigma$ de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ dans $\mathcal{T}(\mathcal{F}, \mathcal{X})$ par :

- $\mathcal{H}_\sigma(x) = x$ si $x \notin \mathcal{D}om(\sigma)$,
- $\mathcal{H}_\sigma(x_i) = x_i\sigma$ si $x_i \in \mathcal{D}om(\sigma)$,
- $\mathcal{H}_\sigma(f(s_1, \dots, s_m)) = f(\mathcal{H}_\sigma(s_1), \dots, \mathcal{H}_\sigma(s_m))$ si $f(s_1, \dots, s_m)$ est un terme de $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

À partir de maintenant nous confondrons volontairement une substitution σ et son extension $\mathcal{H}_\sigma$.

3.1.3 Théorie équationnelle

Les théories équationnelles rendent partiellement compte de l'égalité dans les algèbres de termes : les groupes rentrent dans ce cadre, mais pas les corps. En effet, les mathématiciens définissent les groupes par l'ensemble d'axiomes suivant :

$$\begin{array}{lrcl}
\forall x, & e \bullet x & = & x \\
\forall x, & x \bullet e & = & x \\
\forall x, & I(x) \bullet x & = & e \\
\forall x, & x \bullet I(x) & = & e \\
\forall x\ y\ z, & (x \bullet y) \bullet z & = & x \bullet (y \bullet z)
\end{array}$$

et les corps par :

$$
\begin{array}{llcl}
\forall x, & 0 + x & = & x \\
\forall x, & x + 0 & = & x \\
\forall x, & -(x) + x & = & 0 \\
\forall x, & x + -(x) & = & 0 \\
\forall x\ y\ z, & (x + y) + z & = & x + (y + z) \\
\forall x\ y, & x + y & = & y + x \\
\forall x, & 1 \bullet x & = & x \\
\forall x, & x \bullet 1 & = & x \\
\forall x, & x \neq 0 \Rightarrow x^{-1} \bullet x & = & 1 \\
\forall x, & x \neq 0 \Rightarrow x \bullet x^{-1} & = & 1 \\
\forall x\ y\ z, & (x \bullet y) \bullet z & = & x \bullet (y \bullet z)
\end{array}
$$

Dans le cas des groupes, tous les axiomes sont des égalités universellement quantifiées, tandis que dans les corps, les axiomes pour l'inverse à gauche et l'inverse à droite sont conditionnels et demandent que l'élément à inverser soit non nul.

$$\frac{}{s = s} \quad \text{Réflexivité}$$

$$\frac{s = t}{t = s} \quad \text{Symétrie}$$

$$\frac{s = t \qquad t = u}{s = u} \quad \text{Transitivité}$$

$$\frac{s = t}{u[s\sigma]_p = u[t\sigma]_p} \quad \text{Remplacement}$$

FIGURE 3.1 – Règles d'inférence de la logique équationnelle.

Dans le cadre général d'une algèbre de termes $\mathcal{T}(\mathcal{F}, \mathcal{X})$, la théorie équationnelle associée à un ensemble d'identités implicitement universellement quantifiées E entre les termes de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ est l'ensemble des égalités que l'on peut déduire de E. Grâce au théorème de Birkhoff, il est possible d'utiliser plusieurs notions équivalentes de déduction :

- une notion sémantique, $E \models s = t$, si tout modèle de E est un modèle de $s = t$;
- une notion syntaxique, $E \vdash s = t$, si $s = t$ peut se déduire de E en utilisant les règles d'inférence de la logique équationnelle – symétrie, transitivité, instanciation et mise sous contexte (voir la figure 3.1).

Exemple 3.10 *La théorie équationnelle associée aux identités de l'arithmétique de Peano*

sur l'algèbre $\mathcal{T}(\mathscr{F}_{Peano}, \{x_1, x_2, \ldots\})$

$$\begin{array}{lcl} Add(x_1, Zero) & = & x_1 \\ Add(x_1, Succ(x_2)) & = & Succ(Add(x_1, x_2) \\ Mul(x_1, Zero) & = & Zero \\ Mul(x_1, Succ(x_2)) & = & Add(Mul(x_1, x_2), x_1) \\ PlusPetit(Zero, x_1) & = & vrai \\ PlusPetit(Succ(x_1), Zero) & = & faux \\ PlusPetit(Succ(x_1), Succ(x_2)) & = & PlusPetit(x_1, x_2) \end{array}$$

permet de caractériser les égalités closes entre les entiers naturels munis de l'addition et de la multiplication et le prédicat $\leq$ *sur ces entiers.*

Dans le reste de ce chapitre, nous noterons par $s =_E t$ le fait que l'égalité $s = t$ est une conséquence de l'ensemble d'identités E et nous nous intéresserons uniquement aux preuves syntaxiques ; l'exposition de ces preuves sous forme arborescente est encombrante et est habituellement remplacée par une séquence plus compacte d'*étapes équationnelles* :

$$s \xleftrightarrow[u=v,\sigma]{p} t \text{ si et seulement si } s|_p = u\sigma \text{ et } t = s[v\sigma]_p$$

Lorsque cette information est inutile, nous omettrons la position p, la substitution σ, ou remplacerons l'indication de l'identité $u = v$ utilisée par E, un ensemble d'identités qui la contient ; ainsi $s =_E t$ est équivalent à $s \xleftrightarrow[E]{*} t$.

Dans le cadre des algèbres de termes, le problème du mot, à savoir si étant donnés E, s et t, $s =_E t$ est valide, intéresse une bonne partie de la communauté de la démonstration automatique.

3.1.4 Réécriture

La réécriture traite du prédicat d'égalité en utilisant l'idée simple d'orienter les identités entre deux termes. Une identité orientée, $l \rightarrow r$, appelée *règle de réécriture*, ne peut être utilisée que de gauche à droite. L'analogue d'une étape équationnelle dans le cas orienté est une étape de de réécriture :

$$s \xrightarrow[l\rightarrow r,\sigma]{p} t \text{ si et seulement si } s|_p = l\sigma \text{ et } t = s[r\sigma]_p$$

Un ensemble de règles – ou système – de réécriture R définit une relation $\underset{R}{\rightarrow}$ de la même façon qu'un ensemble d'identités E définit $\underset{E}{\longleftrightarrow}$. La différence, qui semble minime au premier abord, entre $\underset{E}{\longleftrightarrow}$ et $\underset{R}{\rightarrow}$ est que $\underset{E}{\longleftrightarrow}$ est symétrique, tandis que $\underset{R}{\rightarrow}$ ne l'est pas.

Le caractère orienté de la réécriture permet de la voir comme une réduction, ou un calcul ; un exemple typique en est le λ-calcul, même si sa formulation standard ne peut pas être donnée sous forme d'un système de règles de réécriture ; cependant certaines variantes, notamment celles avec substitutions explicites, qui décomposent la règle de β-réduction en plusieurs étapes élémentaires, rentrent dans le cadre de la réécriture du premier ordre.

Le revers de la médaille de l'orientation des identités est que certaines séquences d'étapes équationnelles ne peuvent pas être simulées par la réécriture. L'étude des liens entre ces deux notions peut se faire dans un premier temps sur des systèmes abstraits, qui ne prennent pas en compte la définition exacte de $\underset{R}{\longleftrightarrow}$ et $\underset{R}{\rightarrow}$, mais seulement le fait que $\underset{R}{\longleftrightarrow}$ est la clôture symétrique de $\underset{R}{\rightarrow}$.

Définition 3.11 (Church-Rosser, confluence, confluence locale) *Soit $\mathscr{A}$ un ensemble. Une relation $\underset{R}{\rightarrow}$ sur $\mathscr{A}$*

– *possède la propriété de* Church-Rosser *si et seulement si*

$$\forall x, y \in \mathscr{A},\ x \ \underset{R}{\overset{*}{\longleftrightarrow}}\ y \ \Rightarrow\ (\exists z \in \mathscr{A},\ x \ \underset{R}{\overset{*}{\rightarrow}}\ z \ \underset{R}{\overset{*}{\leftarrow}}\ y).$$

– *est* confluente *si et seulement si*

$$\forall x, y, z \in \mathscr{A},\ y \ \underset{R}{\overset{*}{\leftarrow}}\ x \ \underset{R}{\overset{*}{\rightarrow}}\ z \ \Rightarrow\ (\exists v \in \mathscr{A},\ y \ \underset{R}{\overset{*}{\rightarrow}}\ v \ \underset{R}{\overset{*}{\leftarrow}}\ z).$$

– *est* localement confluente *si et seulement si*

$$\forall x, y, z \in \mathscr{A},\ y \ \underset{R}{\leftarrow}\ x \ \underset{R}{\rightarrow}\ z \ \Rightarrow\ (\exists v \in \mathscr{A},\ y \ \underset{R}{\overset{*}{\rightarrow}}\ v \ \underset{R}{\overset{*}{\leftarrow}}\ z).$$

Certaines preuves d'équivalence entre les notions définies ci-dessus font appel à l'induction sur la structure des séquences de réécriture, la notion de terminaison est donc cruciale :

Définition 3.12 (Terminaison) *Soit $\mathscr{A}$ un ensemble. Une relation $\underset{R}{\rightarrow}$ sur $\mathscr{A}$ termine s'il n'existe pas de suite infinie $(a_n)_n$ dans $\mathscr{A}$, telle que*

$$a_1 \underset{R}{\rightarrow} a_2 \underset{R}{\rightarrow} \dots a_n \underset{R}{\rightarrow} a_{n+1} \dots$$

Proposition 3.13 *Une relation $\rightarrow_R$ possède la propriété de Church-Rosser si et seulement si elle est confluente.*

Définition 3.14 *Soient $\mathscr{A}$ un ensemble et $\underset{R}{\rightarrow}$ une relation sur $\mathscr{A}$. $\underset{R}{\rightarrow}$ est convergente si elle termine et est confluente.*

Théorème 3.15 (Lemme de Newman) *Soit $\mathscr{A}$ un ensemble et $\underset{R}{\rightarrow}$ une relation sur $\mathscr{A}$ qui termine ; la relation $\underset{R}{\rightarrow}$ est confluente si et seulement si elle est localement confluente.*

Si un système de réécriture R définit une relation $\underset{R}{\rightarrow}$ localement confluente et qui termine, la théorie équationnelle associée est parfaitement connue : pour savoir si deux termes sont égaux modulo cette théorie, il suffit de les réduire complètement par $\underset{R}{\rightarrow}$, et de tester si les termes réduits sont identiques.

Les deux problèmes cruciaux qui se posent pour résoudre le problème du mot modulo R sont donc celui de la terminaison de $\underset{R}{\rightarrow}$, et celui de sa confluence locale. Dans ce qui suit, je n'aborderai que le problème de la terminaison.

3.1.5 Terminaison de $\underset{R}{\rightarrow}$

Ordres de réduction

Dans un article de 1970 [88], Zohar Manna et Stephen Ness ont divisé le problème de la terminaison d'une relation de réécriture en deux parties indépendantes :

- Trouver un *ordre* [2] bien-fondé, compatible avec la mise sous contexte et l'instanciation ;
- Montrer que les *règles* de réécriture sont strictement décroissantes vis-à-vis d'un tel ordre.

Un ordre ayant les propriétés ci-dessus est appelé *ordre de réduction*. Une fois posée cette méthodologie, ils déclinent une collection d'exemples de systèmes de réécriture, appelés historiquement algorithmes de Markov, et les ordres utilisés pour prouver leur terminaison.

Cet article pose les fondements de l'étude de la terminaison des systèmes de réécriture, en donnant des conditions très générales sur les propriétés des ordres utilisables, étendant les travaux précédents de Renato Iturriaga [99].

De là, l'intérêt de découvrir des ordres de réduction, ce qui amena en une dizaine d'années la description de plusieurs familles de tels ordres.

En 1978, David Plaisted a proposé un ordre possédant les propriétés requises et défini récursivement sur la structure arborescente des termes [96]. L'année suivante, Nachum Dershowitz et Zohar Manna ont défini MPO [3] [54]. En 1980, en utilisant les mêmes idées, Sam Kamin et Jean-Jacques Levy ont introduit LPO [4] [75]. Ces définitions ont été fondues dans le cadre général du RPO [5] par Nachum Dershowitz [51].

À la même période (1979), et dans une autre lignée, Dallas Lankford [81] donnait une définition générale des ordres polynomiaux, dont Manna et Ness n'avaient donnés que des exemples dans leur article de 1970.

Motivés par des considérations un peu différentes, l'étude de la confluence locale, et sa « réparation » par la complétion, en 1970, Donald Knuth et Peter Bendix [78] ont également proposé un ordre de réduction, KBO [6], mais sans se placer dans l'optique de prouver la terminaison des systèmes de réécriture.

Pour conclure sur les débuts de l'histoire de la terminaison de la réécriture, en 1978, Gérard Huet et Dallas Lankford ont montré que le problème était décidable dans le cas clos par une preuve directe, et indécidable dans le cas général [71] par une réduction au problème de l'arrêt pour les machines de Turing. Ce dernier résultat sera raffiné une dizaine d'années plus tard, quand Max Dauchet [46] montrera qu'il suffit d'une seule règle linéaire gauche pour simuler une machine de Turing universelle.

L'étude théorique la la terminaison de la réécriture standard s'est endormie pendant quelques années, tandis que commençait à se développer la mécanisation des ordres, et la recherche automatique de preuves de terminaison. Le premier logiciel dans cette lignée est REVE [83] de Pierre Lescanne au début des années 1980, suivi d'ORME [84], toujours de Pierre Lescanne. À la fin des années 1990, un certain nombre de logiciels dédiés à la preuve

2. En fait, la partie stricte d'un ordre, c'est-à-dire une une relation transitive.
3. « Multiset Path Ordering »
4. « Lexicographic Path Ordering »
5. « Recursive Path Ordering »
6. « Knuth-Bendix Ordering »

de terminaison automatique des systèmes de réécriture ont vu le jour : CiME [40], auquel j'ai activement participé, AProVE [61], TTT [69], et d'autres, ce qui a amené en la mise en place en 2004 d'une compétition d'outils pour créer une synergie dans leur développement. La floraison de ces outils a été encouragée par l'apparition d'une nouvelle technique, plus puissante que celle de Manna et Ness, et qui nécessite des ordres « plus faibles ».

Paires de dépendance

En 1997, Thomas Arts et Jürgen Giesl ont proposé un nouveau critère de terminaison [7] fondé sur l'analyse des réductions en cascade potentielles dans les membres droits des règles de réécriture. Ces réductions sont appelées *paires de dépendance*.

Définition 3.16 (Paire de dépendance, chaîne de dépendance) *Les* paires de dépendance *associées au système de réécriture R sont définies par :*

$$\mathscr{D}(R) = \{f \in \mathscr{F} \mid \exists l \to r \in R, l(\Lambda) = f\}$$
$$\mathrm{DP}(R) = \{\langle u, v\rangle \mid \exists l \to r \in R, \exists p, u = l \wedge v|_p = r \wedge r(\Lambda) \in \mathscr{D}(R)\}$$

Une chaîne de dépendance *est une séquence de paires de* $\mathrm{DP}(R)$, $(\langle u_i, v_i\rangle)_i$, *telle qu'il existe une séquence de substitutions* σ_i *avec*

$$v_i\sigma_i \xrightarrow[R]{\neq\Lambda\ *} u_{i+1}\sigma_{i+1}$$

Théorème 3.17 (Arts & Giesl) *Soit R un ensemble de règles de réécriture tel que :*
- $\forall\, l \to r \in R,\ l \notin \mathscr{X}$ *(membre gauche non variable) ;*
- $\forall\, l \to r \in R, \mathcal{V}ar(r) \subseteq \mathcal{V}ars(l)$ *(régularité).*

La relation $\underset{R}{\to}$ *termine s'il n'existe pas de chaîne de dépendance infinie pour R.*

Les hypothèses ne restreignent pas la portée du résultat. En effet, il est immédiat qu'un système de réécriture qui n'est pas régulier, ou qui a des règles dont les membres gauches sont variables, ne termine pas.

Nachum Dershowitz a remarqué que grâce à la minimalité des sous-termes immortels [7] dans la preuve d'Arts & Giesl, certaines paires pouvaient directement être éliminées car inutiles, les paires $\langle u, v\rangle$ telles que v est un sous-terme strict de u [52].

D'autre part, il est possible de de transformer une chaîne de dépendance en « marquant » tous les symboles de tête des termes qui y apparaissent. Comme les étapes de réécriture par le système de départ n'ont jamais lieu en tête, cela conduit seulement à modifier la définition des paires elles-mêmes.

De manière plus générale, à la suite de ce premier énoncé, grâce à de nombreux raffinements, la notion de chaîne de dépendance a été étendue, de manière à éliminer la relation entre le système utilisé pour réécrire dans les sous-termes stricts et la relation principale $\mathrm{DP}(R)$, ce qui a conduit à la définition d'étape relative ou « relative dependancy step » :

7. qui peuvent se réduire infiniment.

Définition 3.18 (étape RDP) *Soient D et R deux systèmes de réécriture, et deux termes s et t,*

$$s \xrightarrow[D;R]{} t \text{ si et seulement si } s \xrightarrow[R]{\neq\Lambda\ *} \xrightarrow[D]{\Lambda} t$$

Le théorème d'Arts & Giesl énonce donc que la terminaison de $\xrightarrow[R]{}$ est équivalente à celle de $\xrightarrow[\mathrm{DP}(R);R]{}$. Tous les développements ultérieurs sur les critères de terminaison étudiés dans ce cadre ont consisté à étudier la terminaison des relations de la forme $\xrightarrow[D;R]{}$, que ce soit en décomposant D en des sous-relations dont la terminaison est modulaire, en trouvant des ordres permettant d'enlever des paires dans D, des règles dans R, ou en examinant soigneusement quelles sont les règles de R qui peuvent s'appliquer sur les sous-termes de D. Tout un bestiaire de critères a vu le jour, et associé à eux, un certain nombre d'ordres bien fondés, mais qui ne sont plus nécessairement des ordres de réduction.

3.2 Implanter et formaliser : C*i*ME et Coccinelle

3.2.1 Comment implanter une algèbre universelle ?

Une algèbre $\mathscr{T}(\mathscr{F},\mathscr{X})$ est donnée par une signature $\mathscr{F}$ et un ensemble de variables $\mathscr{X}$. Si dans certains livres et dans la section précédente, les signatures et les variables sont accompagnées de sortes, si l'on part de termes bien formés, bien sortés, tous les nouveaux termes fabriqués par les mécanismes décrits ci-dessus, calcul de paires critiques, réduction par un système de réécriture, *etc*, ne fabriquent que des termes bien formés et bien sortés. C'est pourquoi la plupart des outils de démonstration automatique qui manipulent des équations et des règles de réécriture au premier ordre se placent dans un cadre non sorté.

Reprenons l'exemple des entiers de Peano et examinons quelles en sont les implantations/modélisations possibles. Il faut commencer par représenter les termes définis grâce à la signature $\{Zero, Succ, Add, Mul, vrai, faux, PlusPetit\}$. Si l'on décide de représenter les variables par leurs noms, c'est-à-dire des chaînes de caractères, un choix possible pour implanter cette algèbre en OCaml est :

```
type peano_term =
  | Var of string
  | Zero
  | Succ of peano_term
  | Add of peano_term * peano_term
  | Mul of peano_term * peano_term
  | Vrai
  | Faux
  | PlusPetit of peano_term * peano_term
```

avec pour contrepartie en Coq :

```
Inductive peano_term : Set :=
  | Var : string → peano_term
  | Zero : peano_term
  | Succ : peano_term → peano_term
  | Add : peano_term → peano_term → peano_term
  | Mul : peano_term → peano_term → peano_term
```

```
| Vrai : peano_term
| Faux : peano_term
| PlusPetit : peano_term → peano_term → peano_term.
```

Le terme $Add(x, Succ(Zero))$ est alors exprimé en OCaml par Add (Var "x", Succ Zero) et par Add (Var "x") (Succ Zero) en Coq.

Ainsi chaque algèbre a un type propre de terme, et il n'est pas possible de construire des termes mal formés, comme $Add(Zero, Zero, Add)$. Cette façon de représenter les algèbres de termes est appelée *plongement superficiel* (de l'algèbre dans le langage hôte, ici OCaml ou Coq).

La contrepartie malheureuse de cette modélisation est que tous les algorithmes et procédures classiques (unification, filtrage, réécriture, complétion) doivent être réécrits pour chaque nouvelle algèbre.

En vérité, il ne s'agit pas d'implanter *une* algèbre, mais *un schéma* d'algèbre, qui prenne justement en compte le caractère *universel* des algèbres de termes. C'est l'objet du *plongement profond* décrit ci-après.

Les versions successives de la boîte à outils CiME ont été implantées dans différents dialectes de ML. Dans la première version de 1993 en Camllight [38], les termes étaient définis comme un type polymorphe en deux paramètres, l'un pour les symboles de fonction, l'autre pour les symboles de variables :

```
type ('symb, 'var) term =
    Var of 'var
  | Term of 'symb * ('symb, 'var) term list
```

Ce type permet d'accueillir toutes les algèbres universelles, simplement en instanciant le paramètre 'symb par un type propre à chaque algèbre ; en définissant de façon indépendante la signature des entiers de Peano par

```
type peano_sig = Zero | Succ | Add | Mul | Vrai | Faux | PlusPetit
```

et en conservant les chaînes de caractères pour les variables, le type peano_term est simplement (peano_sig, string) term et le terme $Add(x, Succ(Zero))$ est exprimé par Term (Add, [Var "x"; Term (Succ, [Term (Zero,[])])]).

Il faut noter que si le plongement profond offre la possibilité d'écrire une fois pour toutes, c'est-à-dire pour le type ('symb, 'var) term, tous les algorithmes et procédures des algèbres universelles, il a le désavantage de permettre d'exprimer des termes mal formés : ($Add(Zero, Zero, Add)$ se code en Term (Add,[Term (Zero,[]); Term (Zero,[]); Term (Add,[])])).

Avec le plongement profond, une algèbre universelle est une constellation de types et de fonctions, et le choix du polymorphisme semble au premier abord très approprié : il est possible de construire facilement des algèbres libres, où chaque symbole de fonction est simplement un nom (*i. e.* une chaîne de caractères), mais aussi des algèbres où certains symboles sont associatifs et commutatifs (AC), les symboles peuvent ainsi être codés par des enregistrements qui comportent des informations supplémentaires, comme l'arité, le statut vis-à-vis de AC, des indications pour l'impression, telles que préfixe, infixe ou suffixe. Nous avons également essayé de représenter les symboles par des entiers, les informations nécessaires étant retrouvées grâce à un dictionnaire (map).

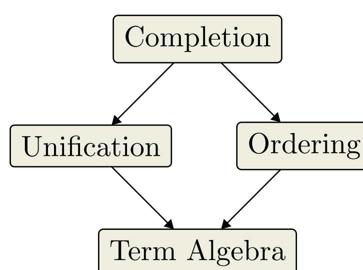
Malheureusement, il est apparu très délicat d'implanter les algorithmes classiques, en particulier ceux utiles pour la complétion modulo AC, ainsi divers ordres sur les termes compatibles avec AC. Pour corriger les erreurs dans le code, il a été nécessaire de pouvoir imprimer les termes depuis presque toutes les fonctions Caml. À cause du polymorphisme, il était impossible d'avoir une unique fonction d'impression disponible au niveau global ; la seule possibilité fut donc d'ajouter une fonction d'impression comme argument supplémentaire aux fonctions à corriger.

Lorsqu'OCaml succéda à Camllight, nous tirâmes parti du nouveau système de modules, en considérant une algèbre de termes comme un module, construit par un foncteur à partir de deux modules, l'un pour les symboles de fonction, l'autre pour les variables. Le module des symboles de fonction contenait essentiellement des indications pour l'impression. L'architecture de la deuxième version de C*i*ME [39] repose entièrement sur le système de modules d'OCaml ; par exemple, l'unification y est un foncteur qui prend comme argument un module algèbre de termes, et l'impression des termes est directement disponible pour toutes les fonctions définies dans ce foncteur. Grâce à l'efficacité des modules d'OCaml, nous avons obtenu une version aussi élégante qu'efficace des algèbres de termes, chaque objet mathématique ayant une exacte contrepartie au niveau de l'implantation.

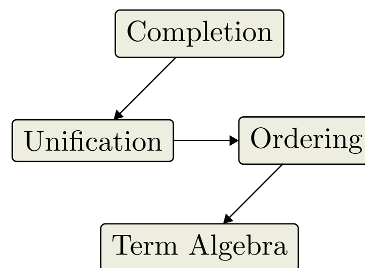
La première modélisation des algèbres universelles que j'ai effectuée en Coq visait à certifier l'algorithme de filtrage modulo AC de C*i*ME [28]. Elle remonte à 2004, précisément au moment où Coq s'est doté d'un système de modules comparable à celui d'OCaml.

Le manuel de référence de Coq version 8 [105] indique :
« Enfin, le système de modules de Coq complète le tableau de Coq 8.0. Bien que diffusé à titre expérimental dans la précédente version 7.4, il doit être considéré comme un trait saillant de la nouvelle version. »
En version anglaise, « Finally, the module system of Coq completes the picture of Coq version 8.0. Though released with an experimental status in the previous version 7.4, it should be considered as a salient feature of the new version. ».

C'était une décision naturelle de fonder le développement en Coq, voulu alors comme un reflet de C*i*ME, sur l'utilisation de modules, comme l'outil C*i*ME lui-même. Malheureusement, quelques années plus tard, il s'est avéré que bien que très satisfaisant pour l'esprit, le système de modules de Coq était assez lent à instancier les foncteurs. De plus, il était impossible de définir un foncteur à plusieurs arguments, partageant certains types ou modules : un graphe de dépendance en OCaml tel que



doit être linéarisé, par exemple en :



Cette restriction du partage conduit à un empilement de dépendances entre les modules d'autant plus dommageable que plus un module est haut dans la hiérarchie, plus est lente l'instanciation du foncteur qui l'engendre. C'est un problème devenu critique lorsque Coccinelle a été utilisée pour produire des certificats de terminaison pour les systèmes de réécriture : les foncteurs sont compilés une fois pour toutes, mais le processus d'instanciation est lancé pour chaque certificat, et prend alors un temps très important.

Dans une première tentative pour supprimer les foncteurs de Coccinelle, j'ai remplacé les modules par des enregistrements, ce qui résout les problèmes d'efficacité, mais rend la recherche des lemmes connus par la directive Coq SearchAbout particulièrement fastidieuse : chaque lemme est un champ dans un enregistrement dont la taille excède de beaucoup celle des pages d'un éditeur de preuves standard.

D'un autre côté, comme un développement formel en Coq est moins sujet à des erreurs que sa contrepartie en OCaml, une connexion extrêmement étroite entre toutes les fonctions d'une algèbre de termes n'est pas indispensable ; la conception actuelle de Coccinelle repose sur le choix premier fait dans CiME, à savoir une constellation de types et de fonctions polymorphes. Une liaison faible est assurée par le mécanisme de sections de Coq, qui permet d'avoir des variables globales à l'intérieur d'une section, et de masquer ainsi les arguments supplémentaires dans les lemmes techniques.

Les termes sont modélisés en Coq par plongement profond en

```
Inductive term (symbol variable : Set) : Set :=
    Var : variable → term symbol variable
  | Term : symbol → list (term symbol variable) → term symbol variable
```

où variable et symbol sont des variables de types.

En réalité, le mécanisme de section de Coq permet d'alléger un peu les notations :

```
Section Sec.
Hypothesis symbol : Set.
Hypothesis variable : Set.

Inductive term : Set :=
  | Var : variable → term
  | Term : symbol → list term → term.

...
End Sec.
```

Dans ce qui suit, les variables variable et symbol seront supposées fixées, et resteront implicites.

Le choix du polymorphisme a déjà été discuté plus haut. Cependant la richesse des types de Coq, en particulier ses types dépendants, pourraient autoriser une plus grande précision, par exemple en contraignant le nombre d'arguments d'un symbole de fonction à être égal à

l'arité de ce symbole. Il est même possible d'imaginer un système de types réifiés, qui s'appuyerait sur les sortes des symboles, afin de construire uniquement des termes parfaitement bien formés. Ce n'est pas le chemin que j'ai suivi, pour plusieurs raisons :

– Comme déjà dit plus haut, à partir de termes bien formés, les procédures classiques ne fabriquent que des termes bien formés ;
– Pour être vraiment des miroirs des structures de données OCaml, les termes ne doivent pas contenir de preuves ;
– Les preuves des propriétés classiques sont en général plus légères, et les théorèmes où la bonne formation des termes est nécessaire sont exactement identifiés ;
– Dans certaines techniques de preuve de terminaison, en particulier les AFS (« argument filtering systems »), certains symboles voient leur arité modifiée : une contrainte de bonne formation stricte sur les termes obligerait à une traduction vers une autre algèbre, inutile dans notre cas.

En sus de Coccinelle, à ma connaissance, il existe deux autres développements formels conséquents sur les algèbres de termes et la terminaison de la réécriture, CoLoR, écrit en Coq et dont le principal contributeur est Frédéric Blanqui, et CeTA/IsaFoR, écrit en Isabelle par René Thiemann.

Les termes de CeTA sont construits en Isabelle dans le même esprit que celui de Coccinelle, mais ce choix n'est pas celui de CoLoR, où deux types de termes du premier ordre cohabitent, l'un identique aux termes CeTA/Coccinelle, et le second avec adéquation forcée entre l'arité et le nombre d'arguments. Il est bien sûr nécessaire d'avoir des traductions de l'un vers l'autre, d'autant que CoLoR réutilise certaines parties de mon propre développement, en particulier le RPO.

Dans tout langage de programmation raisonnable, il existe une fonction qui permet de tester si deux données de base (entiers, chaînes de caractères, *etc*) sont identiques. Dans le cas des algèbres universelles, il est essentiel de pouvoir comparer les symboles de fonction (et également les symboles de variables). Il faut donc intégrer ce fait au sein de l'assistant de preuves, ce qui ne va pas de soi, car Coq ne possède pas de façon native le principe du tiers exclus - deux symboles sont égaux ou non. C'est une hypothèse qui est faite sur les briques de base que constituent les symboles de fonctions, et les variables, et c'est le seul vestige du choix des enregistrements qui subsiste dans le développement actuel : en Coccinelle, une signature comporte un type pour les symboles de fonction, une fonction booléenne pour tester l'égalité, ainsi qu'une preuve d'adéquation de cette fonction. Il en va de même pour les variables.

3.2.2 Modéliser la réécriture et les étapes RDP

Une relation de réécriture est définie dans les livres comme la clôture par instanciation et mise sous contexte d'un ensemble de règles de réécriture :

$$s \longrightarrow_R t \quad \text{if} \quad l \longrightarrow r \in R, \ \ s|_p \equiv l\sigma \ \ \text{and} \ \ t \equiv s[r\sigma]_p$$

Dans les premières versions de C*i*ME, qui ne produisaient pas de traces, la notion d'étape de réécriture était implicite : à partir d'un terme et d'un système de réécriture, nous avions implanté une fonction de normalisation « innermost », de manière récursive. La no-

tion de substitution [8] subsistait, mais pas celle de position de réduction. Une représentation plus proche de la littérature a été implantée lors de la production de traces, que ce soit pour la première version de 2004 [32], ou celle de 2011 [36].

La modélisation en Coccinelle fait également l'économie de la notion de sous-terme, et de position. Elle est définie en plusieurs strates. La première étape définit la généralisation $R^{\#}$ d'une relation R définie sur un domaine A aux n-uplets de A par

$$\forall\, t, t', t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_n,$$
$$(t_1, \dots, t_{i-1}, t, t_{i+1}, \dots, t_n)\ R^{\#}\ (t_1, \dots, t_{i-1}, t', t_{i+1}, \dots, t_n) \iff t\, R\, t'$$

La formalisation (sur des listes au lieu de n-uplets) par one_step_list distingue deux cas, soit les éléments en relation sont en tête, soit ils se trouvent dans la dernière partie de la liste :

```
Inductive one_step_list (A : Type) (R : relation A) : relation (list A) :=
   | head_step : ∀t1 t2 l, R t1 t2 → one_step_list R (t1 :: l) (t2 :: l)
   | tail_step : ∀t l1 l2, one_step_list R l1 l2 → one_step_list R (t :: l1) (t :: l2).
```

Enfin (une étape de) la réécriture est tout d'abord définie en tête (clôture par instanciation), puis dans les sous-termes (clôture par mise sous contexte).

```
(** *** One step at top. *)
Inductive axiom (R : relation term) : relation term :=
   | instance : ∀t1 t2 σ , R t1 t2 → axiom R (apply_subst X σ t1) (apply_subst X σ t2).

(** *** One step at any position. *)
  Inductive one_step (R : relation term) : relation term :=
  | at_top : ∀t1 t2, axiom R t1 t2 → one_step R t1 t2
  | in_context: ∀f l1 l2, one_step_list (one_step R) l1 l2 → one_step R (Term f l1) (Term f l2).
```

Remarquons que $\underset{R}{\rightarrow}$ est définie pour une relation R quelconque, et pas nécessairement pour un ensemble fini de règles de réécriture.

Comme dans les articles théoriques, dans la modélisation, la définition des paires de dépendance et celle des étapes RDP sont indépendantes :

```
Inductive defined (R : relation term) : symbol → Prop := Def : ∀f l t, R t (Term f l) → defined R f.

Inductive dp (R : relation term) : term → term → Prop:=
  | Dp : ∀t1 t2 p f2 l2, R t2 t1 → subterm_at_pos t2 p = Some (Term f2 l2) → defined R f2 →
          dp R (Term f2 l2) t1.

Inductive rdp_step (D R : relation term) : term → term → Prop :=
   | Rdp_step : ∀f l1 l2 t3, refl_trans_clos (one_step_list (one_step X R)) l2 l1 → D t3 (Term f l2) →
                rdp_step D R t3 (Term f l1).
```

Le théorème principal d'Arts & Giesl s'exprime alors simplement, sous les hypothèses usuelles, avec la condition supplémentaire que l'on sait décider si un symbole de fonction est défini ou non pour le système de réécriture. Cette condition reflète le fait que la preuve, même directe, repose sur une analyse par cas. Bien sûr, comme on dispose d'une égalité décidable sur les symboles de fonction et de variables, si l'on suppose que R est un ensemble fini de règles de réécriture, cette condition est trivialement satisfaite, mais pour énoncer le théorème en toute généralité, elle est nécessaire.

8. instance obtenue par un algorithme de filtrage du terme à réduire vers les membres gauches de règles.

```
Section SecDP.

Hypothesis R : relation term.
Hypothesis RR_var : R_var R.
Hypothesis RR_reg : R_reg X R.
Hypothesis RDC : ∀f, {defined R f} + {¬ defined R f}.

(** Plain standard dependency pair criterion *)
Lemma dp_criterion : well_founded (rdp_step (axiom (dp R)) R) → well_founded (one_step R).

End SecDP.
```

Coccinelle formalise et prouve bon nombre des critères du bestiaire dompté par CFP :

- Étiquettage sémantique des symboles pour des systèmes de réécriture (FiniteModelLab.v) ;
- Mise sous contexte « plat » des systèmes de réécriture (FlatContext.v)
- Paires de dépendance usuelles (DependencyPairs.v) ;
- Paires de dépendance à la Dershowitz ;
- Paires de dépendance marquées ;
- Critère de graphe : terminaison modulaire par (surapproximation des) composantes du graphe de connexion des paires (DPGraph.v) ;
- Élimination de certaines paires (ou règles) grâce à des ordres bien fondés compatibles (DPRemoval.v) ;
- Analyse des règles utilisables (UsableRules.v) ;
- Critère « size change » (SizeChange.v),

ainsi que de nombreux ordres, polynomiaux sur $\mathbb{N}$, sur $\mathbb{Q}$, sur des algèbres tropicales, et RPO.

Plutôt que détailler tous les théorèmes et toutes les preuves formelles de Coccinelle, je vais expliquer quelles ont été les difficultés rencontrées sur l'exemple la formalisation de RPO, et les problèmes soulevés par la génération automatique de scripts de preuves à partir des certificats produits par les outils de preuve de terminaison automatiques.

3.3 Un éclairage sur RPO

3.3.1 Définition, propriétés et formalisation

La définition classique de l'ordre récursif sur les chemins, RPO [51], procède par la définition de l'ordre large $\preceq_{rpo}$, et en dérive l'ordre strict $\prec_{rpo}$ et l'équivalence associée $\simeq_{rpo}$.

Définition 3.19 *Soit $\preceq$ une précédence (c'est-à-dire une relation réflexive et transitive) sur $\mathscr{F}$, et soit status une application de $\mathscr{F}$ dans l'ensemble à deux éléments $\{lex, mul\}$ compatible avec $\simeq$*[9]*. En outre, si $f \simeq g$ et $status(f) = status(g) = lex$, alors f et g ont la même arité. $\preceq$ est étendue à $\mathscr{F} \cup \mathscr{X}$, une variable n'étant comparable à aucun élément de $\mathscr{F} \cup \mathscr{X}$ (autre qu'elle-même). L'ordre récursif sur les chemins $\preceq_{rpo}$ est défini par*

$$s \equiv f(s_1, \ldots, s_n) \preceq_{rpo} t \equiv g(t_1, \ldots, t_m)$$

9. Comme à l'ordinaire, $\simeq$ est la relation d'équivalence définie par $\preceq \cap \succeq$, et $\prec$ est l'ordre strict associé $\preceq \setminus \succeq$.

si l'une des conditions ci-dessous est remplie :

1. $\exists t_j \;\; s \preceq_{rpo} t_j$
2. $f \prec g$ *et* $\forall i \in \{1,\ldots,n\} \;\; s_i \prec_{rpo} t$
3. $f \simeq g$, $status(f) = lex$, $(s_1,\ldots,s_n)(\preceq_{rpo})_{lex}(t_1,\ldots,t_n)$ *et* $\forall i \in \{1,\ldots,n\} \;\; s_i \prec_{rpo} t$.
4. $f \simeq g$, $status(f) = mul$ *et* $\{s_1,\ldots,s_n\}(\preceq_{rpo})_{mul}\{t_1,\ldots,t_m\}$

La formalisation de RPO dans Coccinelle diffère essentiellement sur deux points de l'article original de Nachum Dershowitz :

- Dans la formalisation, les relations principales sont $\simeq_{rpo}$ et $\prec_{rpo}$, et $\preceq_{rpo}$ est définie comme leur union,
- La preuve formalisée de bonne fondation de $\prec_{rpo}$, est différente de la preuve originale, qui ne reposait sur le lemme de Kruskal, et utilisait le fait que RPO contient le plongement.

L'équivalence $\simeq_{rpo}$, peut être définie directement par le type inductif equiv, sans aucune référence à $\preceq_{rpo}$:

```
(** ** Definition of the equivalence associated with rpo. *)
Inductive equiv : term → term → Prop :=
  | Eq : ∀ t, equiv t t
  | Eq_lex : ∀ f l1 l2, status f = Lex → equiv_list_lex l1 l2 → equiv (Term f l1) (Term f l2)
  | Eq_mul : ∀ f l1 l2, status f = Mul → _permut equiv l1 l2 → equiv (Term f l1) (Term f l2)

with equiv_list_lex : list term → list term → Prop :=
   | Eq_list_nil : equiv_list_lex nil nil
   | Eq_list_cons : ∀ t1 t2 l1 l2, equiv t1 t2 → equiv_list_lex l1 l2 → equiv_list_lex (t1 :: l1) (t2 :: l2).

(** equiv is actually an equivalence. *)
Lemma equiv_equiv : equivalence term equiv.
```

La construction auxilliaire _permut equiv l1 l2 indique que les listes de termes l1 et l2 sont des permutations l'une de l'autre vis-à-vis de la relation equiv. La preuve qu'equiv est une relation d'équivalence est simple et courte.

La partie stricte $\prec_{rpo}$ est formalisée par l'inductif rpo. Une dernière différence – mineure – par rapport à la définition originale, et la possibilité de comparer deux termes de même symbole de tête f, de statut lexicographique, mais n'ayant pas le même nombre d'arguments (en dessous d'une certaine borne bb pour conserver la bonne fondation). Cette extension a été faite à la demande des membres de l'équipe de développement d'AProVE [61] dans le but de pouvoir comparer des termes après avoir appliqué une transformation AFS (arguments filtering systems). Après cette transformation, les termes ne sont plus bien formés selon la signature de départ.

```
(** ** Definition of the strict part of rpo. *)
Inductive rpo (bb : nat) : term → term → Prop :=
  | Subterm : ∀ f l t s, mem equiv s l → rpo_eq bb t s → rpo bb t (Term f l)
  | Top_gt : ∀ f g l l', prec g f → (∀ s', mem equiv s' l' → rpo bb s' (Term f l)) → rpo bb (Term g l') (Term f l)
  | Top_eq_lex :
        ∀ f l l', status f = Lex → (length l = length l' ⋁ (length l' ≤ bb ⋀ length l ≤ bb)) → rpo_lex bb l' l →
        (∀ s', mem equiv s' l' → rpo bb s' (Term f l)) →
        rpo bb (Term f l') (Term f l)
  | Top_eq_mul : ∀ f l l', status f = Mul → rpo_mul bb l' l → rpo bb (Term f l') (Term f l)

with rpo_eq (bb : nat) : term → term → Prop :=
```

```
  | Equiv : ∀ t t', equiv t t' → rpo_eq bb t t'
  | Lt : ∀ s t, rpo bb s t → rpo_eq bb s t

with rpo_lex (bb : nat) : list term → list term → Prop :=
  | List_gt : ∀ s t l l', rpo bb s t → rpo_lex bb (s :: l) (t :: l')
  | List_eq : ∀ s s' l l', equiv s s' → rpo_lex bb l l' → rpo_lex bb (s :: l) (s' :: l')
  | List_nil : ∀ s l, rpo_lex bb nil (s :: l)

with rpo_mul ( bb : nat) : list term → list term → Prop :=
  | List_mul : ∀ a lg ls lc l l', permut l' (ls ++lc) → permut l (a :: lg ++lc) →
      (∀ b, mem equiv b ls → ∃a', mem equiv a' (a :: lg) ⋀ rpo bb b a') → rpo_mul bb l' l.
```

Les constructeurs de rpo correspondent exactement aux quatre cas de la définition originale.

Les liens entre les relations $\prec_{rpo}$, $\preceq_{rpo}$ et $\simeq_{rpo}$ sont formellement prouvés :

```
Lemma equiv_rpo_equiv_1 : ∀ bb t t', equiv t t' → (∀ s, rpo bb s t ⟷ rpo bb s t').

Lemma equiv_rpo_equiv_2 : ∀ bb t t', equiv t t' → (∀ s, rpo bb t s ⟷ rpo bb t' s).

Lemma equiv_rpo_equiv_3 : ∀ bb t t', equiv t t' → (∀ s, rpo_eq bb s t ⟷ rpo_eq bb s t').

Lemma equiv_rpo_equiv_4 : ∀ bb t t', equiv t t' → (∀ s, rpo_eq bb t s ⟷ rpo_eq bb t' s).
```

Nous avons ensuite démontré que $\prec_{rpo}$ est bien un ordre de réduction :

```
Lemma rpo_trans : ∀ bb u t s, rpo bb u t → rpo bb t s → rpo bb u s.

(** ** RPO is compatible with adding context. *)
Lemma equiv_monotonic : ∀ f l1 l2, one_step_list equiv l1 l2 → equiv (Term f l1) (Term f l2).
Lemma rpo_monotonic : ∀ bb f l1 l2, one_step_list (rpo bb) l1 l2 → rpo bb (Term f l1) (Term f l2).

(** ** RPO is compatible with the instanciation by a substitution. *)
Lemma equiv_stable : ∀ s t σ , equiv s t → equiv (apply_subst X σ s) (apply_subst X σ t).

Lemma rpo_stable : ∀ bb s t σ , rpo bb s t → rpo bb (apply_subst X σ s) (apply_subst X σ t).

(** ** Main theorem: when the precedence is well−founded, so is the rpo. *)
Lemma wf_rpo : well_founded prec → ∀ bb, well_founded (rpo bb).
```

Curieusement, la preuve la plus longue et la plus complexe est celle de la transitivité de $\prec_{rpo}$.

La preuve originale de bonne fondation de $\prec_{rpo}$ repose sur le lemme de Kruskal, et utilise le fait que RPO contient le plongement. Cette preuve n'est pas directement constructive. Dans Coccinelle, j'ai formalisé une preuve directe fondée sur des candidats de réductibilité à la Tait-Girard, à partir de celle proposée par Jean-Pierre Jouannaud et Albert Rubio dans leur article sur l'extension de RPO aux termes d'ordre supérieur [74], et formalisée par Adam Koprowski [79]. Cependant, la définition de Koprowski et la nôtre sont incomparables, car si Koprowski traite de l'ordre supérieur, notre définition est plus puissante, c'est-à-dire compare plus de termes, que celle de Koprowski restreinte au premier ordre : nous autorisons les statuts lexicographique et multi-ensemble, tandis que dans [79], seul le statut multi-ensemble est disponible.

3.3.2 Comparaison effective des termes

Cette formalisation permet de se convaincre que $\prec_{rpo}$ est bien un ordre de réduction, mais après tout, nous aurions pu tout aussi bien nous contenter de l'article sur papier de

1982 ; en effet, le but est de mécaniser la comparaison des termes, et avec la définition par un type inductif, pour montrer que deux termes sont comparables, il faut en donner une *preuve*.

Une possibilité serait de tenter de programmer une telle preuve grâce à la définition d'une tactique. Dans ce cas précis, cette solution est à proscrire :
- Une tactique peut échouer ;
- La définition de $\prec_{rpo}$, avec son grand nombre de cas, et son caractère récursif demanderait une tactique très complexe.

Une autre possibilité serait de demander aux outils automatiques de terminaison de donner une trace des calculs qu'ils ont effectués pour parvenir à un résultat sur la comparaison de deux termes. Outre le fait que les développeurs de ces outils ne sont pas nécessairement prêts à faire cet effort, les calculs réalisés ne sont pas un reflet exact du type inductif Coq.

Comme la mécanisation de $\prec_{rpo}$ est déjà à l'œuvre dans C*i*ME, j'ai donc adopté l'approche suivante :

1. Formaliser en Coq la *fonction* de C*i*ME qui évalue le résultat de la comparaison de deux termes par RPO ;
2. Prouver que cette fonction est bien équivalente à l'inductif rpo.

Les avantages sont de deux ordres, tout d'abord renforcer la confiance dans les algorithmes de C*i*ME, et ensuite obtenir une preuve par calcul, qui ne peut pas échouer. La définition de la fonction rpo_eval en Coq demande environ 150 lignes [10], ce qui est comparable à la fonction écrite en OCaml. La différence principale est que les fonctions doivent être totales en Coq, et vu la complexité des appels récursifs, j'ai choisi d'avoir un argument entier supplémentaire qui décroît de 1 à chaque appel récursif. La fonction rpo_eval prend donc les arguments suivants [11] :
- Une borne bb pour comparer lexicographiquement les listes de sous-termes de longueur différente ;
- Un compteur entier ;
- Deux termes à comparer.

Elle retourne un résultat dans le type des comparaisons :

```
Inductive comp : Set :=
  | Equivalent
  | Less_than
  | Greater_than
  | Uncomparable.
```

Le lemme suivant montre non seulement que lorsque la fonction d'évaluation retourne un résultat, celui-ci est correct, mais que si le compteur est suffisamment grand, elle donnera effectivement un résultat :

```
Lemma rpo_eval_is_complete :
  ∀ bb n t1 t2, size t1 + size t2 ≤ n →
        match rpo_eval bb n t1 t2 with
        | Some Equivalent ⇒ equiv t1 t2
        | Some Less_than ⇒ rpo bb t1 t2
        | Some Greater_than ⇒ rpo bb t2 t1
```

10. Le lecteur intéressé peut consulter le code en ligne à l'adresse `http://coccinelle.lri.fr/`
11. La précédence et le statut sont donnés dans une section, et peuvent être considérés comme des paramètres « cachés ».

```
| Some Uncomparable ⇒ ¬ equiv t1 t2 ∧ ¬ rpo bb t1 t2 ∧ ¬ rpo bb t2 t1
| None ⇒ False
end.
```

3.3.3 Quelques chiffres

Les deux aspects, formalisation et preuves du RPO, et définition de la fonction rpo_eval avec sa preuve d'adéquation, sont dans deux fichiers séparés. Voici les résultats produits par `coqwc`, qui indique pour chacun la quantité de définitions (formalisation), de preuves, et de commentaires :

Nom de fichier	spec	proof	comments	contenu
RpoDef.v	284	1568	213	Définitions et preuves
RpoCompute.v	327	1335	357	Fonction de calcul et adéquation
total	611	2903	570	

3.4 Conclusion

3.4.1 Statistiques

Dans son état actuel, Coccinelle comporte un peu plus de 50 000 lignes de code, dont la répartition entre les différentes parties, définition des termes, réécriture, critères de terminaison, ordres de réduction et autres est détaillée ci-dessous.

sous-répertoire	spec	proof	comments	contenu
Terms	2008	6321	991	termes, substitutions, sous-termes, etc.
Rewriting	458	2166	115	définition des théories équationnelles et de la réécriture
Termination/DependencyPairs	1017	3106	1573	critères de terminaison pour les relations RDP
Termination/RewritingSystems	465	1321	162	critères de terminaison pour la réécriture
Termination/TermOrderings	2194	9312	1425	ordres variés sur les termes
Termination/Tranformations	286	867	108	transformations des relations qui préservent la terminaison
NonTermination	105	176	16	critères basiques de non-terminaison
Structures	3817	10637	2357	définitions et lemmes auxiliaires
total	10350	33906	6747	

La génération automatique de certificats est quant à elle écrite en ℂDuce (1 500 lignes) pour l'analyse des fichiers xml au format CPF et OCaml (7 000 lignes) pour la partie génération proprement dite. Actuellement, environ 400 traces sont transformées en certificats pour chacun des ensembles de tests proposés lors des compétitions (environ un millier à

chaque fois). Ce nombre était monté à environ 800 avant la restructuration de la génération de certificats par une approche réflexive.

La marge de progression est due à deux facteurs :

– Exploiter ceux qui ne le sont pas encore (ou plus) parmi les critères déjà formalisés ;
– Formaliser de nouveaux critères.

Le second point est parlant et se passe d'explications plus détaillées. Concernant le premier point, ça n'est pas parce qu'un critère est formellement prouvé qu'il est immédiatement exploitable, il faut réfléchir à la façon de vérifier que ses hypothèses sont bien satisfaites quand il est appliqué sur un problème de terminaison en particulier. Cette preuve peut se faire par tactique, et dans ce cas échouer, ou alors de façon réflexive et par calcul, et cela demande parfois un travail supplémentaire de formalisation (*confer* l'exemple du RPO).

Une troisième source d'amélioration, davantage du ressort des concepteurs des outils de terminaison automatique, est l'élaboration de nouveaux critères, qui pourront alors être formalisés et exploités. Ce n'est pas complètement en dehors du champ de la formalisation, car il s'avére parfois que cette dernière mette en évidence des hypothèses inutil(isé)es des preuves sur papier, donnant ainsi un critère plus puissant [33].

3.4.2 Liens entre outils automatiques et formalisation

L'embryon de Coccinelle a démarré en 2004, et les premières preuves formelles de terminaison écrites « à la main » ont suivi de peu. CoLoR, développé en Coq principalement par Frédéric Blanqui a vu le jour peu après en 2005. Les réponses discordantes des outils de terminaision automatique ont convaincu la communauté travaillant sur ce sujet que la certification formelle des résultats, qui semblait réalisable, était un enjeu important. La catégorie « terminaison certifiée » a ainsi vu le jour en 2007 et ses premiers participants ont été C*i*ME certifié par Coccinelle, et TPA et TTT 2, tous deux certifiés par CoLoR+Rainbow [12].

René Thiemann, tout d'abord développeur d'AProVE, un outil de terminaison automatique, s'est tourné en 2009 vers le développement de la troisième bibliothèque formelle de terminaison, IsaFoR/CeTA, en Isabelle/HOL, dans le but précisément de certifier les résultats d'AProVE. En 2010, le petit groupe de « certificateurs » (dont je faisais partie) s'est mis d'accord sur un format commun de traces, CPF, « Certification Problem Format » de façon à « brancher » n'importe quel outil automatique de terminaison sur n'importe quel outil de certification. Une petite communauté s'est ainsi formée, regroupant également les développeurs d'outils automatiques sensibilisés aux problématiques des preuves formelles (du moins dans le cadre de la terminaison de la réécriture). C'est ainsi que Coccinelle peut vérifier aussi bien les certificats produits par C*i*ME, que ceux produits par AProVE.

3.4.3 Dissémination

De 2005 à 2009, j'ai participé au projet ANR A3PAT « Assister Automatiquement les Assistants de Preuves avec des Traces », qui a soutenu le démarrage du développement de Coccinelle. Pierre Castéran en était également membre, et travaillait sur la formalisation des ordinaux en Coq. Nous avons alors intégré une version de RPO spécialisée aux ordinaux

12. Le générateur de certificats associé.

pour faire facilement et rapidement des preuves d'accessibilité sur lesquelles il travaillait depuis quelques temps déjà. Le résultat de cette collaboration a donné lieu à la contribution pour Coq nommée Cantor.

Toute la partie de Coccinelle sur RPO, preuves et calcul, a été intégrée par Frédéric Blanqui dans sa bibliothèque de terminaison certifiée CoLoR Rainbow. Cette utilisation est grandement facilitée par le fait que CoLoR est également développé en Coq.

Pour terminer avec l'utilisation de RPO en dehors de Coccinelle, Sorin Stratulat l'utilise dans la production de scripts Coq à partir de son démonstrateur Spike pour faire des preuves par « induction sans induction ».

3.4.4 Futur

Le beau dialogue rendu possible par l'adoption de CPF trouve toutefois ses limites : pour un système de réécriture donné, il existe plusieurs preuves de terminaison possibles, mais toutes ne sont pas également vérifiables par tous les outils, qui n'implantent/ne formalisent pas tous les critères disponibles. Les outils automatiques doivent donc être finement réglés pour chaque certificateur. Pour disposer de toute la puissance de toutes les bibliothèques de terminaison formelles, *au sein d'une même preuve*, il faudrait que CPF évolue vers un cadre permettant d'échanger des preuves partielles, en n'éludant évidemment pas le problème de la cohérence des différentes théories logiques qui sous-tendent chaque développement formel.

Chapitre 4

Perspectives

Quand j'ai commencé à faire de la recherche, j'ai abordé les preuves mécanisées dans l'optique de la démonstration automatique, et même sous l'angle précis de la résolution et de la réécriture, qui étaient les thématiques de l'équipe Demons à l'époque où j'y ai fait ma thèse. Tout en conservant un intérêt marqué pour ces sujets, grâce à différentes rencontres scientifiques, j'ai peu à peu élargi mon point de vue sur les preuves. Dans l'équipe de Christine Paulin, j'ai découvert Coq et la beauté des preuves interactives, avec Sylvain Conchon, je me suis initiée aux démonstrateurs SMT, et avec Véronique Benzaken, j'ai compris l'intérêt de formaliser les bases de données. Cette « aventure de l'esprit » m'a enrichie, et je souhaite poursuivre mes travaux sur plusieurs voies.

4.1 Ingénierie de la preuve

J'ai été amenée à réaliser des développements en Coq très conséquents (plus de 70 000 lignes en tout) sur la réécriture et sa terminaison, ainsi que sur le modèle relationnel de données. Outre les formalisations et théorèmes particuliers à chaque domaine, ces développements partagent des fondements communs, comme les propriétés des (itérateurs sur les) listes, des fonctions de comparaison totales, et les fonctions booléennes d'égalité ou d'équivalence associées. Un travail utile à la communauté des utilisateurs de Coq, serait de prendre le temps de fabriquer une « contribution » avec ces bases. Ainsi Coccinelle et Datacert seraient de taille plus resserrée, et centrés sur leur objet principal.

Les leçons plus générales que j'ai tirées de ce travail sont que dès lors que l'on cherche à refléter des objets et/ou des algorithmes déjà définis dans des langages de programmation classiques, le mieux est de séparer autant que possible les structures de données des preuves. J'y vois deux avantages :

- L'expérimentation est aisée, puisqu'il n'est pas nécessaire d'écrire des preuves pour fabriquer des objets ;
- Les propriétés qui ont besoin d'hypothèses qui auraient pu s'exprimer dans des types (par exemple, bonne formation des termes) sont précisément identifiées, et pour celles qui n'en n'ont pas besoin, les preuves sont plus légères.

4.2 Procédures de (semi-)décision

Les interactions entre les deux familles de prouveurs automatiques (TPTP et SMT) se sont révélées très fécondes pour combiner le traitement des symboles associatifs et commutatifs et les procédures de décision issues des théories de Shostak. Cependant les résultats obtenus jusqu'à présent ne sont que les prémices d'une moisson bien plus considérable.

4.2.1 Complétion

Je pense étendre, avec Sylvain Conchon et Mohamed Iguernelala, notre travail sur la combinaison des théories de Shostak avec la théorie équationnelle AC. Pour obtenir les résultats de notre article [23], nous avons intégré un solveur et un canoniseur de Shostak dans la complétion AC close. L'étape suivante est d'étendre ce travail à la complétion (modulo AC) en général. Bien sûr, comme la complétion n'est pas une procédure de décision, mais seulement de semi-décision, il est illusoire d'espérer une procédure de décision. Toutefois une procédure de semi-décision serait grandement appréciable, car elle apporterait un traitement en quelque sorte prédéfini et systématique, à l'égalité modulo de théories, au lieu de se reposer sur le mécanisme des déclencheurs, qui est assez aléatoire et encore mal étudié (les travaux de Sylvain Conchon, Claire Dross et Andrei Paskevich sur ce sujet [56] sont précurseurs et très isolés). La correction de cette intégration ne souffre aucun doute, le défi est de démontrer que nous obtenons là une procédure de semi-décision.

4.2.2 Types

Une autre direction est d'utiliser à bon escient les sortes, les types et le polymorphisme, souvent occultés dans la démonstration automatique pour les logiques du premier ordre, au prétexte que les algorithmes/procédures classiques, à partir de termes bien formés/sortés/-typés, ne fabriquent pas de « monstres ». Notre but est de contraindre les procédures de décision à « bien se comporter » lors de leur combinaison. Plus précisément, il est connu depuis une dizaine d'années que, si les canoniseurs de Shostak se combinent, il n'en va pas de même pour les solveurs [80]. Pourtant, nombreux étaient ceux persuadés du contraire, par leur expérience sur des exemples très concrets et pertinents. Notre pari est que dans les exemples réels venant du monde de la programmation, les types peuvent servir de garde-fous, empêcher de se poser des problèmes sans fondements et, une fois les « bons » problèmes posés, servir à les résoudre.

4.3 Terminaison

La réécriture du premier ordre est un modèle de calcul très général, vers lequel peuvent se traduire bon nombre d'autres formalismes plus utilisés en pratique. L'automatisation de la recherche des preuves de terminaison commence à être suffisamment mûre pour penser à l'utiliser dans d'autres cadres. J'envisage deux applications possibles.

4.3.1 Why3

Tout d'abord, parmi les questions qui se posent lors de la vérification déductive de programmes, si le langage hôte autorise des « boucles », de quelque sorte que ce soit, par des fonctions récursives, des constructeurs « for » ou « while », il est naturel de chercher à démontrer leur terminaison pour prouver la « correction totale » des programmes ainsi définis.

Dans la suite d'outils développée autour de Why3, cette démonstration est soit immédiate, soit laissée à la charge de l'utilisateur : les boucles « for » sont bornées et leur terminaison est triviale ; pour la construction « while », l'utilisateur doit fournir un « variant », c'est-à-dire une quantité entière positive qui décroît à chaque itération. En ce qui concerne les fonctions récursives, seules sont autorisées celles dont la terminaison est immédiate par décroissance structurelle des sous-appels.

L'intégration des techniques provenant de la réécriture paraît naturelle dans le cas de la terminaison des fonctions récursives, où l'utilisateur pourrait définir des fonctions dont la terminaison serait moins immédiate, mais toutefois prouvée automatiquement. La principale différence avec la réécriture est que cette dernière autorise tous les chemins de réduction possibles alors que, dans un langage de programmation, même WhyML, le langage de Why3, il y a une stratégie de réduction. Les techniques mises en œuvre par la réécriture peuvent ainsi s'avérer « trop » puissantes ; il est donc utile de s'intéresser plus en détail à la terminaison de la réécriture sous stratégie dans le cadre de cette intégration.

Un autre angle d'attaque concerne les boucles « while » et la synthèse automatique des variants. La plupart des techniques sont disponibles du côté de la réécriture : définition d'ordres bien fondés, résolution de contraintes associées. Il faut se pencher sur la définition d'un bon analogue à un membre gauche de règle de réécriture dans le cadre des boucles. Une fois cette définition parachevée, il sera possible de tirer parti de toutes les techniques déjà mises en œuvre.

4.3.2 Coq

Dans l'assistant à la preuve Coq, toutes les fonctions définies doivent être totales et, donc, leur terminaison garantie. C'est l'un des fondements de la théorie sous-jacente ; en effet on peut voir une fonction qui ne termine pas comme la promesse toujours différée de fournir un terme de preuve pour la proposition définie par son type. Il y a actuellement plusieurs mécanismes qui permettent de définir des fonctions récursives totales en Coq. Le plus basique d'entre eux, la construction **Fixpoint**, ne demande aucune preuve de terminaison à l'utilisateur, simplement parce que les appels récursifs doivent être structurellement décroissants. La seconde construction, **Function**, autorise des appels récursifs plus généraux mais l'utilisateur doit alors fournir une mesure qui décroît ou, plus généralement, un ordre bien fondé et une preuve que les appels sont bien décroissants. Enfin, la construction **Program** permet de restreindre le domaine de définition d'une fonction, non pas à tous les habitants d'un type, mais à tous ceux qui vérifient une certaine propriété et, comme avec **Function**, l'utilisateur peut choisir une mesure ou un ordre bien fondé pour prouver la terminaison.

Ces mécanismes sont soit très rigides, soit lourds à utiliser. Une large partie des fonctions définies en Coq pourraient bénéficier d'une preuve de terminaison trouvée et vérifiée

automatiquement, par la technique utilisée pour certifier la terminaison de systèmes de réécriture au premier ordre. Pour internaliser complètement une telle preuve, le plus commode est d'avoir une preuve par réflexion, c'est-à-dire une preuve qui se ramène *par calcul* à un énoncé trivial, en général true = true. J'envisage de travailler avec Julien Forest, l'un des contributeurs principaux de **Function** à cette « réification » et à cette intégration.

4.4 Langages et systèmes « centrés données »

Le but à long terme de mon travail avec Véronique Benzaken est de vérifier les systèmes de gestion de base de données avec l'assistant Coq et l'environnement de vérification de programmes Why3 [58]. Nous envisageons ainsi d'étendre notre travail dans plusieurs directions : tout d'abord, pour la partie spécification, nous allons formaliser la théorie de la normalisation (au sens bases de données, et non pas réécriture) utile à la conception des schémas. En utilisant Datacert, la bibliothèque que nous avons développée, nous allons vérifier un compilateur et un optimiseur SQL par rapport à notre spécification. Nous traiterons également les mises à jour et les « déclencheurs », ainsi que les aspects de sécurité et de protection de la vie privée. Enfin, le traitement de la gestion des transactions, le contrôle de concurrence et la reprise sur panne sont probablement les défis les plus ambitieux que nous aurons à relever dans le relationnel.

Une seconde direction consiste à étudier et formaliser, au sein de Datacert, d'autres modèles de données, décrits par des langages tels que NoSQL, XML, NewSQL. La modularité et la flexibilité de notre bibliothèque devrait faciliter cette tâche, en autorisant la réutilisation d'un large pan de modélisation.

4.5 Épilogue

Doit-on parler de « facettes de la preuve » qui lancent différents éclats de façon quasi indépendante, ou d'un continuum linéaire qui irait d'une preuve complètement automatisée [1] à une preuve entièrement (d)écrite par un utilisateur, avec un curseur permettant de régler quelles sont les parties traitées automatiquement, et quelles sont celles qui requièrent une véritable imagination. Bien sûr que je ne peux qu'être du second côté. La démonstration automatique fédérait à ses tous débuts, selon Martin Davis, des tenants de la reproduction de l'activité intelligente humaine – intelligence artificielle – et ceux de l'application systématique d'algorithmes/procédures de décision conçus par une intelligence humaine mais pouvant s'en détacher, devenant ainsi des « Idées » platoniciennes. L'intelligence artificielle et le champ actuel de la démonstration automatique se sont depuis longtemps séparés mais il ne fait aucun doute que la fantaisie humaine est plus puissante que l'aridité systématique des machines. Je crois profondément en une interaction harmonieuse au sein de la démonstration mécanisée entre des preuves faciles et complètement automati/ques/sables et une partie créative, laissée à l'intelligence véritable.

1. une fois bien sûr le programme écrit par un (groupe d') humain(s).

Bibliographie

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Farid Ajili and Évelyne Contejean. Complete solving of linear diophantine equations and inequations without adding variables. In Ugo Montanari and Francesca Rossi, editors, *Proc. First International Conference on Principles and Practice of Constraint Programming*, volume 976 of *Lecture Notes in Computer Science*, pages 1–17, Cassis, France, September 1995. Springer.

[3] Farid Ajili and Évelyne Contejean. Avoiding slack variables in the solving of linear diophantine equations and inequations. *Theoretical Computer Science*, 173(1) :183–208, February 1997.

[4] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Thery, and Benjamin Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In Jean-Pierre Jouannaud and Zhong Shao, editors, *CPP - Certified Programs and Proofs - First International Conference - 2011*, volume 7086 of *Lecture notes in computer science - LNCS*, pages 135–150, Kenting, Taïwan, Province De Chine, December 2011. Springer.

[5] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending Coq with Imperative Features and its Application to SAT Verification. In *Interactive Theorem Proving*, Edinburgh, Royaume-Uni, 2010. This work was supported in part by the french ANR DECERT initiative.

[6] Mickaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Wener. Verifying SAT and SMT in Coq for a fully automated decision procedure. In *PSATTT'11 : International Workshop on Proof-Search in Axiomatic Theories and Type Theories*, Wroclaw, Poland, 2011. Germain Faure, Stéphane Lengrand, Assia Mahboubi.

[7] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236 :133–178, 2000.

[8] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[9] Leo Bachmair, editor. *11th International Conference on Rewriting Techniques and Applications*, volume 1833 of *Lecture Notes in Computer Science*, Norwich, UK, July 2000. Springer.

[10] Véronique Benzaken, Évelyne Contejean, and Stefania Dumbrava. A Coq Formalization of the Relational Data Model. In Zhong Shao, editor, *ESOP*, Lecture Notes in Computer Science, Grenoble, April 2014. Springer.

[11] Garrett Birkhoff. On the structure of abstract algebras. In *Proc. Cambridge Phil. Society, 31*, 1935.

[12] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover, 2008. http://alt-ergo.lri.fr/.

[13] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Assia Mahboubi, Alain Mebsout, and Guillaume Melquiond. A Simplex-based extension of Fourier-Motzkin for solving linear integer arithmetic. In Bernhard Gramlich, Dale Miller, and Ulrike Sattler, editors, *IJCAR 2012 : Proceedings of the 6th International Joint Conference on Automated Reasoning*, volume 7364 of *Lecture Notes in Computer Science*, pages 67–81, Manchester, UK, June 2012. Springer.

[14] François Bobot, Sylvain Conchon, Évelyne Contejean, and Stéphane Lescuyer. Implementing Polymorphism in SMT solvers. In Clark Barrett and Leonardo de Moura, editors, *SMT 2008 : 6th International Workshop on Satisfiability Modulo*, volume 367 of *ACM International Conference Proceedings Series*, pages 1–5, 2008.

[15] François Bobot and Andrei Paskevich. Expressing Polymorphic Types in a Many-Sorted Language. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium, Proceedings*, volume 6989 of *Lecture Notes in Computer Science*, pages 87–102, Saarbrücken, Germany, October 2011.

[16] Alexandre Boudet, Évelyne Contejean, and Hervé Devie. A new AC-unification algorithm with a new algorithm for solving diophantine equations. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 289–299, Philadelphia, Pennsylvania, USA, June 1990. IEEE Comp. Soc. Press.

[17] Nicolas Bourbaki. *Eléments de mathématique ; théorie des ensembles*, chapter IV. Hermann, Paris, 1970.

[18] Robert S. Boyer and J. Strother Moore. Proving Theorems about LISP Functions. *Journal of the ACM*, 22(1) :129–144, 1975.

[19] Adam Chlipala, J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In Graham Hutton and Andrew P. Tolmach, editors, *ICFP*, pages 79–90. ACM, 2009.

[20] Sylvain Conchon and Évelyne Contejean. The Alt-Ergo automatic theorem prover. http://alt-ergo.lri.fr/, 2008. APP deposit under the number IDDN FR 001 110026 000 S P 2010 000 1000.

[21] Sylvain Conchon, Évelyne Contejean, and Mohamed Iguernelala. Ground Associative and Commutative Completion Modulo Shostak Theories. In Andrei Voronkov, editor, *LPAR, 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, EasyChair Proceedings, Yogyakarta, Indonesia, October 2010. (short paper).

[22] Sylvain Conchon, Évelyne Contejean, and Mohamed Iguernelala. Canonized Rewriting and Ground AC Completion Modulo Shostak Theories. In Parosh A. Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 45–59, Saarbrücken, Germany, April 2011. Springer.

[23] Sylvain Conchon, Évelyne Contejean, and Mohamed Iguernelala. Canonized rewriting and ground AC completion modulo Shostak theories : Design and implementation. *Logical Methods in Computer Science*, 8(3) :1–29, September 2012. Selected Papers of the Conference *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS 2011), Saarbrücken, Germany, 2011.

[24] Sylvain Conchon, Évelyne Contejean, and Johannes Kanig. CC(X) : Efficiently combining equality and solvable theories without canonizers. In Sava Krstic and Albert Oliveras, editors, *SMT 2007 : 5th International Workshop on Satisfiability Modulo*, 2007.

[25] Sylvain Conchon, Évelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. Lightweight Integration of the Ergo Theorem Prover inside a Proof Assistant. In John Rushby and N. Shankar, editors, *Proceedings of the second workshop on Automated formal methods*, pages 55–59. ACM Press, 2007.

[26] Sylvain Conchon, Évelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. CC(X) : Semantical combination of congruence closure with solvable theories. In *Post-proceedings of the 5th International Workshop on Satisfiability Modulo Theories (SMT 2007)*, volume 198(2) of *Electronic Notes in Computer Science*, pages 51–69. Elsevier Science Publishers, 2008.

[27] Évelyne Contejean. Solving linear diophantine constraints incrementally. In David S. Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, Logic Programming, pages 532–549, Budapest, Hungary, June 1993. MIT Press.

[28] Évelyne Contejean. A certified AC matching algorithm. In van Oostrom [110], pages 70–84.

[29] Évelyne Contejean. Coccinelle, 2005. `http://www.lri.fr/~contejea/Coccinelle/coccinelle.html`.

[30] Évelyne Contejean. Modelling permutations in Coq for Coccinelle. In Hubert Comon-Lundth, Claude Kirchner, and Hélène Kirchner, editors, *Rewriting, Computation and Proof*, volume 4600 of *Lecture Notes in Computer Science*, pages 259–269. Springer, 2007. Jouannaud Festschrift.

[31] Évelyne Contejean. Coccinelle, a Coq library for rewriting. In *Types*, Torino, Italy, March 2008.

[32] Évelyne Contejean and Pierre Corbineau. Reflecting proofs in first-order logic with equality. In Robert Nieuwenhuis, editor, *20th International Conference on Automated Deduction (CADE-20)*, volume 3632 of *Lecture Notes in Artificial Intelligence*, pages 7–22, Tallinn, Estonia, July 2005. Springer.

[33] Évelyne Contejean, Pierre Courtieu, Julien Forest, Andrei Paskevich, Olivier Pons, and Xavier Urbain. A3PAT, an Approach for Certified Automated Termination Proofs. In John P. Gallagher and Janis Voigtländer, editors, *Partial Evaluation and Program Manipulation*, pages 63–72, Madrid, Spain, January 2010. ACM Press.

[34] Évelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Certification of automated termination proofs. In Boris Konev and Frank Wolter, editors, *6th International Symposium on Frontiers of Combining Systems (FroCos 07)*, volume 4720 of *Lecture Notes in Artificial Intelligence*, pages 148–162, Liverpool,UK, September 2007. Springer.

[35] Évelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. CiME3, 2007. http://cime.lri.fr.

[36] Évelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Automated Certified Proofs with CiME3. In Manfred Schmidt-Schauß, editor, *22nd International Conference on Rewriting Techniques and Applications (RTA 11)*, volume 10 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21–30, Novi Sad, Serbia, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[37] Évelyne Contejean and Hervé Devie. An efficient algorithm for solving systems of diophantine equations. *Information and Computation*, 113(1) :143–172, August 1994.

[38] Évelyne Contejean and Claude Marché. CiME : Completion Modulo E. In Ganzinger [60], pages 416–419. System Description available at http://cime.lri.fr/.

[39] Evelyne Contejean, Claude Marché, Benjamin Monate, and Xavier Urbain. CiME version 2, 2000. Available at http://cime.lri.fr/.

[40] Évelyne Contejean, Claude Marché, Benjamin Monate, and Xavier Urbain. Proving termination of rewriting with CiME. In Rubio [102], pages 71–73. Technical Report DSIC II/15/03, Universidad Politécnica de Valencia, Spain.

[41] Évelyne Contejean, Claude Marché, Ana Paula Tomás, and Xavier Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4) :325–363, 2005.

[42] Évelyne Contejean and Xavier Urbain. The A3PAT approach. In *Workshop on Certified Termination WScT08*, Leipzig, Germany, May 2008.

[43] Pierre Corbineau. *Démonstration Automatique en Théorie des Types*. Thèse de doctorat, Université Paris-Sud, September 2005.

[44] Jean-François Couchot and Stéphane Lescuyer. Handling polymorphism in automated deduction. In *21th International Conference on Automated Deduction (CADE-21)*, volume 4603 of *LNCS (LNAI)*, pages 263–278, Bremen, Germany, July 2007.

[45] Certification problem format. http://cl-informatik.uibk.ac.at/software/cpf/.

[46] Max Dauchet. Simulation of a turing machine by a left-linear rewrite rule. In *Proc. 3rd Rewriting Techniques and Applications, Chapel Hill, LNCS 355*, 1989.

[47] Martin Davis. The early history of automated deduction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 1, pages 3–15. Elsevier Science, 2001.

[48] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, 1962.

[49] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3) :201–215, July 1960.

[50] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. http://research.microsoft.com/projects/z3/.

[51] Nachum Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3) :279–301, March 1982.

[52] Nachum Dershowitz. Termination dependencies. In Rubio [102], pages 27–30. Technical Report DSIC II/15/03, Universidad Politécnica de Valencia, Spain.

[53] Nachum Dershowitz and Jean-Pierre Jouannaud. Notations for rewriting. *EATCS Bulletin*, 43 :162–172, 1990.

[54] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8) :465–476, August 1979.

[55] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4) :771–785, 1980.

[56] Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Reasoning with triggers. Research Report RR-7986, INRIA, June 2012.

[57] The ELAN system. http://elan.loria.fr/.

[58] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[59] Olivier Fissore, Isabelle Gnaedig, and Hélène Kirchner. CARIBOO : A multi-strategy termination proof tool based on induction. In Albert Rubio, editor, *Proceedings of the 6th International Workshop on Termination 2003*, pages 77–79, Valencia (Spain), June 2003.

[60] Harald Ganzinger, editor. *7th International Conference on Rewriting Techniques and Applications*, volume 1103 of *Lecture Notes in Computer Science*, New Brunswick, NJ, USA, July 1996. Springer.

[61] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. Aprove 1.2 : Automatic termination proofs in the dependency pair framework. In Ulrich Furbach and Natarajan Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR '06)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 281–286, Seattle, USA, August 2006. SV.

[62] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automated Reasoning*, 37(3) :155–203, 2006.

[63] Paul C. Gilmore. A program for the production from axioms, of proofs for theorems derivable within the first order predicate calculus. In *IFIP Congress*, pages 265–272, 1959.

[64] Kurt Gödel. Uber formal unentscheidbare Satze der Principia Mathematica und verwandter Systeme, 1931.

[65] Carlos Gonzalia. Towards a formalisation of relational database theory in constructive type theory. In Rudolf Berghammer, Bernhard Möller, and Georg Struth, editors, *RelMiCS*, volume 3051 of *Lecture Notes in Computer Science*, pages 137–148. Springer, 2003.

[66] Carlos Gonzalia. *Relations in Dependent Type Theory*. PhD thesis, Chalmers Göteborg University, 2006.

[67] George Grätzer. *Universal Algebra*. Springer, second edition, 1979.

[68] Jacques Herbrand. Recherches sur la théorie de la démonstration. Thèse d'Etat, Univ. Paris, 1930. Also in : Ecrits logiques de Jacques Herbrand, PUF, Paris, 1968.

[69] Nao Hirokawa and Aart Middeldorp. Tyrolean termination tool : Techniques and features. *ic*, 205(4) :474–511, 2007.

[70] Gérard Huet. A complete proof of correctness of the Knuth-Bendix completion algorithm. *Journal of Computer and System Sciences*, 23 :11–21, 1981.

[71] Gérard Huet and Dallas S. Lankford. On the uniform halting problem for term rewriting systems. Research Report 283, INRIA, March 1978.

[72] Jean-Marie Hullot. Associative commutative pattern matching. In *Proc. 6th IJCAI (Vol. I), Tokyo*, pages 406–412, August 1979.

[73] Mohamed Iguernelala. *Strengthening the Heart of an SMT-Solver : Design and Implementation of Efficient Decision Procedures*. Thèse de doctorat, Université Paris-Sud, June 2013.

[74] Jean-Pierre Jouannaud and Albert Rubio. The higher-order recursive path ordering. In Giuseppe Longo, editor, *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, Trento, Italy, July 1999. IEEE Comp. Soc. Press.

[75] Sam Kamin and Jean-Jacques 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.

[76] Claude Kirchner. Méthodes et outils de conception systématique d'algorithmes d'unification dans les théories equationnelles. Thèse d'Etat, Univ. Nancy, France, 1985.

[77] Claude Kirchner and Francis Klay. Syntactic theories and unification. In *Proc. 5th IEEE Symp. Logic in Computer Science, Philadelphia*, June 1990.

[78] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[79] Adam Koprowski. Certified Higher-Order Recursive Path Ordering. In Frank Pfenning, editor, *17th International Conference on Rewriting Techniques and Applications (RTA'06)*, volume 4098 of *Lecture Notes in Computer Science*, pages 227–241, Seattle, USA, August 2006. Springer.

[80] Sava Krstić and Sylvain Conchon. Canonization for disjoint unions of theories. *Information and Computation*, 199(1-2) :87–106, May 2005.

[81] Dallas 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.

[82] Jean Largeault. *Logique Mathématique, textes*. Collection U. Armand Colin, 1972.

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

[84] Pierre Lescanne. Implementation of completion by transition rules + control : Orme. In *Proceedings of the Second International Conference on Algebraic and Logic Programming, LNCS 463*, October 1990.

[85] Stéphane Lescuyer. *Formalisation et développement d'une tactique réflexive pour la démonstration automatique en Coq*. Thèse de doctorat, Université Paris-Sud, January 2011.

[86] Salvador Lucas. Mu-term : A tool for proving termination of context-sensitive rewriting. In van Oostrom [110], pages 200–209.

[87] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *ACM International Conference on Principles of Programming Languages (POPL)*, 2010.

[88] Zohar Manna and Stephen Ness. On the termination of markov algorithms. In *Proc. third Hawaii International Conference on Systems Sciences*, pages 789–792, Honolulu, HI, 1970. http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html.

[89] Yuri V. Matiyasevich. Enumerable sets are diophantine. *Soviet Mathematics (Dokladi)*, 11(2) :354–357, 1970.

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

[91] R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors. *Selected Papers on Automath (Studies in Logic and the Foundations of Mathematics)*, volume 133 of *Studies in Logic and the Foundations of Mathematics,*. Elsevier Science, 1994.

[92] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27 :356–364, 1980.

[93] Allen Newell, J. C. Shaw, and Herbert A. Simon. Report on a general problem-solving program. In *IFIP Congress*, pages 256–264, 1959.

[94] Tobias Nipkow. Proof transformations for equational theories. In *Proc. 5th IEEE Symp. Logic in Computer Science, Philadelphia*, June 1990.

[95] Enno Ohlebusch, Claus Claves, and Claude Marché. TALP : A tool for the termination analysis of logic programs. In Bachmair [9], pages 270–273. Available at http://bibiserv.techfak.uni-bielefeld.de/talp/.

[96] David A. Plaisted. A recursively defined ordering for proving termination of term rewriting systems. Unpublished report R-78-943, University of Illinois, Urbana, IL, 1978.

[97] Dag Prawitz, Haåkan Prawitz, and Neri Voghera. A mechanical proof procedure and its realization in an electronic computer. *J. ACM*, 7(2) :102–128, 1960.

[98] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[99] Renato Iturriga. *Contributions to Mechanical Mathematics*. PhD thesis, Carnegie-Mellon University, 1967.

[100] John A. Robinson. Automatic Deduction with Hyper-Resolution. *International Journal of Computer Mathematics*, 1 :227–234, 1965.

[101] John A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1) :23–41, 1965.

[102] Albert Rubio, editor. *Extended Abstracts of the 6th International Workshop on Termination, WST'03*, June 2003. Technical Report DSIC II/15/03, Universidad Politécnica de Valencia, Spain.

[103] Dana S. Scott. A type-theoretical alternative to iswim, cuch, owhy. *Theoretical Computer Science*, 121(1&2) :411–440, 1993.

[104] Geoff Sutcliffe, Christian Suttner, and Theodor Yemenis. The TPTP problem library. In Alan Bundy, editor, *Proc. 12th Conference on Automated Deduction CADE, Nancy/France*, pages 252–266. Springer-Verlag, 1994.

[105] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, 2008. http://coq.inria.fr.

[106] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 42(2) :230–265, 1936.

[107] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 43 :544–546, 1937.

[108] Jeffrey D. Ullman. *Principles of Database Systems, 2nd Edition*. Computer Science Press, 1982.

[109] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science : Volume B : Formal Models and Semantics*, volume B. Elsevier, Amsterdam, 1990.

[110] Vincent van Oostrom, editor. *15th International Conference on Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, Aachen, Germany, June 2004. Springer.

[111] Laurent Vigneron. *Déduction automatique avec contraintes symboliques dans les théories equationnelles*. Thèse de doctorat, Université Henri Poincaré Nancy 1, 1994.

[112] Laurent Vigneron. DATAC, 1998. http://www.loria.fr/equipes/cassis/softwares/daTac/.

[113] Alfred North Whitehead. *A treatise on universal algebra, with applications*. Cambridge : The University Press, 1898.

[114] Alfred North Whitehead and Bertrand Arthur William Russell. *Principia mathematica ; 2nd ed.* Cambridge University Press, Cambridge, 1927.

Table des matières

1 Liminaire **3**
1.1 Comment la rhétorique s'est faite machine . . . 3
1.2 Preuves mécanisées . . . 5
1.2.1 Preuve assistée . . . 5
1.2.2 Démonstration automatique . . . 7
1.3 Organisation de ce mémoire . . . 9

2 Survol de mes contributions récentes **11**
2.1 Procédures de décision . . . 12
2.1.1 Polymorphisme . . . 14
2.1.2 Combinaison . . . 16
2.1.3 Vérification formelle . . . 16
2.1.4 Arithmétique . . . 17
2.1.5 Valorisation . . . 17
2.2 Preuve automatique ou assistée ? . . . 18
2.2.1 Égalité et filtrage . . . 20
2.2.2 Terminaison . . . 21
2.3 Vers des données certifiées . . . 23
2.3.1 Modèle relationnel de données . . . 23
2.3.2 Formalisations existantes . . . 24
2.3.3 Datacert . . . 24

3 Algèbres de termes **27**
3.1 Préliminaires et résultats classiques . . . 28
3.1.1 Termes . . . 28
3.1.2 Substitutions . . . 30
3.1.3 Théorie équationnelle . . . 30
3.1.4 Réécriture . . . 32
3.1.5 Terminaison de $\underset{R}{\rightarrow}$ . . . 34
3.2 Implanter et formaliser : C*i*ME et Coccinelle . . . 36
3.2.1 Comment implanter une algèbre universelle ? . . . 36
3.2.2 Modéliser la réécriture et les étapes RDP . . . 40
3.3 Un éclairage sur RPO . . . 42
3.3.1 Définition, propriétés et formalisation . . . 42

3.3.2 Comparaison effective des termes 44
3.3.3 Quelques chiffres 46
3.4 Conclusion 46
3.4.1 Statistiques 46
3.4.2 Liens entre outils automatiques et formalisation 47
3.4.3 Dissémination 47
3.4.4 Futur 48

4 Perspectives 49
4.1 Ingénierie de la preuve 49
4.2 Procédures de (semi-)décision 50
4.2.1 Complétion 50
4.2.2 Types 50
4.3 Terminaison 50
4.3.1 Why3 51
4.3.2 Coq 51
4.4 Langages et systèmes « centrés données » 52
4.5 Épilogue 52

Bibliographie 53