Programmation et confiance

Pierre-Etienne Moreau

Institut National
Polytechnique de Lorraine

Département de formation doctorale en informatique

École doctorale IAEM Lorraine

Programmation et confiance

Habilitation à Diriger des Recherches

présentée et soutenue publiquement le 13 juin 2008

par

Pierre-Etienne Moreau

Composition du jury

Rapporteurs :	Hassan Aït-Kaci	Chercheur émérite, ILOG, Inc., Sunnyvale, CA (USA)
	Gilles Dowek	Professeur, École Polytechnique, Palaiseau
	Jean-Louis Giavitto	Directeur de Recherche, CNRS, Evry
Examinateurs :	Yves Caseau	Directeur Général Adjoint, Bouygues Telecom, Paris
	Charles Consel	Professeur, ENSEIRB, Bordeaux
	Claude Kirchner	Directeur de Recherche, INRIA, Bordeaux
	Karl Tombre	Professeur, École des Mines de Nancy, INPL

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Sommaire

1

Le tour en 80 lignes

Ce manuscrit présente les travaux de recherche, d'encadrement et de développement que j'ai effectués au cours des huit dernières années. J'aime chercher, concevoir des solutions, écrire des programmes. J'attache une importance particulière à la façon dont un programme est écrit. J'apprécie notamment qu'un programme soit facile à lire, à comprendre, et que son écriture semble naturelle. Il me parait également important qu'un programme puisse évoluer, être modifié, sans pour autant entraîner de nouvelles erreurs. Pour cela, les schémas, les « assemblages » utilisés lors de la conception et l'écriture du programme sont déterminants. Pour pouvoir proposer et utiliser de bons schémas, flexibles, robustes, il faut que le langage offre des constructions à la fois expressives et ayant de bonnes propriétés. Je me suis naturellement intéressé à l'étude des langages de programmation, à leur pouvoir expressif, à leur sémantique, ainsi qu'à la façon de les rendre opérationnels, c.-à-d. à l'étude et à la construction de compilateurs.

Le chapitre 2 est introductif. Il présente le contexte de mes recherches, mon intérêt pour les langages fondés sur la notion de réécriture, ainsi que ma volonté de rendre utilisables et de diffuser les concepts développés autour de cette notion.

Le chapitre 3 introduit le langage Tom, qui est une extension de langages existants, C, Java ou Python par exemple, ajoutant des constructions de filtrages inspirées de la réécriture et de la programmation fonctionnelle. Cette approche consistant à étendre un langage hôte plutôt que de définir un nouveau langage est un moyen de promouvoir des méthodes formelles : cela fournit de nouvelles constructions aux programmeurs sans pour autant changer leur environnement de travail.
Nous introduisons ensuite la notion d'« îlot formel » qui est une généralisation de cette approche, formalisant l'extension d'un langage quelconque par de nouvelles constructions. Dans ce contexte, l'idée d'« ancrage formel » joue un rôle majeur. Cela permet de relier les constructions introduites et les objets manipulés par le langage hôte. Ainsi, dans le cadre de Tom par exemple, les

constructions de filtrage introduites « voient » les objets comme des termes, ce qui permet de filtrer directement vers des objets du langage hôte. Enfin, le langage Tom est présenté comme une instance de ce concept d'îlot formel.

Le chapitre 4 présente deux travaux liés à l'implantation de structures de données. Le premier est une amélioration du gestionnaire mémoire de la bibliothèque des ATerms. Initialement développée au Cwi, elle permet de représenter des termes avec un partage maximum. Nous avons utilisé cet invariant pour proposer un « ramasse-miettes » à générations, dans un cadre conservatif, c.-à-d. ne déplaçant pas les objets. En inspectant plus souvent la mémoire correspondant aux objets « jeunes », et moins souvent celle des objets « vieux », nous obtenons un gain en efficacité compris entre 20% et 35%.
La deuxième contribution est la conception et la réalisation d'un générateur de structures de données typées réduisant le nombre d'erreurs à l'exécution. En introduisant un nouveau patron de conception, appelé `SharedObject`, nous avons abstrait et généralisé le mécanisme de partage des ATerms. Ce patron est disponible sous forme de bibliothèque et est utilisé comme implantation sous-jacente à la structure typée qui est générée.

Le chapitre 5 vise à donner confiance dans les outils utilisés. Dans un premier temps nous présentons une méthode permettant de certifier la correction du code correspondant à la compilation de constructions de filtrage syntaxique. Cette approche est appliquée au compilateur Tom en fournissant des certificats vérifiables avec Coq.
Dans un deuxième temps, nous présentons un nouveau schéma de compilation du filtrage consistant à séparer clairement les étapes de compilation des étapes d'optimisation. L'intérêt principal étant d'avoir une implantation efficace du filtrage dans le contexte des îlot formels. En effet, les symboles de fonction n'ayant pas nécessairement de représentation, l'instruction `switch/case` ne peut plus être utilisée pour implanter la discrimination. L'algorithme d'optimisation est présenté sous la forme d'un système de réécriture dont la correction est montrée.
Dans un troisième temps, nous présentons Gom, une extension du générateur de structures de données décrit précédemment. Une fonctionnalité essentielle est de maintenir les termes en forme canonique en appliquant des invariants lors de la construction. La contribution principale est d'avoir réussi à intégrer ce mécanisme, rappelant la notion de type privé de Caml, dans l'environnement de programmation Java.

Le chapitre 6 présente des extensions qui rendent plus expressif le filtrage et qui permettent de mieux contrôler l'application de règles de réécriture. Dans un premier temps, nous introduisons la notion d'« anti-pattern », qui correspond à la notion de complément sur les termes. Un anti-pattern peut contenir un nombre arbitraire de négations ainsi que des variables non linéaires. Nous en donnons une sémantique précise ainsi qu'un algorithme

de filtrage montré correct, complet, et unitaire dans le cas syntaxique ; cet algorithme est étendu au cas équationnel. Nous donnons également une version plus efficace pour une sous-classe de motifs dans le cas associatif avec élément neutre, qui correspond à de nombreux cas rencontrés en pratique.
Tom propose une notation variadique pour représenter et filtrer des listes. Nous formalisons la correspondance entre cette notation et les termes associatifs avec élément neutre. Nous montrons également que les formes normales modulo associativité et élément neutre sont identiques aux formes normales des termes variadiques correspondants. Ces travaux sont le point de départ pour établir que le filtrage proposé par Tom correspond bien au filtrage associatif avec élément neutre.
Dans une troisième partie nous présentons un langage de stratégie pour Tom, et plus généralement pour tout programme Java. C'est une contribution importante qui permet de contrôler finement la façon dont les règles sont appliquées. Cela permet en particulier de décrire des parcours classiques tels que *leftmost-innermost* ou *breadth-first search* par exemple. Le langage proposé permet des définitions récursives et rend explicite la notion de position dans un terme. Les stratégies sont réifiées au niveau des termes, permettant de filtrer, de construire, et même de transformer dynamiquement une stratégie. Enfin, une stratégie étant un terme, il est possible d'appliquer une stratégie sur une stratégie.

Le chapitre 7 met en évidence que l'ensemble des travaux présentés est implanté et converge vers un système utilisable, contribuant à promouvoir l'utilisation de méthodes formelles dans un cadre aussi bien académique qu'industriel. L'état actuel de Tom est décrit et quelques applications importantes sont présentées.

Contexte :

Je m'intéresse depuis longtemps aux langages de programmation, à leurs caractéristiques, aux avantages qu'ils procurent et aux limites qu'ils imposent à ceux qui les utilisent. Ce n'est donc pas un hasard si aujourd'hui encore je cherche à améliorer les langages existants, en étudiant et en proposant de nouveaux concepts et formalismes.

Depuis le début de ma thèse, je porte une attention particulière aux langages qui permettent de décrire facilement des transformations de structures arborescentes telles que des expressions symboliques, des termes, ou des documents XML.

Mon objectif est de comprendre, et de trouver, quelles constructions élémentaires aident à décrire de telles transformations, tout en donnant au programmeur un certain nombre d'indicateurs ou d'indices lui permettant d'avoir *confiance* dans les programmes qu'il écrit.

Mon post-doctorat effectué sous la direction d'Yves Caseau chez Bouygues Telecom, m'a amené à concevoir et à expérimenter une nouvelle approche pour rendre utilisables les concepts développés autour de la notion de réécriture et pour les diffuser. J'ai en particulier compris que s'il était important de développer du logiciel efficace, il l'était encore plus de le rendre adaptable à son environnement, facile à maintenir et à faire évoluer par un ensemble de personnes. Ce sont ces idées élémentaires qui ont guidé les travaux que je présente dans ce document.

2

Expédition Réécriture

ÉCRIRE **v. tr.** – lat. *scribere* **I**◇ **1**◇ Tracer (des signes d'écriture, un ensemble organisé de ces signes). **2**◇ Consigner, noter par écrit. **3**◇ Rédiger (un message destiné à être envoyé à qqn). **4**◇ INFORM. Transférer (des informations) dans un registre, une mémoire (opposé à *lire*). **II**◇ **1**◇ Composer (un ouvrage scientifique, littéraire). **2**◇ Exprimer de telle ou telle façon sa pensée par le langage écrit. **3**◇ Exposer (une idée) dans un ouvrage.

PROGRAMME **n.m.** – gr. *programma* « ce qui est écrit à l'avance » **1**◇ Écrit annonçant et décrivant les diverses parties d'une cérémonie, d'un spectacle, *etc.* **2**◇ Annonce des matières d'un cours, du sujet d'un concours, d'un prix. **3**◇ Suite d'actions que l'on se propose d'accomplir pour arriver à un résultat. **4**◇ Ensemble ordonné d'opérations effectuées par un système automatique.

[extraits du *Petit Robert de la langue française*, édition 2007]

2.1 Objectifs

J'ai découvert l'informatique en 1982, à une époque où de nombreux modèles de micro-ordinateurs étaient disponibles sur le marché ; des appareils, bien souvent construits autour d'un clavier, un peu comme les ordinateurs portables actuels, mais plus épais. Ils se branchaient sur un moniteur ou une télévision et démarraient rapidement. Ils étaient dotés d'une mémoire morte, appelée ROM, contenant le système d'exploitation et d'une mémoire vive, appelée RAM, allant de un à quelques dizaines de kilo-octets pour les plus puissants.

J'ai passé beaucoup de temps à « programmer » différents modèles de micro-ordinateurs, réalisant ainsi des applications, relativement simples au début, de plus en plus complexes ensuite. L'informatique au début des années 1980 se pratiquait dans un cercle relativement fermé. Il n'y avait pas ou

très peu d'enseignement de la discipline au lycée. Cela a eu l'effet stimulant de m'amener à apprendre et à expérimenter de nombreux langages de programmation. J'ai évidemment fait mes premiers pas en utilisant le langage Basic, mais j'ai pu élargir mes connaissances en essayant tour à tour des variantes plus structurées et d'autres langages tels que Logo, Lse, Forth, Apl, Assembleur, Pascal, C, Ada, Lisp, etc. Au début des années 1990 j'ai découvert la programmation objets en étudiant le langage Eiffel.

Je m'intéresse depuis longtemps aux langages de programmation, à leurs caractéristiques, aux avantages qu'ils procurent et aux limites qu'ils imposent à ceux qui les utilisent. Ce n'est donc pas un hasard si aujourd'hui encore je cherche à améliorer les langages existants, en étudiant et proposant de nouveaux concepts et formalismes.

Un langage de programmation est un système d'expression permettant de décrire des algorithmes sous forme de programmes afin qu'ils soient exécutés par une machine (un ordinateur). C'est donc quelque chose de formel, qui possède une « syntaxe ». Celle-ci permet de distinguer les programmes corrects de ceux incorrects. Par correct, nous entendons ici « syntaxiquement correct », ce qui garantit que le programme est bien formé, qu'il respecte les règles du système d'expression, mais cela ne donne aucune indication sur l'intérêt du programme écrit. Au même titre qu'une phrase exprimée en français correct n'a pas forcément de sens. En plus de sa syntaxe, le deuxième élément qui caractérise un langage de programmation est sa « sémantique ». C'est-à-dire le sens que sont supposés avoir les programmes écrits.

Étant données une syntaxe et une sémantique, on s'intéresse souvent au pouvoir d'expression du langage ainsi décrit. En général, les langages de programmation permettent de décrire une machine de Turing universelle, ce qui leur procure une expressivité théorique équivalente. Cependant, en pratique, un langage donné peut permettre d'exprimer plus facilement une certaine classe de programmes, ce qui se traduit bien souvent par des programmes courts ou faciles à comprendre. Ce type d'expressivité est apprécié parce qu'il permet d'écrire vite de petits programmes. Cependant leur facilité apparente d'utilisation conduit bien souvent à écrire des programmes qui ne font pas ce que l'on pense. En contraignant le programmeur à utiliser certaines constructions, on lui évite généralement de commettre des erreurs, au même titre que les bandes blanches sur les routes réduisent le nombre d'accidents en aidant les conducteurs à mieux se repérer. Concevoir un bon langage de programmation est difficile. Il faut dans un premier temps décrire et étudier sa syntaxe et sa sémantique, de manière à réduire le nombre de programmes incorrects ou ambigus. Mais il faut également prendre en compte les habitudes ou attentes des personnes qui vont l'utiliser, afin de faciliter son utilisation tout en minimisant les possibilités d'écrire des programmes incorrects.

Depuis le début de ma thèse, je m'intéresse à des langages de programmation qui permettent de décrire facilement des transformations de struc-

tures arborescentes telles que des expressions symboliques, des termes, ou des documents XML. Mon objectif est de comprendre, et de trouver, quelles constructions élémentaires aident à décrire de telles transformations, tout en donnant au programmeur un certain nombre d'indicateurs ou d'indices lui permettant d'avoir *confiance* dans les programmes qu'il écrit.

2.2 Préparatifs

Une des ambitions de tout informaticien est d'être sûr que les programmes qu'il écrit soient corrects. Ce qui revient à produire des programmes qui fonctionnent sans erreur tout en ayant un comportement conforme à ce qu'attend l'utilisateur ou le concepteur. Cela sous-entend que le programme a un comportement idéal attendu, correspondant à un cahier de charges ou à une spécification de ses fonctionnalités. Il existe différentes approches pour tenter d'atteindre ce but. L'une d'elles consiste à écrire des jeux de tests, évaluant différentes fonctions du logiciel en les comparant à des résultats attendus. En augmentant le nombre de tests, et surtout leur couverture, on s'assure du bon fonctionnement d'un certain nombre de fonctions, idéalement toutes, du logiciel. Une autre approche consiste à considérer une abstraction du logiciel, c.-à-d. un modèle mathématique, et à en montrer formellement des propriétés. L'avantage de cette approche est qu'elle enlève des doutes : la question n'est plus de savoir si le jeu de tests est suffisamment bien pensé, mais de s'assurer que la preuve des propriétés est correcte. Une propriété prouvée étant par définition *vraie*. Une difficulté rencontrée dans cette approche est qu'une propriété est vraie pour un modèle mathématique donné, mais le passage du modèle à l'implantation n'est pas forcément garanti. L'idéal serait de rapprocher les notions de modèle mathématique et de programme. C'est à mes yeux une des propriétés importantes de la *réécriture*.

La notion de *relation de réécriture* est apparue au cours de l'année 1970, lorsque Donald E. Knuth et Peter B. Bendix ont proposé une méthode permettant de savoir si deux mots composés d'opérateurs et de variables étaient égaux, en présence de propriétés satisfaites par les opérateurs. Cette méthode, appelée plus tard « Complétion de Knuth-Bendix » [KB70], utilise la notion d'égalité orientée, appelée relation de réécriture. Quelques années plus tard, en 1975, la notion de programmation par équations ou par règle de réécriture est apparue suite aux travaux de Joseph A. Goguen et de Michael J. O'Donnell [HO82]. L'intérêt de cette approche est qu'elle permet de programmer un ordinateur en utilisant des règles de réécriture. Ce qui offre la possibilité de raisonner sur le programme écrit, réduisant ainsi l'écart entre programme et modèle mathématique. En utilisant la procédure de complétion de Knuth-Bendix, il devient ainsi possible de s'assurer qu'un programme exprimé par des règles de réécriture a les propriétés de confluence et de terminaison par exemple.

Depuis, ont été conduits de nombreux travaux relatifs aux propriétés des systèmes de réécriture ainsi que de nombreuses implantations d'outils et de langages reposant sur ce principe, OBJ [Gog78] étant certainement le plus connu. Après avoir travaillé avec Joseph A. Goguen et Jose Meseguer sur le langage OBJ [GKK+87], Claude Kirchner et Hélène Kirchner développèrent à partir de 1991 le langage Elan [Vit94], basé sur la réécriture, introduisant pour la première fois la notion de stratégie définie par l'utilisateur.

Mes premiers travaux de recherche, dans le cadre du stage de DEA, ont porté sur l'étude et l'implantation d'une procédure de complétion de Knuth-Bendix utilisant des contraintes pour modéliser les problèmes d'unification [KM95]. Ce fut l'un des premiers programmes écrits en Elan. Il fonctionne encore à ce jour.

Participant à la réalisation d'un environnement permettant à la fois de programmer et de prouver les programmes, au début de ma thèse j'ai étudié de quelle façon le langage Elan pouvait être étendu pour offrir des fonctionnalités réflexives [KM96]. Je me suis ensuite concentré sur le développement de méthodes de compilation visant à rendre efficace l'exécution de programmes exprimés par des règles et des stratégies. Une particularité d'Elan est d'offrir un langage de stratégie puissant [Bor98], permettant de choisir de manière non-déterministe des règles à appliquer. Cette approche permet de décrire de façon élégante l'exploration d'un espace de recherche, mais présente des difficultés d'implantation proches de celles rencontrées dans les implantations de langages tels que Prolog [War83, AK90]. Une deuxième particularité est de supporter la réécriture modulo les théories associatives et commutatives, ce qui rend plus difficiles les opérations de filtrage pour savoir si une règle peut s'appliquer : savoir si une solution d'un problème de filtrage existe est simplement NP-complet [KN86]. Par ailleurs, le nombre de solutions d'un problème donné est exponentiel en la taille du terme filtré [BKN87], ce qui ajoute une deuxième source de non-déterminisme au choix des réécritures à effectuer.

Outre l'implantation et la diffusion d'un compilateur pour le langage Elan, mes principales contributions ont été la définition d'un algorithme permettant de factoriser la compilation de plusieurs règles impliquant des symboles associatifs-commutatifs, ainsi que la conception d'un algorithme d'analyse du non-déterminisme d'une stratégie [KM01]. L'intérêt du premier est de réduire la redondance des calculs. L'intérêt du deuxième est de détecter certaines conditions dans lesquelles un calcul est déterministe, ce qui a pour effet de permettre l'utilisation d'algorithmes de filtrages incomplets, ne calculant qu'une seule solution, mais de manière beaucoup plus efficace, c'est à dire linéaire en général. La combinaison de ces différentes approches a donné lieu à des algorithme efficaces, mais particulièrement difficiles à implanter.

Les choix faits ainsi que les solutions proposées dans le cadre de ma thèse étaient raisonnables et motivés par l'envie de rendre efficace et utilisable un

langage expressif reposant essentiellement sur la notion de réécriture.

Mon post-doctorat effectué chez Bouygues Telecom sous la direction d'Yves Caseau, ainsi que mes réflexions sur la façon de développer un logiciel, m'ont amené à concevoir et à expérimenter une nouvelle approche pour rendre utilisables les concepts développés autour de la notion de réécriture. J'ai en particulier compris que s'il était important de développer du logiciel efficace, il l'était encore plus de le rendre adaptable à son environnement, facile à maintenir et à faire évoluer par un ensemble de personnes. Ce sont ces idées élémentaires qui ont guidé les travaux que je présente dans ce document.

2.3 Départ

Qui n'a jamais rêvé d'être aventurier, photographe, ou de réaliser un tour du monde ? Autrement dit, qui n'a jamais rêvé d'être chercheur ?

En septembre 2000, j'ai été recruté en tant que chargé de recherche à l'Inria pour continuer à développer des recherches autour des langages de programmation et des méthodes formelles, et plus généralement sur la sûreté et la sécurité des programmes que l'on peut écrire. C'est cette aventure que je vais vous décrire.

Contexte :

Aujourd'hui, de nombreuses personnes écrivent des programmes. Il est toujours difficile de s'assurer que les programmes ainsi écrits fonctionnent bien et répondent aux attentes des auteurs.

L'utilisation de méthodes formelles permet de prendre du recul, de faire abstraction du langage d'implantation, et de s'intéresser au comportement que doit avoir le programme une fois écrit. Cela implique des étapes de réflexion, de spécification, de formalisation, de preuve, et naturellement de maintenance. En retour, on peut espérer avoir pensé aux problèmes avant qu'ils ne se présentent.

Cette approche a néanmoins des inconvénients. Bien qu'elle se justifie facilement lors du développement de logiciels critiques mettant en jeu des vies humaines, on lui reproche souvent de demander un effort supplémentaire et de ne donner que peu de garanties lors du passage de la spécification formelle à l'implantation réelle.

Effectuant mes recherches dans ce domaine lié à la qualité des logiciels, voici les questions que je me suis posées.

Questions :

- Comment promouvoir et rendre les méthodes formelles plus facilement utilisables ?
- Comment rendre ré-utilisables les outils que nous développons ?
- Comment réduire les efforts d'implantation d'un nouveau langage ?

Contributions :

J'ai participé au projet Elan en concevant et en réalisant le compilateur, puis j'ai défini un format d'échange permettant à différents composants de coopérer. Avec Mark G. J. VAN DEN BRAND, nous avons rendu générique l'environnement Asf+Sdf afin de diminuer les efforts d'implantation d'environnements futurs. Cela a permis de réutiliser facilement les composants du méta-environnement pour en construire un spécifique à Elan.

En parallèle à ces travaux, j'ai essayé de répondre à la première question en proposant un langage de programmation qui intègre des constructions provenant des méthodes formelles, tout en évitant de changer le cadre de travail des développeurs. J'ai ainsi défini le concept d'*îlot formel*, qui correspond à l'intégration de nouvelles constructions dans un langage existant. Une contribution importante est l'identification et la définition de la notion d'*ancrage formel*, qui permet de relier les objets introduits par les nouvelles constructions à ceux du langage hôte.

Ma deuxième contribution est la définition du langage TOM, instance du concept d'îlot formel, qui ajoute des constructions de filtrage, syntaxiques et associatives, dans différents langages dont C, Java et Caml. L'expressivité et la puissance du langage proviennent en grande partie de la notion d'ancrage formel, rendant les constructions introduites indépendantes de toute implantation. Cela permet d'intégrer des constructions de filtrage dans des applications existantes, rendant ainsi plus facile l'utilisation d'idées et d'outils issus des méthodes formelles.

3

Îlots Formels

ÎLOT **n.m.** – de *île* **1**◇ Très petite île **2**◇ Petit espace isolé dans un ensemble d'une autre nature.

FORMEL adj. – lat. *formalis*, de *forma* → forme **1**◇ Dont la précision et la netteté excluent toute méprise, toute équivoque.

3.1 Tectonique

La notion de relation de réécriture est connue et utilisée par de nombreux chercheurs proches des méthodes formelles et du calcul symbolique. Cependant, le nombre de personnes utilisant l'outil réécriture pour prototyper ou développer leurs travaux est encore relativement faible par rapport à d'autres communautés, telles que celles liées à la programmation logique, à la programmation fonctionnelle, ou à la programmation objets. Nous ne sommes également qu'un petit nombre à développer des outils tels qu'Elan, Maude, Stratego, Asf+Sdf, Mgs, Txl, Hats, etc. Il est par conséquent primordial de s'organiser pour diminuer les efforts d'implantation.

Une solution assez naturelle consiste à rendre nos logiciels suffisamment génériques, inter-opérables et réutilisables pour pouvoir les partager entre les différents groupes de recherche. Cette tâche est bien souvent beaucoup plus difficile qu'elle n'y parait parce qu'il faut discerner, pour chaque caractéristique, celles qui peuvent être généralisées et celles qui sont propres à nos besoins. Mes premiers pas dans cette direction ont débuté lors de la conception du compilateur d'Elan en 1995. Il fallait « connecter » l'interpréteur (contenant le parseur) avec le compilateur. Ces deux outils étant écrits dans des langages différents (C++ pour le premier et Java pour le second), nous avons introduit un *format d'échange* appelé Ref (*Reduce Elan Format*) [BJMR98]

[BJMR98] Peter Borovanský, Salma Jamoussi, Pierre-Etienne Moreau, and Christophe Ringeissen. Handling ELAN rewrite programs via an exchange format. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 2nd In-*

permettant de représenter un programme Elan. Ce format textuel a été utilisé pour échanger des programmes Elan entre l'interpréteur et le compilateur dans un premier temps. Plus généralement, cela permettait de faire coopérer différents outils d'analyse ou de transformation de programmes Elan, et par conséquent d'étendre nos environnements de programmation en y connectant des outils de vérification de confluence ou de terminaison par exemple. Il est important de noter qu'un programme encodé en Ref est représenté par un terme, ce qui permet d'utiliser tout outil fondé sur la réécriture pour effectuer des traitements. L'autre intérêt de cette approche était de définir clairement une interface d'entrée du compilateur Elan, le rendant ainsi réutilisable pour compiler tout autre langage se rapprochant d'Elan [Mor00].

En 1998, j'ai été invité à travailler dans l'équipe de Paul Klint, au Cwi, qui développe le méta-environnement Asf+Sdf [Kli93], pour y développer un « connecteur » permettant de compiler des spécifications Asf en utilisant le compilateur Elan par l'intermédiaire du format Ref. Cette équipe s'intéresse depuis longtemps, avec un certain succès, aux problèmes d'échanges et de réutilisation. Outre le bus de coordination, appelé ToolBus [BK98], fondé sur une algèbre de processus, j'y ai découvert une technologie d'échange basée sur un format, appelé asFix, capable d'encoder non seulement la sémantique d'un programme, mais également sa représentation concrète (l'ensemble des caractères, espaces et tabulations composant le programme d'origine). Cela se révèle être essentiel pour faire de la transformation de programmes préservant la mise en forme, ainsi que pour faire remonter des messages d'erreurs en lien avec l'entrée.

Dans le cadre de l'équipe associée « AirCube » liant l'équipe de Nancy dirigée par Claude Kirchner à celle d'Amsterdam, dirigée par Paul Klint, Mark G. J. van den Brand, chercheur du groupe Asf+Sdf, nous a rejoint en 2001 pour travailler dans l'équipe pendant 14 mois. Asf+Sdf se compose de deux parties : Sdf (*Syntax Definition Formalism*) permet de décrire des syntaxes en utilisant des règles de grammaire hors contexte, ce qui donne une grande expressivité. Asf (*Abstract Definition Formalism*) est le formalisme permettant de décrire des règles de réécriture. Il y a de nombreuses similitudes entre Elan et Asf+Sdf, en particulier les notions de termes et de règles de réécriture. Mais il y a aussi de grandes différences. Elan a un formalisme de règles beaucoup plus riche, permettant d'utiliser des symboles associatifs-commutatifs et des stratégies possédant des opérateurs de *backtracking*. À l'inverse, le formalisme de description de syntaxe Sdf est plus riche que celui

---

ternational Workshop on Rewriting Logic and its Applications, WRLA'98 (Pont-à-Mousson, France), volume 15, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science.

[Mor00] Pierre-Etienne Moreau. REM (Reduce Elan Machine) : Core of the new ELAN compiler. In *Proceedings 11th Conference on Rewriting Techniques and Applications*, volume 1833 of *Lecture Notes in Computer Science*, pages 265–269, Norwich, UK, 2000. Springer-Verlag.

d'Elan, à la fois plus expressif (possibilité de définir des familles de lexèmes) et permettant de réduire plus facilement les ambiguïtés. Asf+Sdf ne se réduit pas seulement à deux formalismes, c'est un environnement offrant une édition structurée, ainsi que de nombreux modules de visualisation de graphes et de termes, permettant de trouver facilement des ambiguïtés de *parsing* ou de mieux comprendre les relations d'importation entre modules par exemple.

Lors de la refonte du langage Elan, nous avons entrepris de composer Sdf avec Elan et de réutiliser le méta-environnement pour avoir un environnement de programmation agréable. Outre un prototype correspondant à la version 4 d'Elan, le résultat le plus intéressant fut l'abstraction et la généralisation de Asf+Sdf, devenant Asf+$\langle\cdots\rangle$, permettant de l'adapter à n'importe quel langage de spécification. Ces travaux ont donné lieu aux publications [vdBMR02] et [vdBMV03].

À retenir :

> Le développement de nouveaux environnements ou langages demande de l'énergie et du temps. L'introduction de formats d'échange et la décomposition de nos outils en « briques élémentaires », ré-utilisables indépendamment, permet bien souvent de réduire ces efforts d'implantation.

3.2 Genèse de Tom

En parallèle à ces travaux visant à rendre moins coûteuse en temps de développement et de maintenance la construction d'environnements complets de programmation, j'ai poursuivi une voie de recherche un peu plus personnelle en essayant de répondre à la question : *comment promouvoir et rendre les méthodes formelles plus facilement utilisables ?*

Cette question a pris plus d'importance à mes yeux lorsque j'ai essayé d'appliquer mes travaux de thèse dans un cadre industriel. Je me suis alors rendu compte que savoir résoudre un problème était important, mais qu'il l'était autant, sinon plus, de préserver la façon dont travaillent les utilisateurs potentiels. Lors d'une intervention au cours des journées Paristic 2006, Gérard Berry, directeur scientifique de Esterel Technologies, a insisté sur le fait qu'on ne pouvait pas « changer le workflow d'une entreprise ». Ne

[vdBMR02] Mark. G. J. van den Brand, Pierre-Etienne Moreau, and Christophe Ringeissen. The ELAN environment : a rewriting logic environment based on ASF+SDF technology. In Mark G. J van den Brand and Ralf Lämmel, editors, *Proceedings of the 2st International Workshop on Language Descriptions, Tools and Applications*, volume 65, Grenoble (France), April 2002. Electronic Notes in Theoretical Computer Science.

[vdBMV03] Mark. G. J. van den Brand, Pierre-Etienne Moreau, and Jurgen Vinju. Environments for Term Rewriting Engines for Free ! In R. Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, volume 2706 of *Lecture Notes in Computer Science*, pages 424–435, Valencia (Spain), june 2003. Springer-Verlag.

pas prendre en compte cette contrainte réduirait l'impact de nos efforts de recherche.

Sans avoir pleinement conscience de ce dernier point, en 2000, nous sommes partis du constat suivant : « lorsque nous avons besoin d'un analyseur syntaxique pour un langage donné, il est rare de développer soi-même un analyseur, nous utilisons habituellement des générateurs tels que `Lex` et `Yacc`. Par contre, lorsque nous avons besoin d'un moteur de réécriture pour implanter une procédure de simplification, il est fréquent de ré-implanter complètement des algorithmes de filtrage. » La réutilisation de nos outils est très rare en pratique. Nous avons voulu mettre nos travaux de recherche à disposition d'un plus grand nombre de personnes en concevant un outil qui réduirait les efforts d'implantation de toute procédure de simplification par réécriture. Cet outil se devait d'être multi-langages et de permettre l'écriture de programmes en C ou Java par exemple. La structure de données principale, les termes, ne devait plus être imposée mais devenir un paramètre de l'outil. Enfin, il devait faciliter la réutilisation des programmes et des librairies existantes.

Le langage Tom n'est pas conçu comme un langage à part entière. Rappelant le concept de *Domain-Specific Language* [MHS03], sa conception repose sur l'idée d'« îlot formel » : un programme Tom est composé de constructions algébriques nouvelles, correspondant aux notions de *signature*, de *filtrage*, de *règle*, et de *stratégie*. Ces constructions, imagées par la notion d'îlot, sont de petits espaces isolés dans un ensemble d'une autre nature : le programme écrit dans le langage hôte. Il y a peu de restriction sur la nature du langage hôte et celui-ci peut sans difficulté correspondre aux langages C, C++, C#, Java, Eiffel, Python, Caml, etc. Cette idée correspond à l'approche *piggyback pattern*, décrite dans [Spi01].

Tom ajoute principalement deux nouvelles constructions : « `%match` » et « ‘ » (appelé *backquote*). La première est une extension de `switch/case` permettant de discriminer sur un *terme* plutôt que sur des valeurs atomiques comme des entiers ou des caractères. Les *motifs* sont utilisés pour discriminer et récupérer de l'information dans la structure de données algébrique filtrée. La seconde, inspirée du langage Lisp, permet de construire un terme.

Exemple 1 *Un exemple élémentaire pour présenter* Tom *est la définition de l'addition sur les entiers de Peano, représentés par la constante* `Zero()` *et le* successeur `Suc(Nat)`*. En considérant* Java *comme langage hôte, l'addition peut être définie en* Tom *de la façon suivante :*

```
public class Peano {
  public final static void main(String[] args) {
    Nat two = `Suc(Suc(Zero()));
    System.out.println("2␣+␣2␣=␣" + plus(two,two));
  }
```

```
  Nat plus(Nat t1, Nat t2) {
    %match(t1, t2) {
      x, Zero() -> { return `x; }
      x, Suc(y) -> { return `Suc(plus(x,y)); }
    }
  }
}
```

Dans cet exemple, nous distinguons bien les parties îlots introduites par `%match` *et « ‘ », dont la portée correspond à un terme bien formé. Il est à noter qu'un îlot peut contenir des morceaux écrits en langage hôte, appelés « lacs », comme le sont les actions introduites par* `-> { ... }`*. Ces lacs, pouvant bien sûr contenir des îlots.*

La construction « ‘ » est utilisée pour exprimer la construction de termes de manière algébrique. Elle permet d'utiliser les variables qui sont instanciées par filtrage (comme `x` *et* `y`*), des constructeurs algébriques (*`Suc`*), et des fonctions du langage hôte, telles que* `plus` *dans cet exemple. La définition de* `plus` *est donnée par filtrage et correspond à une fonction Java, qui peut être utilisée de manière classique, partout ailleurs.*

Afin d'être le plus fidèle possible au concept d'îlot formel, le compilateur Tom ne parse pas les parties écrites en langage hôte. Comme l'illustre la figure 3.1, seules les constructions ajoutées sont examinées pour être traduites en des instructions du langage hôte. Cette étape de « dissolution » est faite *in situ* : la traduction se fait en lieu et place des constructions reconnues, sans modifier ni déplacer le code hôte. Cette caractéristique est essentielle pour permettre le déboguage des applications ainsi produites.

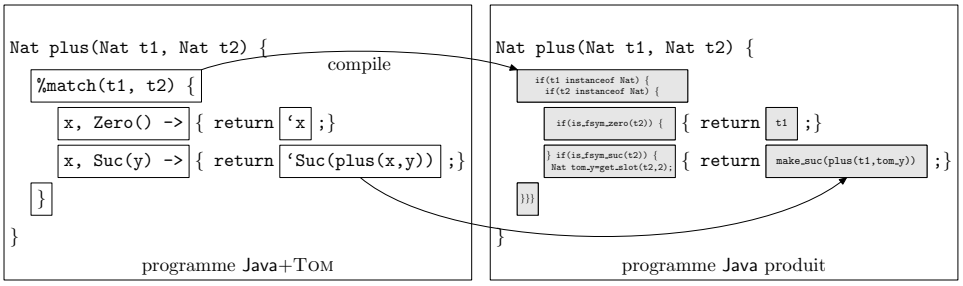


Figure 3.1 – *Compilation d'un programme Java+*Tom *: les constructions* Tom *sont traduites en des instructions Java. Le code Java initial n'est pas modifié lors de la dissolution.*

L'exemple présenté illustre une des caractéristiques principales de Tom : les constructions `%match` et « ‘ » sont intégrées de manière peu intrusive

au sein du langage hôte. Cela permet en particulier de réutiliser les outils de développement existants pour les langages hôtes, tels les modes pour les éditeurs `emacs`, `vim`, ou les environnements intégrés (IDE) que sont `Eclipse` ou `Visual Studio` par exemple.

Changer le workflow d'une entreprise est notoirement impossible. Promouvoir un nouveau langage et faire changer les techniques de développement d'une équipe ou d'un programmeur constitue un défi difficile. Il faut que le gain potentiel soit supérieur à l'investissement et aux risques encourus. La facilité d'utilisation ainsi que la possibilité de réutiliser un langage et son environnement de développement (éditeurs, outils d'analyse, bibliothèques, etc.) sont essentielles, mais pas suffisantes. Accroître la productivité est un argument souvent bien compris. C'est ce que nous faisons en proposant des constructions de haut niveau, diminuant le nombre de lignes à écrire, le risque d'erreurs, et augmentant la lisibilité ainsi que la compréhension des programmes écrits. Un exemple bien connu de la « communauté réécriture », le *filtrage de listes*, correspondant au filtrage modulo l'associativité avec élément neutre, fait partie des constructions à la fois utiles, faciles à utiliser, mais difficiles à programmer.

Exemple 2 *Le filtrage de liste permet d'itérer sur les éléments d'une liste. Si on considère les listes d'entiers naturels comme des objets algébriques de sorte* `List` *ainsi que l'opérateur de liste* `conc`*, on peut définir l'affichage de tous les éléments d'une liste par :*

```
public final static void main(String[] args) {
  List l = `conc(Zero(), Suc(Zero()), Suc(Suc(Zero())));
  affiche(l);
}

public void affiche(List l) {
  %match(l) {
    conc(X1*,e,X2*) -> {
      System.out.println("element␣"+ `e + "\t" +
                         "en␣position␣" + `X1*.size());
    }
  }
}
```

L'action associée au motif est exécutée pour chaque filtre trouvé, en instanciant les variables de liste `X1*` *et* `X2*` *par les sous-listes préfixes et suffixes de la liste* `l`*. L'exécution du programme produit ainsi en sortie :*

```
element Zero()           en position 0
element Suc(Zero())      en position 1
element Suc(Suc(Zero())) en position 2
```

On voit ici que la variable `X1*` *a été successivement instanciée par des sous-listes comportant* 0, 1, *puis* 2 *éléments. Il est important de noter que les variables* `l` *et* `X1*` *sont du type* `List`*, défini dans la bibliothèque standard de* Java*. Il est ainsi possible d'utiliser les fonctions de cette bibliothèque, en particulier la méthode* `size()`*.*

Contrairement au filtrage syntaxique, le filtrage associatif avec élément neutre n'est pas unitaire. Il peut exister plusieurs filtres, solutions d'un problème de filtrage. Dans ce cas, l'action associée au motif sera exécutée pour chaque filtre. Le calcul des solutions sera bien évidemment interrompu si une instruction modifiant le flot de contrôle (`break`, `return`, `throw`, etc.) est exécutée.

Exemple 3 *En s'appuyant sur ces principes élémentaires, la combinaison du filtrage de liste avec le test conditionnel offert par le langage hôte permet de décrire élégamment un algorithme de tri par exemple :*

```
public List bubbleSort(List l) {
  %match(l) {
    conc(X1*,x,y,X2*) -> {
      if(greater(‘x,‘y)) {
        return ‘bubbleSort(conc(X1*,y,x,X2*));
      }
    }
    _ -> { return l; }
  }
}
```

Pour chaque filtre instanciant `X1*`*,* `x`*,* `y`*, et* `X2*`*, la condition* `greater(‘x,‘y)` *est évaluée. Celle-ci retournant* `true` *lorsque* `x` *est plus grand que* `y`*. Dans ce cas, les deux éléments sont permutés et la fonction* `bubbleSort` *est appelée récursivement. Lorsqu'il n'existe plus de filtre tel que* `x` *et* `y` *soient permutables, c'est que la liste* `l` *est triée.*

L'algorithme de tri présenté n'est pas le plus efficace puisqu'il s'exécutera en n^2*, avec* n *la longueur de la liste* `l`*. Mais c'est indéniablement une présentation simple, lisible et correcte, qui peut suffire dans bon nombre d'applications non critiques en temps, où la taille des listes manipulées n'excède pas quelques dizaines d'éléments.*

Contrairement aux constructions de filtrage offertes par les langages fonctionnels tels que Caml, le filtrage de Tom n'est pas unitaire et offre de plus la possibilité d'exprimer des motifs non-linéaires. Ainsi, lorsqu'un même nom de variable apparaît plusieurs fois dans un motif, il est nécessaire que ces variables aient même valeur pour obtenir un filtre. Cela est particulièrement

utile combiné avec le filtrage de liste, pour exprimer la recherche et l'élimination des doublons dans une liste, ou pour vérifier la présence d'un élément donné dans une liste.

Exemple 4 *Cet exemple illustre l'élimination des doublons en ne conservant que la première occurrence.*

```
public List removeDouble(List l) {
  %match(l) {
    conc(X1*,x,X2*,x,X3*) -> {
      return `removeDouble(conc(X1*,x,X2*,X3*));
    }
    _ -> { return l; }
  }
}
```

Dans les exemples 1, 2, 3 et 4, on peut remarquer que les types des objets filtrés (`Nat` et `List`) sont des types du langage hôte. Leur implantation n'est donc pas imposé par Tom. Cette généricité, qui fait de la structure de données un paramètre de l'algorithme de filtrage, est une caractéristique forte du langage. Pour que le code généré puisse s'exécuter correctement, il faut bien entendu qu'un « lien » existe entre les données manipulées et les motifs algébriques utilisés en partie gauche de règle. Ce lien s'appelle un « ancrage formel ». Cette approche est essentielle dans notre démarche : c'est en facilitant l'intégration de nouvelles constructions, en laissant l'utilisateur continuer à utiliser ses structures de données, en évitant des traductions inutiles, que nous pensons promouvoir l'utilisation de méthodes formelles dans un plus grand nombre d'applications, éventuellement industrielles.

Ces idées d'ajout de nouvelles primitives dans un langage quelconque par intégration d'îlots, étaient assez novatrices lorsqu'elles ont été développées en 2000. Elles ont été présentées dans [MRV01] et [MRV03]. Elles apparaissent depuis dans les travaux [LM03, EOW07, RL07].

À retenir :

En instanciant le concept d'îlot formel, Tom ajoute une primitive de filtrage (`%match`) et une primitive de construction (« ` ») dans les langages

[MRV01] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A Pattern-Matching Compiler. In Didier Parigot and Mark G. J. van den Brand, editors, *Proceedings of the 1st International Workshop on Language Descriptions, Tools and Applications*, volume 44, Genova (Italy), April 2001. Electronic Notes in Theoretical Computer Science.

[MRV03] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 61–76, Warsaw, Poland, May 2003. Springer-Verlag.

existants. Contrairement aux règles de réécriture, le membre droit n'est pas un terme mais une action écrite dans le langage hôte. Le filtrage n'étant pas unitaire dans le cas associatif, une même action peut être exécutée plusieurs fois, c.-à-d. pour chaque solution du problème de filtrage. Il n'y a pas de restriction sur la forme des membres gauches. Ils peuvent contenir un nombre arbitraire de symboles associatifs ainsi que des variables non linéaires.

3.3 Fondations

Le travail décrit dans cette section a été réalisé avec Claude KIRCHNER et Émilie BALLAND, dans le cadre de sa thèse.

Afin de pouvoir expliquer et généraliser l'approche suivie dans le cadre de TOM, nous avons utilisé la métaphore insulaire et nous avons entrepris de formaliser cette idée d'îlot formel. On appelle ainsi *océan* le langage hôte qui est étendu par l'ajout de nouvelles constructions, appelées *îles*. Le cycle de vie d'une île peut se décomposer en quatre phases :

- l'*ancrage* correspond à la mise en relation des grammaires et des sémantiques des deux langages (île et océan),
- la *construction* correspond à l'intégration d'une nouvelle construction (île) dans un programme existant (océan),
- la *récolte* correspond aux gains apportés par une telle intégration ; gains qui peuvent être des preuves de correction ou de complétude par exemple,
- la *dissolution* correspond à une étape de traduction de l'île en langage océan.

Étant donnés deux langages (appelés ol et il), pour caractériser leur combinaison (oil) nous avons besoin de caractériser les relations qui existent entre leurs grammaires ($\mathcal{G}_{\mathsf{ol}}$ et $\mathcal{G}_{\mathsf{il}}$), les programmes pouvant être écrits dans ces syntaxes, leurs sémantiques, ainsi que les objets ($\mathcal{O}_{\mathsf{ol}}$ et $\mathcal{O}_{\mathsf{il}}$) manipulés par ces programmes. Nous introduisons pour cela différents concepts et notations. $\mathcal{G} = (A, N, T, R)$ est une *grammaire* où A est l'axiome initial, N et T sont des ensembles finis disjoints correspondant aux non-terminaux et aux terminaux, R est un ensemble fini de règles de la forme $N \to (N \cup T)^*$. On note $\texttt{left}(R)$ l'ensemble des membres gauches de R.

Définition 1 (ancrage syntaxique) *Soit* $\mathcal{G}_{\mathsf{ol}} = (A_{\mathsf{ol}}, N_{\mathsf{ol}}, T_{\mathsf{ol}}, R_{\mathsf{ol}})$ *et* $\mathcal{G}_{\mathsf{il}} = (A_{\mathsf{il}}, N_{\mathsf{il}}, T_{\mathsf{il}}, R_{\mathsf{il}})$ *deux grammaires, on appelle* ancrage syntaxique *une fonction* $\mathtt{anch}(\mathcal{G}_{\mathsf{ol}}, \mathcal{G}_{\mathsf{il}}) \in N_{\mathsf{ol}} \to N_{\mathsf{il}}$ *telle que* $T_{\mathsf{ol}} \cap T_{\mathsf{il}} = \emptyset$ *et* $N_{\mathsf{ol}} \cap N_{\mathsf{il}} = \emptyset$.
La grammaire correspondant à la combinaison de $\mathcal{G}_{\mathsf{ol}}$ *et* $\mathcal{G}_{\mathsf{il}}$ *est donnée par* $\mathcal{G}_{\mathsf{oil}} = (A_{\mathsf{ol}}, N_{\mathsf{ol}} \cup N_{\mathsf{il}}, T_{\mathsf{ol}} \cup T_{\mathsf{il}}, R_{\mathsf{ol}} \cup R_{\mathsf{il}} \cup \mathtt{anch}(\mathcal{G}_{\mathsf{ol}}, \mathcal{G}_{\mathsf{il}}))$.

La notion d'ancrage formel permet de décrire précisément comment deux syntaxes de langages se combinent. Les conditions $T_{\mathsf{ol}} \cap T_{\mathsf{il}} = \emptyset$ et $N_{\mathsf{ol}} \cap N_{\mathsf{il}} =$

$\emptyset$ assurent qu'il n'y a pas de conflit entre les grammaires considérées. Pour être plus général on peut autoriser l'utilisation de constructions du langage océan dans une île, on parle alors de *lac*. Il suffit pour cela d'affaiblir la dernière condition en : $N_{\mathsf{ol}} \cap \mathtt{left}(R_{\mathsf{il}}) = \emptyset$.

Exemple 5 *Dans le cadre de* Tom*, qui introduit principalement deux nouvelles constructions,* `%match` *et « ‘ », les points d'ancrage, dans la grammaire du langage* Java*, correspondent aux non-terminaux de type Instruction et Expression. Nous avons ainsi :*

$$\mathtt{anch}(\mathcal{G}_{\mathsf{Java}}, \mathcal{G}_{\mathrm{TOM}}) = \left\{ \begin{array}{lcl} \langle Instruction \rangle & ::= & \langle MatchConstruct \rangle \\ \langle Expression \rangle & ::= & \langle BackQuoteConstruct \rangle \end{array} \right\}$$

Deux langages donnés ont habituellement des syntaxes et des sémantiques différentes. Il en est de même pour les objets manipulés. Afin de pouvoir définir la sémantique résultant de la combinaison des deux langages, il est important de préciser les relations existantes entre les modèles objets. Nous introduisons pour cela les notions de *représentation* et d'*abstraction*, permettant de traduire les données d'un monde à l'autre. Ces définitions sont à rapprocher des travaux sur le raffinement de modèles de données [dRE98, ASVO05].

Définition 2 (fonction de représentation) *Soit $\mathcal{O}_{\mathsf{ol}}$ et $\mathcal{O}_{\mathsf{il}}$ les ensembles d'objets des langages océan et îlot, on appelle* fonction de représentation*, notée $\lceil\ \rceil$, une fonction injective totale de $\mathcal{O}_{\mathsf{il}}$ vers $\mathcal{O}_{\mathsf{ol}}$. On appelle* fonction d'abstraction*, notée $\lfloor\ \rfloor$, une fonction surjective de $\mathcal{O}_{\mathsf{ol}}$ vers $\mathcal{O}_{\mathsf{il}}$, éventuellement partielle. Ces fonctions sont telles que $\forall x \in \mathcal{O}_{\mathsf{il}}, \lfloor\ \rfloor \circ \lceil\ \rceil(x) = x$*

Ces deux fonctions permettent d'établir une correspondance entre $\mathcal{O}_{\mathsf{ol}}$ et $\mathcal{O}_{\mathsf{il}}$, mais n'imposent pas de contrainte sur la représentation des objets. En particulier, la fonction $\lceil\ \rceil$ ne préserve pas forcément les propriétés structurelles des objets. À titre d'exemple, supposons que $\mathcal{O}_{\mathsf{ol}}$ corresponde au modèle objet de Java et que $\mathcal{O}_{\mathsf{il}}$ corresponde aux termes algébriques, il existe bien des fonctions $\lceil\ \rceil$ permettant de représenter un terme par une structure de données Java. Parmi celles-ci, nous nous intéressons à celles qui préservent la notion d'égalité, et plus généralement un ensemble de prédicats. C'est précisément cette idée qui est modélisée par les définitions suivantes.

Définition 3 (ancrage de prédicats) *Soit $\mathcal{P}_{\mathsf{il}}$ et $\mathcal{P}_{\mathsf{ol}}$ deux ensembles de prédicats, on appelle* ancrage de prédicats *une fonction injective ϕ de $\mathcal{P}_{\mathsf{il}}$ vers $\mathcal{P}_{\mathsf{ol}}$ telle que $\forall p \in \mathcal{P}_{\mathsf{il}}, arity(p) = arity(\phi(p))$. Cette fonction est étendue en un morphisme sur les formules du premier ordre :*

$$\forall p \in \mathcal{P}_{\mathsf{il}}, \forall t_1, \ldots, t_n, \phi(p(t_1, \ldots, t_n)) = \phi(p)(t'_1, \ldots, t'_n)$$
$$\text{avec } t'_i = \left\{ \begin{array}{ll} \lceil t_i \rceil & \text{si } t_i \in \mathcal{O}_{\mathsf{il}} \\ t_i & \text{sinon (c.-à-d.lorsque } t_i \text{ est une variable)} \end{array} \right.$$

Définition 4 (fonction de représentation ϕ-formelle) *Soit ϕ un ancrage de prédicats. Une fonction de représentation $\lceil\ \rceil$ est dite* ϕ-formelle *si $\forall p \in \mathcal{P}_{il}, \forall o_1, \ldots, o_n \in \mathcal{O}_{il}$ avec $n = ar(p)$, on a :*

$$p(o_1, \ldots, o_n) \Leftrightarrow \phi(p)(\lceil o_1 \rceil, \ldots, \lceil o_n \rceil)$$

Exemple 6 *Dans le cadre de* TOM*, vu comme un îlot formel pour* Java*, les données manipulées sont des termes algébriques. Considérons par exemple que $\mathcal{O}_{il}$ corresponde aux entiers de Peano, représentés par les constructeurs* `Zero` *et* `Suc`*. Parmi les différentes structures de données offertes par* Java*, nous considérons que $\mathcal{O}_{ol}$ corresponde aux entiers machine de sorte* `int`*.*

Il existe bien des fonctions $\lfloor\ \rfloor$ et $\lceil\ \rceil$ permettant de représenter les entiers de Peano par des `int` *et réciproquement. Par exemple, $\lceil Zero \rceil = 0$ et $\lceil Suc(n) \rceil = 1 + \lceil n \rceil$. La fonction d'abstraction se définissant par : $\lfloor 0 \rfloor = Zero$ et $\lfloor n \rfloor = Suc(\lfloor n - 1 \rfloor)$ si $n > 0$.*

Considérons maintenant les relations d'égalité $\equiv$ et $==$ comme exemple de prédicats respectivement définis sur les entiers de Peano et les `int`*. En considérant l'ancrage de prédicat $\phi = \{\equiv, ==\}$, la fonction de représentation $\lceil\ \rceil$ est ϕ-formelle parce que deux entiers de Peano sont égaux (avec $\equiv$) si et seulement si leur représentations sont égales (avec $==$).*

On considère maintenant que les deux langages il et ol ont des sémantiques à grand pas bien définies. On se donne aussi une notion d'environnement notée $\mathcal{E}nv_{\mathsf{il}}$ et $\mathcal{E}nv_{\mathsf{ol}}$. Il est alors possible de définir la sémantique du langage oil résultant de l'ancrage d'il dans ol.

Définition 5 (sémantique de oil) *Étant données deux sémantiques bs_{il} et bs_{ol} respectivement définies par des ensembles de règles d'inférence $\mathcal{R}_{il}$ et $\mathcal{R}_{ol}$, on définit la sémantique de oil par :*

- *la relation de réduction $bs_{oil} = bs_{ol}$,*
- *l'ensemble de règles d'inférence $\mathcal{R}_{oil} = \mathcal{R}_{ol} \cup \mathcal{R}'_{il} \cup \{r_1, r_2\}$ avec*
  - *$\mathcal{R}'_{il} = \mathcal{R}_{il}$ où $\langle \epsilon, i \rangle \mapsto_{bs_{il}} \epsilon'$ est remplacé par $\langle \epsilon, \delta, i \rangle \mapsto_{bs_{il}} \langle \epsilon', \delta \rangle$ ($\delta \in \mathcal{E}nv_{ol}$ et $\epsilon, \epsilon' \in \mathcal{E}nv_{il}$),*
  - *les règles d'inférence r_1 et r_2 :*

$$\frac{\langle \lfloor \epsilon \rfloor, \gamma(\epsilon), i \rangle \mapsto_{bs_{il}} \langle \epsilon', \delta \rangle}{\langle \epsilon, i \rangle \mapsto_{bs_{ol}} \lceil \epsilon' \rceil \cup \delta}\ r_1 \qquad \frac{\langle \lceil \epsilon \rceil \cup \delta, i \rangle \mapsto_{bs_{ol}} \epsilon'}{\langle \epsilon, \delta, i \rangle \mapsto_{bs_{il}} \langle \lfloor \epsilon' \rfloor, \gamma(\epsilon') \rangle}\ r_2$$

où $\gamma(x) = x - \lceil \lfloor x \rfloor \rceil$ dénote les éléments de l'environnement x qui ne représentent pas un objet de $\mathcal{O}_{il}$.

Étant données deux grammaires $\mathcal{G}_{\mathsf{il}}$ et $\mathcal{G}_{\mathsf{ol}}$, on appelle fonction de *dissolution* une fonction $\texttt{diss} : \mathsf{AST}(\mathcal{G}_{\mathsf{il}}) \rightarrow \mathsf{AST}(\mathcal{G}_{\mathsf{ol}})$, transformant des programmes reconnus par la grammaire $\mathcal{G}_{\mathsf{il}}$ en des programmes de $\mathcal{G}_{\mathsf{ol}}$. Une telle

fonction *préserve les lacs* lorsque pour tout programme i, les lacs de $\texttt{diss}(i)$ sont également des lacs de i. Étant données des fonctions de représentation et d'abstraction, une fonction de *dissolution* est dite *bien formée* lorsque la sémantique est préservée ($\forall i \in \mathsf{AST}(\mathcal{G}_{\mathsf{il}}), \forall \epsilon \in \mathcal{E}nv_{\mathsf{il}}$, on a : $\langle \epsilon, i\rangle \mapsto_{bs_{\mathsf{oil}}} \epsilon' \Leftrightarrow \langle \lceil \epsilon \rceil, \texttt{diss}(i)\rangle \mapsto_{bs_{\mathsf{ol}}} \epsilon'$), le programme ol résultant de la dissolution est syntaxiquement correct ($\forall i \in \mathsf{AST}(\mathcal{G}_{\mathsf{il}}), \texttt{getSort}(i) \in \texttt{anch}(\texttt{getSort}(\texttt{diss}(i)))$), la fonction de dissolution préserve les lacs.

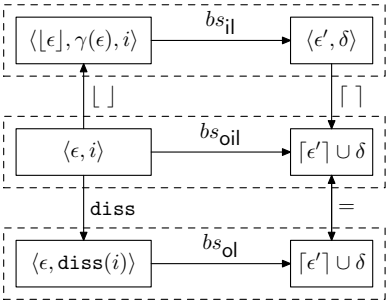


FIGURE 3.2 – *Cette figure illustre les liens existants entre l'évaluation d'une instruction d'*il *et l'exécution de l'instruction* ol *correspondante par dissolution. Partant d'un état* $\langle \epsilon, i\rangle$*, l'évaluation de* i *mène à un environnement* $\lceil \epsilon' \rceil \cup \delta$*, où* δ *représente ce qui a pu être modifié par l'évaluation d'un lac. Par dissolution,* i *est transformé en* $\texttt{diss}(i)$*, puis évalué dans le langage océan, menant à l'état* $\lceil \epsilon' \rceil \cup \delta$*. Dans le langage* il*,* i *peut être évalué directement, menant à l'état* $\langle \epsilon', \delta\rangle$*. Notons que* $\gamma(\epsilon)$ *correspond aux objets de* ϵ *qui ne peuvent pas être représentés dans* il*, mais qui peuvent servir à évaluer un lac.*

Définition 6 (îlot formel) *Soit* ol *et* il *deux langages,* il *est un* îlot formel *construit sur* ol *si il existe :*

- *un ancrage syntaxique* $\texttt{anch}$ *(définition 1 page 19),*
- *des fonctions de représentation et d'abstraction* $\lceil\ \rceil, \lfloor\ \rfloor$ *(définition 2 page 20),*
- *un ancrage de prédicats* ϕ *(définition 3 page 20) tel que la fonction de représentation* $\lceil\ \rceil$ *soit* ϕ*-formelle (définition 4 page précédente),*
- *une fonction de dissolution* $\texttt{diss}$ *bien formée*

Proposition 1 *Soit* il *un îlot formel construit sur* ol*, pre et post deux formules du premier ordre construites sur les prédicats* $\mathcal{P}_{\mathsf{il}}$*, une fonction de dissolution telle que* $\forall i \in \texttt{dom}(\texttt{diss})$*, un environnement* $\epsilon \in \mathcal{E}nv_{\mathsf{il}}$*, on a :*

$$\epsilon \models \{pre\}i\{post\} \Leftrightarrow \lceil \epsilon \rceil \models \{\phi(pre)\}\texttt{diss}(i)\{\phi(post)\}$$

La définition précédente et cette proposition nous donnent les conditions et les outils pour assurer que des propriétés établies sur un îlot formel sont bien préservées par dissolution. Les travaux présentés dans cette section ont été publiés dans [BKM06].

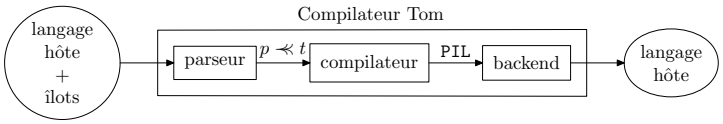
À RETENIR :

> Le concept d'îlot formel est un moyen d'étendre un langage existant en y ajoutant de nouvelles constructions. Les objets manipulés par ces nouvelles constructions sont liés par les notions de *fonction de représentation* et d'*ancrage de prédicats* aux objets existant dans le langage hôte. Lorsque les conditions d'îlot formel sont vérifiées, les propriétés établies sur la construction introduite sont préservées par dissolution vers le langage hôte.

3.4 Instanciation

Le langage TOM, présenté en début de chapitre, est une instance du concept d'îlot formel. Le langage océan peut ainsi être n'importe quel langage de programmation. Les îles sont des constructions algébriques, comme les termes clos, les motifs et les problèmes de filtrage.

L'intégration syntaxique d'un îlot dans le langage océan peut se faire en utilisant un analyseur syntaxique spécifique à chaque langage océan. Dans le cadre de TOM, le langage océan est analysé lexicalement mais pas syntaxiquement. Cette approche permet de réduire les efforts d'implantation, de s'adapter facilement à des évolutions ou des nouvelles syntaxes, et de se synchroniser avec l'analyseur syntaxique propre à TOM. Ce dernier a besoin de compter les accolades ouvrantes et fermantes pour pouvoir détecter la fin d'une construction. De ce fait, pour prendre en compte un nouveau langage il suffit d'étendre l'analyseur pour qu'il reconnaisse les chaînes de caractères et les commentaires.



En pratique, des fonctions de dissolution existent pour les langages C, Java, Python et Caml. Pour ajouter un nouveau langage à cette liste, il suffit d'écrire une nouvelle fonction de dissolution vers le langage considéré. Cette étape est relativement facile dans la mesure où les constructions de TOM sont traduites, de manière complètement indépendante du langage île, vers

[BKM06] Emilie Balland, Claude Kirchner, and Pierre-Etienne Moreau. Formal Islands. In Michael Johnson and Varmo Vene, editors, *11th International Conference on Algebraic Methodology and Software Technology*, volume 4019 of *LNCS*, pages 51–65, Kuressaare, Estonia, July 2006. Springer-Verlag.

$$
\begin{array}{lll}
\texttt{PIL} & ::= & \langle instr \rangle \\
\langle instr \rangle & ::= & \texttt{let}(\textbf{variable}, \langle term \rangle, \langle instr \rangle) \\
& | & \texttt{if}(\langle bexpr \rangle, \langle instr \rangle, \langle instr \rangle) \\
& | & \texttt{accept} \\
& | & \texttt{refuse} \\
\langle variable \rangle & ::= & \ulcorner x \urcorner \ (x \in \mathcal{X}) \\
\langle term \rangle & ::= & \ulcorner t \urcorner \ (t \in \mathcal{T}(\mathcal{F})) \\
& | & \langle variable \rangle \\
& | & \texttt{subterm}_f(\langle term \rangle, \ulcorner n \urcorner) \ (f \in \mathcal{F}, n \in \mathbb{N}) \\
\langle bexpr \rangle & ::= & \ulcorner b \urcorner \ (b \in \mathbb{B}) \\
& | & \texttt{eq}(\langle term \rangle, \langle term \rangle) \\
& | & \texttt{is\_fsym}(\langle term \rangle, \ulcorner f \urcorner) \ (f \in \mathcal{F})
\end{array}
$$

FIGURE 3.3 – *Syntaxe du langage intermédiaire* PIL.

un langage intermédiaire appelé PIL, correspondant à une abstraction de C, Java ou Caml. Ce dernier est alors traduit vers le langage océan considéré.

La syntaxe du langage PIL est décrite figure 3.3. Celle-ci dépend de $\mathcal{F}$, $\mathcal{X}$, $\mathcal{T}(\mathcal{F})$, $\mathbb{B}$, et $\mathbb{N}$. L'ensemble des termes de $\langle term \rangle$ est construit à partir des représentants de $\mathcal{T}(\mathcal{F})$, des variables, et de la construction $\texttt{subterm}_f$ qui permet de récupérer le i^{e} fils d'un terme donné. L'ensemble des expressions $\langle bexpr \rangle$ contient les représentants des booléens, ainsi que deux prédicats : `eq` qui permet de comparer deux termes, et `is_fsym` qui permet d'observer si un terme donné a effectivement un certain symbole de tête. L'ensemble des instructions $\langle instr \rangle$ contient seulement quatre constructions : `let` représente l'assignation d'une valeur à une variable, `if` correspond à la construction « *if-then-else* », `accept` et `refuse` sont deux instructions qui sont des approximations des actions des cas de filtrage. `accept` et `refuse` indiquent si le motif filtre le sujet ou non. Ces dernières peuvent être paramétrées par du code hôte, formant ainsi des lacs.

La sémantique de PIL repose sur l'existence d'un ancrage formel $\lceil\ \rceil$ tel que les propriétés structurelles de $\mathcal{T}(\mathcal{F})$ soient préservées par la sémantique de `eq`, `is_fsym` et $\texttt{subterm}_f$. $\forall t, t_1, t_2 \in \mathcal{T}(\mathcal{F}), \forall f \in \mathcal{F}, \forall i \in [1..ar(f)]$ on a :

$$
\begin{array}{rcl}
\texttt{eq}(\lceil t_1 \rceil, \lceil t_2 \rceil) & \equiv & \lceil t_1 = t_2 \rceil \\
\texttt{is\_fsym}(\lceil t \rceil, \lceil f \rceil) & \equiv & \lceil \mathcal{S}ymb(t) = f \rceil \\
\texttt{subterm}_f(\lceil t \rceil, \lceil i \rceil) & \equiv & \lceil t_{|i} \rceil \text{ si } \mathcal{S}ymb(t) = f
\end{array}
$$

C'est précisément en définissant les opérations `eq`, `is_fsym` et $\texttt{subterm}_f$ que l'implantation de la structure de terme manipulée devient un paramètre du compilateur. En pratique, l'environnement de programmation offre des

outils et des ancrages prédéfinis permettant de cacher au programmeur la complexité de cette approche.

Exemple 7 *Afin d'illustrer ces idées, considérons à nouveau les entiers de Peano et les entiers machine de sorte* `int`*. Dans* Tom*, il est possible de définir une sorte algébrique* `Nat`*, ainsi que deux constructeurs* `Zero` *et* `Suc`*. Les termes de sorte* `Nat` *pourraient être implantés par une structure arborescente, comme c'est le cas en* Caml *par exemple. C'est également possible ici, mais à titre d'exemple, supposons que nous voulions utiliser la sorte* `int` *pour représenter ces entiers de Peano. Il suffit pour cela de « dire à* Tom *» que la sorte* `Nat` *est représentée par* `int`*. Il faut également « expliquer à* Tom *» qu'un entier i de type* `int` *peut être considéré comme la constante* `Zero` *lorsque $i = 0$, ou comme un entier de Peano de la forme* `Suc(n)` *lorsque $i > 0$. Cela revient à considérer les définitions suivantes :*

$$\begin{aligned} \texttt{eq}(i_1, i_2) &\triangleq i_1 = i_2 \\ \texttt{is\_fsym}(i, Zero) &\triangleq i = 0 \\ \texttt{is\_fsym}(i, Suc) &\triangleq i > 0 \\ \texttt{subterm}_{Suc}(i, 1) &\triangleq i - 1 \end{aligned}$$

En Tom*, il est également possible de spécifier comment construire la représentation d'un terme algébrique. Ces « explications » s'expriment de la façon suivante :*

```
%typeterm Nat {
  implement { int }
  equals(t1,t2) { t1 == t2 }
}

%op Nat Zero() {
  is_fsym(i) { i==0 }
  make() { 0 }
}

%op Nat Suc(p:Nat) {
  is_fsym(i) { i>0 }
  get_slot(p,i) { i-1 }
  make(i) { i+1 }
}
```

Exemple 8 *Un autre exemple, inspiré des* views *de Philip Wadler [Wad87], est l'utilisation d'une* vue *des points sur un plan en coordonnées polaires, alors que les objets concrets représentant les points en mémoire sont des couples de coordonnées x et y cartésiennes.*

Les points peuvent être représentés comme un record `(x :int, y :int)` *de coordonnées cartésiennes, mais vus comme comme ayant des coordonnées polaires. On peut alors définir les accès aux sous-termes* $\texttt{subterm}_{point}(t, \rho)$ *et* $\texttt{subterm}_{point}(t, \theta)$ *représentant la norme et l'angle polaire du point.*

$$
\begin{aligned}
\texttt{eq}(t_1, t_2) &\triangleq t_1.x = t_2.x \wedge t_1.y = t_2.y \\
\texttt{is\_fsym}(t, point) &\triangleq \texttt{true} \\
\texttt{subterm}_{point}(t, \rho) &\triangleq \texttt{sqrt(t.x}^2 + \texttt{t.y}^2\texttt{)} \\
\texttt{subterm}_{point}(t, \theta) &\triangleq \texttt{arctan(t.y/t.x)}
\end{aligned}
$$

Cet exemple montre que la structure même de l'objet algébrique qui est manipulé peut être très différente de celle des représentants concrets de ces objets. Cependant, le fait que cette fonction de représentation soit un ancrage formel nous assure qu'il existe un lien fort entre la représentation concrète et la représentation abstraite d'un objet.

À RETENIR :

TOM est une instance du concept d'îlot formel. Son indépendance vis-à-vis de la structure de terme manipulée repose sur l'existence d'un ancrage formel. À l'image des *views* de Philip Wadler, TOM permet de « voir » n'importe quelle structure de données comme un terme. Les constructions de filtrage peuvent alors être utilisées pour rechercher de l'information dans une donnée construite indépendamment de TOM.

3.5 Point de vue

Les solutions paraissent souvent bien simples lorsque le problème a été longtemps étudié. Il est même parfois difficile de percevoir la contribution, tant des variantes proches ont pu être proposées. Les travaux présentés dans ce chapitre font partie de ces résultats qui paraissent simples à établir. Ils correspondent cependant à des étapes importantes de mon activité de recherche.

Le concept d'îlot formel est l'idée qui m'a permis de répondre aux questions posées en début de chapitre. Les principales difficultés ont consisté à identifier un ensemble de constructions et d'ancrages qui soit à la fois restreint et expressif, ainsi qu'à donner des fondements théoriques permettant de généraliser et d'expliquer le concept.

TOM repose essentiellement sur deux constructions (`%match` et « ‘ ») et deux ancrages (`%op` et `%typeterm`). L'idée forte a consisté à ne pas analyser le langage hôte, ce qui a permis de prendre de la distance et de proposer un outil générique, indépendant du langage hôte et de toute structure de données. En contrepartie, la difficulté a été de trouver des modèles et de proposer des algorithmes de compilations à la fois efficaces, n'introduisant pas de sur-coût inutile, et capables de remonter les erreurs, tout en capturant les différents

aspects liés à la réécriture. Ainsi, par exemple, la construction `%match` permet d'intégrer naturellement la notion de filtrage, potentiellement non-unitaire, dans différents langages. Le modèle adopté permet d'arrêter l'énumération de solutions à l'aide des instructions `return`, `break`, ou des exceptions par exemple. Les structures de données ne sont pas imposées ; le filtrage associatif est aussi efficace sur des structures de listes (`head` et `tail` par exemple) que sur des tableaux. Les analyses faites par le compilateur Tom (inférence de type), ainsi que la façon dont le code est généré (le code hôte n'est pas déplacé) permettent de détecter statiquement la plupart des erreurs, qu'elles soient dans les constructions Tom (les îles) ou dans le langage hôte sous-jacent. Ces considérations sont essentielles pour permettre à nos travaux d'être plus facilement utilisés.

Contexte :

Ce chapitre décrit des recherches effectuées dans le cadre d'échanges soutenus entre l'équipe Prothéo et l'équipe de Paul KLINT. Cette coopération, dont j'étais un des principaux animateurs, a été formalisée en 2001 par la création d'une équipe associée « AirCube » (R^3: *Rewrite Rule Research*) ainsi que par l'accueil des chercheurs Mark G. J. VAN DEN BRAND et Jurgen VINJU au sein du projet Protheo.

Un grand nombre d'outils partagés par ces deux équipes reposent sur une représentation standardisée des termes : les ATerms. Nous avons naturellement cherché à améliorer différents aspects de cette bibliothèque en répondant aux questions suivantes.

Questions :

- Comment améliorer la qualité des implantations de structures de données ?
- Comment rendre plus efficace la gestion de termes partagés ?

Contributions :

La bibliothèque des ATerms permet de représenter des termes avec un partage maximal. Ma première contribution a été d'exploiter cette caractérisque pour proposer un algorithme efficace de *ramasse-miettes* utilisant des générations, sans déplacer les objets en mémoire. Le gain par rapport à un algorithme de *mark and sweep* classique vient de l'utilisation de générations dans un environnement où les racines des objets ne sont pas connues.

Ma deuxième contribution a consisté en la conception et la réalisation d'un générateur de structures de données typées. En introduisant un nouveau patron de conception, appelé `SharedObject`, nous avons pu réutiliser le mécanisme de partage des ATerms comme implantation sous-jacente à une interface typée. Les types permettent de détecter des erreurs lors de la compilation. L'utilisation de `SharedObject` et des ATerms garantit efficacité et compatibilité.

4

Route Indirecte

GÉNÉRATION **n.f.** – lat. *generatio* **1**◇ Production d'un nouvel individu. **2**◇ Ensemble des êtres qui descendent de qqn à chacun des degrés de filiation. **3**◇ Ensemble des individus ayant à peu près le même âge. *La jeune, la nouvelle génération.*

4.1 Gestion mémoire à générations

Le travail décrit dans cette section a été fait en partie avec Olivier ZENDRA, chargé de recherche.

En 1998, savoir produire des implantations efficaces de langages à base de règles était un sujet de recherche partagé par plusieurs équipes. Il y a eu de nombreuses avancées liées à l'amélioration des algorithmes de filtrage, à la définition de méthodes permettant de mieux gérer et de réduire le non-déterminisme, ainsi qu'à l'optimisation du code généré par les compilateurs. Un autre point clé a été la définition et l'implantation de structures de données particulièrement optimisées pour représenter des termes, en particulier dans les projets Elan et Maude. À cette même époque, Mark G. J. VAN DEN BRAND, Paul KLINT et Pieter OLIVIER, trois chercheurs du projet Asf+Sdf, choisirent une voie différente et proposèrent une bibliothèque générique permettant de représenter les termes : les ATerms. Cette bibliothèque [vdBdJKO00], écrite en C, définit une interface claire ainsi qu'une représentation textuelle normalisée. Cela permet de réutiliser facilement la bibliothèque et d'échanger les données (des termes) entre des applications distinctes, éventuellement écrites dans des langages différents. Ces travaux sont évidemment à relier au développement d'XML, lequel a construit son succès sur ces mêmes idées.

Une particularité de la bibliothèque des ATerms est de représenter les données avec un partage maximum (appelé également *hash-consing* [AG93]) et de proposer un *ramasse-miettes* [Wil92, Wil94] intégré. Le partage maximum permet de réduire la mémoire nécessaire pour représenter les données.

Bien que cela ne puisse pas être vrai de manière générale, dans le cadre de l'implantation des langages à base de règles, le partage maximum permet d'accélérer les applications : le coût introduit par la recherche du partage est souvent inférieur aux gains (réduction du nombre d'allocations, test d'égalité en temps constant, moins d'échecs du cache processeur, etc.). L'intégration d'un ramasse-miettes décharge le programmeur, contribue à réduire le nombre d'erreurs et accélère les applications. Ce dernier point, parfois controversé, est expliqué et défendu dans [Jon96]. Je partage cet avis et les expérimentations montrent que c'est une réalité dans notre domaine.

L'algorithme assurant le partage des données est relativement simple. Son implantation efficace l'est un peu moins. L'idée principale consiste à maintenir dans une table de hachage (un tableau et des listes mémorisant les collisions) les termes déjà construits. Lorsqu'un nouveau terme doit être construit, une fonction de hachage est utilisée *avant* d'allouer de la mémoire pour rechercher dans la liste de collisions s'il existe un terme identique. Lorsqu'un tel terme est trouvé (même symbole de tête et mêmes sous-termes), ce dernier est retourné, évitant ainsi d'allouer de la mémoire. Dans le cas inverse, de la mémoire est allouée et le terme est ajouté à la table de hachage.

Dans ce contexte, gérer automatiquement la mémoire revient à détecter les termes « vivants » utilisés par l'application. Les algorithmes classiques de ramasse-miettes consistent à parcourir une partie de la mémoire (pile et tas) pour y rechercher des références vers de la mémoire allouée. Il existe une excellente bibliothèque appelée Bdw [BW88] qui peut être utilisée dans de nombreux cas. Une particularité de l'implantation des ATerms est qu'il ne faut pas considérer les cellules de la liste de collisions comme des références « actives », sinon des termes seulement référencés par cette liste pourraient être considérés comme vivants, ce qui entraînerait une rétention mémoire. On appelle ce type de cellule des *weak references*. Cette particularité rend difficile et moins efficace l'utilisation de la bibliothèque Bdw. Pour cette raison les auteurs originels de la bibliothèque des ATerms ont décidé de développer leur propre implantation de ramasse-miettes. Il existe deux principaux types d'algorithmes appelés *mark-and-sweep* et *copy-collector*. Le premier marque les objets vivants et parcourt la mémoire allouée pour y éliminer les objets non marqués. Son temps d'exécution dépend ainsi de la mémoire totale allouée. Le deuxième parcourt et copie seulement les objets vivants dans un autre demi-espace mémoire et récupère l'intégralité du premier demi-espace. L'avantage de cette approche est que le temps d'exécution dépend seulement de l'espace occupé par les objets vivants, et non de la taille de la mémoire allouée. En contrepartie, l'utilisation de deux demi-espaces entraîne une plus grande consommation de mémoire.

Les langages à base de règles et les langages fonctionnels ont pour particularité d'allouer beaucoup d'objets intermédiaires ayant une durée de vie très courte (souvent inférieure au temps séparant deux étapes de ramasse-miettes). Par contre, lorsqu'un objet a survécu à plusieurs étapes de ramasse-

miettes, il est fréquent que cet objet continue à vivre longtemps. Partant de ce constat, il a été développé des algorithmes dits *à générations*. L'idée consiste à séparer les objets en les caractérisant par la génération à laquelle ils appartiennent. Il n'y a pas de limite sur le nombre de générations pouvant être utilisées même si dans la suite nous ne considérons que deux types de générations.

Un ramasse-miettes à générations ne traite pas tous les objets de façon équitable : les « jeunes », susceptibles de contenir un plus grand nombre d'objets « morts » sont ramassés plus souvent que les « vieux ». Le bon fonctionnement d'un tel algorithme dépend d'heuristiques permettant de quantifier cette alternance entre le ramassage des « jeunes » (appelé ramassage mineur) et le ramassage des deux générations en même temps (appelé ramassage majeur). Un autre paramètre de l'algorithme permet de décider pour chaque objet détecté comme étant vivant s'il doit par la suite être considéré comme « vieux » ou non.

Les algorithmes à générations sont assez faciles à mettre en œuvre dans le cadre d'une approche *copy-collector*. En effet, les générations peuvent être matérialisées par des demi-espaces mémoires. Ramasser une génération particulière revient à considérer le demi-espace lui correspondant. Affecter un objet d'une génération à une autre revient à copier l'objet d'un demi-espace à un autre.

Dans le cadre d'un programme C utilisant les ATerms, un algorithme de type *copy-collector* ne peut pas être utilisé parce qu'il est impossible de connaître de manière sûre tous les objets racines sans aide du compilateur. En effet, l'inspection de la pile ne permet pas de distinguer les pointeurs vers des objets, des entiers qui auraient la même valeur. Lors de la copie d'un objet d'un demi-espace à un autre, le pointeur référençant l'objet est modifié. Une fausse identification entraînerait la modification d'un entier. Nous devons donc utiliser un algorithme dit *conservatif*, ne déplaçant pas les objets. La combinaison d'un algorithme à générations avec une approche de type *mark-and-sweep* conservatif est assez rare [Zor89, DWH$^+$90] et non triviale. Une difficulté est d'éviter d'introduire une référence des « vieux » vers les « jeunes », sans quoi le ramassage mineur (qui ne marque pas les objets référencés par un « vieux ») deviendrait incorrect en considérant comme morts des objets vivants.

Pour éviter toute ambiguïté, nous distinguons la notion de *génération* de la notion *d'âge*. Un objet a un *âge* qui correspond au temps écoulé depuis sa création. En pratique, ce temps est approximé par le nombre de ramassages auquel l'objet a survécu (dans notre implantation nous utilisons deux bits pour mémoriser cette information : tout objet ayant un âge supérieur ou égal à trois est considéré comme *vieux*). D'un autre côté, la notion de *génération* correspond à une division du tas. Il n'y a pas de relation forte entre la notion d'âge et la notion de génération. Dans notre algorithme, nous distinguons deux phases, le *marquage* et le *balayage*, ainsi que deux générations,

la *première* et la *deuxième*. Ce qui nous amène à distinguer quatre principales étapes : le *marquage mineur* qui marque les objets jeunes, le *balayage mineur* qui ne balaye que la première génération, le *marquage majeur* qui marque tous les objets vivants (jeunes et vieux), et le *balayage majeur* qui balaye tout le tas (première et deuxième génération). Pour que l'algorithme soit correct il faut qu'il garantisse les propriétés suivantes :

1. un objet vieux ne contient pas de référence vers un objet jeune,
2. la deuxième génération ne contient que des objets vieux,
3. un ramassage majeur récupère tous les objets (jeunes et vieux) qui ne sont pas accessibles,
4. un ramassage mineur ne marque que les objets jeunes, et n'inspecte que la première génération pour récupérer les objets inaccessibles,
5. un ramassage mineur ne récupère aucun objet vieux,
6. un ramassage mineur récupère tous les objets jeunes qui ne sont pas accessibles.

Le partage maximum offert par la bibliothèque des ATerms nous assure que les objets de sont pas mutables : un terme alloué en mémoire n'est jamais modifié. Ainsi, la propriété 1 est garantie par construction.

Pour distinguer la première génération de la seconde, nous aurions pu utiliser deux listes de cellules, mais c'est une idée un peu naïve parce que cela fragmente la mémoire et rend difficile l'identification des références ; il est parfois nécessaire de savoir si une valeur correspond à une adresse de mémoire allouée, pour les distinguer des entiers par exemple. Pour des raisons d'efficacité nous utilisons des listes de blocs, chaque bloc contenant des cellules de même taille appartenant à la même génération. L'allocation d'une cellule se fait alors en un temps très court (9 instructions assembleur) en récupérant une cellule libre, ou en considérant le bloc comme un tableau et en incrémentant un compteur. Une conséquence de cette approche est de réduire la fragmentation tout en facilitant la maintenance des listes de cellules libres. Une cellule fraîchement allouée est naturellement jeune et considérée comme appartenant à la première génération. Cette politique a deux conséquences :

- la mémoire libre dans un bloc de deuxième génération ne peut pas être utilisée tant que le bloc n'est pas détruit puis ré-alloué, ou bien dé-promu en première génération,
- lorsqu'un bloc est promu, sa mémoire libre doit être enlevée de la liste des cellules libres.

La promotion d'un bloc se fait en déplaçant le bloc d'une liste vers l'autre. Pour garantir la propriété 2, un bloc n'est promu que s'il ne contient plus d'objet jeune. Comme mentionné ci-dessus, ses cellules libres doivent être enlevées de la liste des cellules libres, ce qui est une opération coûteuse. Nous avons choisi une approche radicalement différente consistant à reconstruire

complètement la liste des cellules vides au cours de la phase de balayage. En pratique, cela permet de réduire la fragmentation en concentrant les allocations dans un même bloc, tout en simplifiant la promotion d'un bloc.

La phase de ramassage majeur marque tous les objets vivants et balaye les deux générations, ce qui assure la propriété 3. Notons que la mémoire récupérée lors du balayage majeur ne peut être ré-utilisée que lorsqu'un bloc devient complètement vide.

Au cours du ramassage mineur, tous les objets jeunes sont marqués et la première génération est balayée, ce qui garantit la propriété 4. Il faut noter qu'un bloc de la première génération peut contenir un objet vieux, vivant. Celui-ci ne sera pas marqué au cours d'un marquage mineur. Le balayage mineur doit donc récupérer les objets jeunes non marqués, mais pas les objets vieux se trouvant dans la première génération. Une conséquence négative de cette approche est que de la mémoire peut ne pas être récupérée, c'est le cas des objets vieux non accessibles. En pratique, cela n'est pas gênant dans la mesure où ils seront récupérés au cours du ramassage majeur suivant. Cette caractéristique du marquage mineur assure la propriété 5.

La complétude de l'algorithme (propriété 6) est assuré par la propriété 1 : « un objet vieux ne contient pas de référence vers un objet jeune ». Tout objet jeune vivant est ainsi référencé par un autre objet jeune, à moins qu'il ne s'agisse d'une racine contenue dans la pile. De ce fait, tous les objets jeunes vivants sont bien marqués au cours du ramassage mineur. La propriété 2 nous assure que les objets jeunes ne peuvent appartenir qu'à la première génération. De ce fait, il sont récupérés par le balayage mineur. La propriété 6 est donc garantie.

Pour rendre cet algorithme intéressant en pratique, différentes heuristiques ont dû être développées. Elles sont décrites en détail dans [MZ04]. L'algorithme présenté dans ce chapitre a été implanté et correspond à environ 4 000 lignes de C, relativement techniques et difficiles à écrire. C'est l'implantation qui est actuellement diffusée avec la bibliothèque des ATerms. Elle est utilisée, entre autres, par les environnements Asf+Sdf et Stratego, ainsi que par toutes les applications produites avec ces environnements. Par rapport à l'ancienne implantation (sans génération), l'algorithme et l'implantation présentées offrent une réduction du temps d'exécution variant de 20% à 35% en fonction des applications testées. C'est un gain significatif.

À RETENIR :

> Nous sommes dans un contexte où les objets ne peuvent pas être déplacés. La bibliothèque BDW ne peut pas être utilisée ici parce que des *weak references* sont nécessaires pour implanter les ATerms. Nous avons défini

[MZ04] Pierre-Etienne Moreau and Olivier Zendra. GC^2 : A Generational Conservative Garbage Collector for the ATerm Library. *Journal of Logic and Algebraic Programming*, 59(1–2) :5–34, April 2004.

un algorithme à générations où la promotion se fait « en bloc ». Nous distinguons la notion d'âge de la notion de génération et nous assurons qu'un bloc de la deuxième génération ne contient pas de référence vers un objet « jeune ». Six invariants sont présentés ; ils sont assurés par la construction même de l'algorithme et l'immutabilité des objets. L'approche présentée est implantée et correspond au ramasse-miettes intégré à la bibliothèque actuelle des ATerms. Le gain d'efficacité est compris entre 20% et 35% par rapport à l'implantation précédente.

4.2 Génération de structures typées

Le travail décrit dans cette section a été initié lors d'une visite de Jurgen VINJU, dans le cadre de sa thèse et de l'équipe associée AirCube.

Le compilateur TOM est capable de générer du code C qui peut être facilement utilisé avec les ATerms comme structure de données. Cela permet en particulier d'utiliser TOM comme cible pour implanter des langages tels qu'Elan ou Asf+Sdf. Bien qu'écrit initialement en C, le compilateur TOM a très rapidement été ré-écrit en Java, puis *bootstrappé*. J'ai participé au portage en Java de la bibliothèque de ATerms. Une de mes principales contributions a été la définition d'un *design pattern* combinant les notions *d'usine* et de *prototype* pour ne fabriquer que des objets partagés, généralisant ainsi la notion de *singleton*. Cela a permis de réduire la taille de la bibliothèque, d'améliorer sa compréhension et sa maintenance, sans pour autant réduire son efficacité.

Un *prototype* est un objet qui est alloué une seule fois et qui peut être utilisé plusieurs fois comme modèle de construction d'un autre objet. Il offre une méthode permettant de se dupliquer. Dans le cadre des ATerms il est utilisé pour représenter l'objet que l'on cherche à construire. Cette approche permet de s'abstraire du type d'objet à construire dans la mesure où la construction n'est plus faite dans l'usine, mais déléguée au prototype. Lorsqu'un objet équivalent au prototype est trouvé dans la table de hachage, l'objet est retourné. Sinon, une copie du prototype est retournée, après ajout dans la table. Le code ci-dessous illustre le fonctionnement de la bibliothèque :

```
public interface SharedObject {
  int hashCode();
  SharedObject duplicate();
  boolean equivalent(SharedObject peer);
}
public class SharedObjectFactory {
  ...
  public SharedObject build(SharedObject prototype) {
    Bucket bucket = getHashBucket(prototype.hashCode());
    while (bucket.hasNext()) {
```

```
      SharedObject found = bucket.next();
      if(prototype.equivalent(found)) {
        return found;
      }
    }
    SharedObject fresh = prototype.duplicate();
    bucket.add(fresh);
    return fresh;
  }
}
```

L'interface `SharedObject` contient une méthode `duplicate` effectuant une copie des champs (`clone` ne peut pas être utilisé parce qu'un `SharedObject` doit être retourné), une méthode `equivalent` pour tester l'équivalence entre deux objets, et une méthode `hashCode` pour calculer une valeur de hachage. Une méthode `initialize` doit également être implantée. Elle ne fait pas partie de l'interface car sa signature dépend du type d'objet à construire. Son rôle est cependant essentiel pour l'efficacité : elle évite toute allocation dynamique de mémoire et copie la valeur des champs passés en paramètres pour initialiser le receveur.

La bibliothèque des ATerms, dont l'implantation est efficace et sûre, a pendant longtemps été utilisée pour développer le compilateur Tom. Il y a cependant un inconvénient à utiliser cette structure de données : il n'y a pas de typage statique. De ce fait, on peut facilement construire par erreur des arbres de syntaxe abstraits n'ayant pas de sens. Au cours d'une transformation, un nœud de type `Instruction` peut être remplacé par un nœud de type `Expression` par exemple, sans que le compilateur Tom ou Java puisse le signaler. Cela se traduit souvent par des erreurs à l'exécution, assez difficiles à déceler. Suite aux travaux de Hayco de Jong et de Pieter Olivier [JO04], en collaboration avec Jurgen Vinju, nous avons utilisé les possibilités offertes par l'interface `SharedObject` pour générer des structures de données typées, avec partage maximal. Il y a eu de nombreux travaux liés à la génération de structures de données. En particulier Asdl [WAKS97, Han99], ApiGen pour C [JO04], JJForester [KV01], Pizza [OW97], Java Tree Builder [PTW] et JastAdd [HM01]. L'originalité de notre approche est de générer une structure avec partage maximum, basée sur les ATerms, ce qui permet des échanges et une certaine inter-opérabilité. Comme nous le verrons plus loin, page 51, ce travail est à l'origine de l'outil Gom [Rei06], permettant non seulement de générer des structures de données typées, mais offrant de nombreuses autres possibilités, dont la génération de fonctions de canonisation, ainsi que des stratégies de parcours. Afin d'illustrer la structure du code généré, considérons le type de données suivant :

```
datatype Expressions
```

```
Bool ::= true
       | false
       | eq(lhs:Expr, rhs:Expr)
Expr ::= id(value:str)
       | nat(value:int)
       | add(lhs:Expr, rhs:Expr)
       | mul(lhs:Expr, rhs:Expr)
```

Deux classes abstraites étendant les ATerms sont générées :

```
abstract public class Bool extends ATermAppl { ... }
abstract public class Expr extends ATermAppl { ... }
```

Pour chaque constructeur, une classe particulière implantant l'interface `SharedObject` est générée :

```
package bool;
public class Eq extends Bool implements SharedObject {
  ...
  public SharedObject duplicate() {
    Eq.clone = new Eq();
    clone.initialize(lhs, rhs);
    return clone;
  }
  public boolean equivalent(SharedObject peer) {
    return (peer instanceof Eq) && super.equivalent(peer);
  }
  protected void initialize(Bool lhs, Bool rhs) {
    super.initialize("Bool_Eq", new ATerm[] {lhs, rhs});
  }
}
```

La création des termes de la structure de données se fait par l'intermédiaire d'une *usine* :

```
class ExpressionFactory extends ATermFactory {
  private bool.Eq prototype;
  public ExpressionFactory() {
    prototype = new bool.Eq();
  }
  public bool.Eq makeBool_Eq(Expr lhs, Expr rhs) {
    prototype.initialize(lhs, rhs);
    return (bool.Eq) build(prototype);
  }
}
```

L'ensemble de ce travail a été implanté, diffusé, et présenté dans [vdBMV05]. La contribution essentielle est la définition du *design pattern* `SharedObjects`. Celui-ci permet d'assembler du code spécialisé, comme des fonctions de hachage dépendant de la signature algébrique par exemple.

À RETENIR :

> À partir d'un type de données, APIGEN génère un ensemble de classes Java offrant une interface typée. L'implantation des classes repose sur le patron de conception `SharedObject`, qui est une *usine* fabriquant des singletons. Le code généré est efficace, parce que spécialisé, et compatible avec l'interface ATerms, ce qui facilite la migration des outils existants ainsi que la possibilité de *sérialiser* les données. Ce travail est à l'origine de GOM, un composant essentiel dans l'environnement TOM.

4.3 Point de vue

Les travaux présentés dans ce chapitre font appel à des compétences différentes. Ils sont cependant complémentaires et participent à l'amélioration des implantations de structures de données. La conception et l'implantation d'un nouvel algorithme de ramasse-miettes est une tâche difficile, qui demande une bonne compréhension des mécanismes de gestion de la mémoire. Sans être fondamental pour la communauté travaillant sur ce type d'algorithme, le travail présenté est important. Il a permis d'améliorer significativement l'efficacité de la bibliothèque des ATerms, rendant son utilisation encore plus attractive. De nombreux systèmes et outils utilisent cette bibliothèque (le papier [vdBdJKO00] fait partie des dix articles *software engineering* les plus cités en 2000 [Woh07]).

La conception du générateur de structures typées a été une étape importante dans le développement du système TOM. Au début, nous disposions d'un compilateur pour les constructions de TOM et de la bibliothèque des ATerms (écrite en Java) pour représenter les données. Il nous a fallu près d'un an pour trouver un moyen de générer des implantations typées, permettant d'éliminer statiquement un grand nombre d'erreurs. La difficulté a consisté à trouver un patron de conception générique, permettant de n'écrire qu'une seule fois le code effectuant le partage, tout en offrant des performances comparables à celles de la bibliothèque des ATerms, pourtant écrite à la main. Récemment, l'implantation de `SharedObjects` a été rendue *thread safe* par un étudiant du groupe de Paul KLINT. Cette extension permet en particulier d'exploiter les capacités multi-cœurs des processeurs modernes. Cela montre également que cinq années après sa création, cette bibliothèque demeure toujours un centre d'intérêt pour plusieurs groupes de recherche.

[vdBMV05] Mark. G. J. van den Brand, Pierre-Etienne Moreau, and Jurgen Vinju. Generator of Efficient Strongly Typed Abstract Syntax Trees in Java. *IEE Proceedings - Software Engineering*, 152(2) :70–78, April 2005.

Contexte :

La compilation du filtrage est étudiée depuis longtemps [Car84, Aug85] et des algorithmes très efficaces existent [FM01]. Quelques algorithmes ont été certifiés [Con04], mais il n'existe cependant aucun langage à base de règles dont l'implantation du filtrage soit complètement sûre.

Dans le cadre du filtrage, compilation et optimisation sont deux étapes souvent indifférenciables parce la compilation se fait en analysant un ensemble de motifs, en discriminant sur les symboles de tête, en mettant en facteur les parties communes, en créant des sauts d'un état à l'autre de l'automate, ce qui revient à construire du code optimisé.

Dans le cadre des îlots formels, il n'est plus possible de discriminer sur le symbole de tête, parce que celui-ci n'est plus nécessairement *représenté* par un entier. Cela nous a amené à présenter une nouvelle façon de compiler le filtrage, efficace, et sûre.

Questions :

- Comment s'assurer que le compilateur n'introduit pas d'erreurs ?
- Peut-on avoir confiance dans l'optimiseur ?
- Peut-on assurer certaines propriétés aux termes manipulés ?

Contributions :

Dans ce chapitre, nous présentons une méthode permettant de montrer automatiquement que le code généré par le compilateur TOM est correct vis-à-vis des constructions de filtrage utilisées. Nous considérons deux types d'objets : les motifs du programme de départ et le code généré. Nous utilisons leurs sémantiques respectives pour extraire des conditions devant être satisfaites afin d'assurer qu'il n'y a pas eu d'erreur de traduction. La validité de ces conditions est assurée par le prouveur Zenon, qui génère également un certificat Coq. Cette approche pourrait être utilisée pour assurer la correction d'autres compilateurs, tel que Caml par exemple.

Une deuxième contribution est la présentation d'une nouvelle approche où l'étape d'optimisation est séparée de celle de compilation. Pour chaque motif, un code correspondant à du filtrage *one-to-one* est généré. Ce dernier est ensuite transformé pour re-créer du partage et factoriser des tests. L'optimiseur est décrit sous forme de règles de réécriture, qui s'implantent de manière naturelle en TOM. La correction des règles a été établie manuellement, ce qui est un moyen de se donner confiance dans la qualité de l'optimiseur. Les résultats expérimentaux montrent de bonnes performances, comparables avec les meilleurs compilateurs existants.

Dans une troisième section, nous présentons GOM, une extension du générateur de structures de données décrit précédemment. Une fonctionnalité essentielle est de maintenir les termes en forme canonique en appliquant des invariants lors de la construction. La contribution est d'avoir réussi à intégrer ce mécanisme, rappelant la notion de type privé de Caml, dans l'environnement de programmation Java. Reposant sur les mécanismes de protection de Java, il devient impossible de construire un terme (c.-à-d. un objet) qui ne soit pas en forme normale par rapport à un ensemble d'invariants.

5

Cap Bonne Espérance

CERTIFIER **v. tr.** – lat. *certificare*, de *certus* et *facere* « faire » **1◇** Assurer qu'une chose est vraie. **2◇** DR. Garantir par un acte.

CONFIANCE **n.f.** – du lat. *confidentia* **1◇** Espérance ferme, assurance de celui qui se fie à qqn ou à qqch.

SÛR **adj.** – lat. *securus* « libre de souci », de *se*, particule privative, et *cura* « soin, soucis ». **I◇** (Sens subjectif) (PERSONNES) SÛR DE… **1◇** Qui envisage (les évènements) avec une confiance tranquille, sereine **2◇** Qui sait avec certitude, qui est assuré de ne pas se tromper. **II◇** (Sens objectif) **A◇** **1◇** Où l'on ne risque rien ; où une personne, une chose est à l'abri du danger. **2◇** En qui l'on peut avoir confiance ; qui ne saurait décevoir, tromper. **3◇** Sur quoi l'on peut compter. **4◇** Qui agit, fonctionne avec efficacité et exactitude, sans erreur. **B◇** (abstrait) **1◇** Dont on ne peut douter, dont on est convaincu ; qui est considéré comme vrai ou inéluctable.

5.1 Certification du filtrage

Le travail décrit dans cette section a été réalisé avec Claude KIRCHNER et Antoine REILLES, dans le cadre de sa thèse.

Nous avons introduit la notion d'îlot formel, permettant d'ajouter des constructions de plus haut niveau dans des langages existants. L'objectif est de faciliter l'utilisation de méthodes formelles, rendant ainsi plus sûre l'écriture de programmes complexes. Cette approche n'a de sens que si la traduction de ces îlots formels en code hôte est effectuée sans introduire d'erreur. Cette remarque s'applique également aux compilateurs de code hôte vers du code machine directement exécutable. Ces couches inférieures ont cependant l'avantage d'être intensivement testées, ce qui rend la présence d'erreurs moins probable que dans un outil, même de bonne qualité, développé et utilisé par une communauté plus petite. Pour donner confiance dans la démarche proposée et dans les outils que nous réalisons, nous avons entrepris de « certifier » le bon fonctionnement du compilateur TOM.

S'assurer qu'un compilateur préserve les propriétés de son entrée est un problème qui a été étudié dès l'écriture des premiers compilateurs. De nombreux efforts ont porté sur la preuve de correction de parties et même parfois de compilateurs complets. Ces preuves étant faites soit manuellement [MP67, Mor73, ORW95, Wan95, LJVWF02] soit avec l'aide d'un assistant de preuve [Str02, BDL06, Ler06].

Cette dernière approche, consistant à essayer de prouver par des méthodes formelles que le compilateur lui-même est correct, c'est-à-dire compile correctement un sous-ensemble des programmes d'entrée (par exemple, en ne considérant que des programmes syntaxiquement valides et bien typés), est encore bien souvent hors de portée. Toutefois, des travaux récents ont montré qu'il est possible de certifier un compilateur (ou des parties de celui-ci) pour un sous ensemble du langage C en utilisant l'assistant de preuve Coq [Ler06, BDL06]. Ce dernier permet d'extraire automatiquement une implantation certifiée du compilateur à partir de ces preuves.

Dans le cadre d'un langage comme Tom, qui est amené à évoluer rapidement au cours du temps, cette approche n'est pas envisageable. En effet, elle nécessiterait de revoir toutes les preuves à chaque changement, ce qui introduirait une certaine inertie et augmenterait la résistance au changement.

Nous avons donc choisi une autre stratégie permettant de s'assurer formellement de la fiabilité du compilateur. Il s'agit d'arriver à prouver *automatiquement* la correction du code compilé, indépendamment du compilateur. Cette approche « sceptique » vis-à-vis du code produit par le compilateur permet de s'assurer que pour un programme donné, le code produit par le compilateur est correct. Cela peut permettre de détecter la présence éventuelle de bogues dans le compilateur. Cette idée est proche des travaux précurseurs de [BY92], et plus récemment de la méthode *translation validation* [PSS98, NL98]. Une approche comparable, appelée *credible compilation*, à été présentée dans [RM99], laquelle permet de gérer l'utilisation de pointeurs dans le programme source. Notons que ces techniques sont différentes de la méthode *proof-carrying code* [Nec97, WAS03], dont le but n'est pas de prouver que le programme compilé est correct vis-à-vis de son code source, mais plutôt de montrer que certaines propriétés du programme compilé, comme l'absence d'erreurs de typage à l'exécution (*type safety*), l'absence d'accès mémoire non autorisés, ou le respect de certains invariants de sécurité, sont vérifiées.

Dans ce chapitre, nous présentons une méthode, illustrée figure 5.1 page ci-contre, permettant de montrer automatiquement que le code généré par le compilateur Tom est correct vis-à-vis des constructions de filtrage utilisées dans le programme de départ. Pour traiter ce problème, on considère deux types d'objets : d'une part les constructions algébriques, comme les termes clos ($t \in \mathcal{T}(\mathcal{F})$), les motifs ($p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$), et les problèmes de filtrage ($p \ll t$, avec $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et $t \in \mathcal{T}(\mathcal{F})$) ; d'autre part les programmes, exprimés en

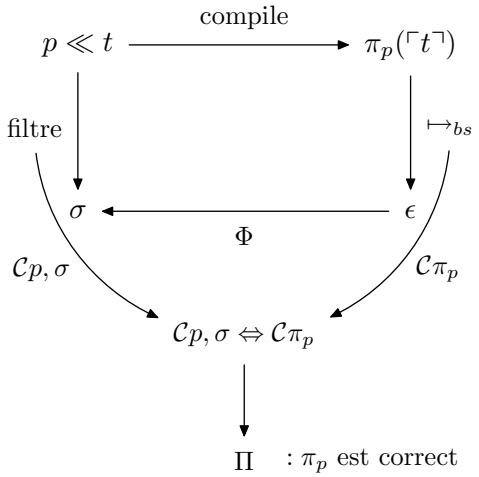


FIGURE 5.1 – *Schéma général de la certification. Étant donné le motif p, on veut montrer que le programme π_p, résultant de sa compilation, résout bien les problèmes de filtrage $p \prec\!\!\prec t$ pour tout terme t. Pour cela, nous extrayons les conditions nécessaires $\mathcal{C}p, \sigma$ pour que σ soit solution, c'est à dire $\sigma(p) = t$. Par ailleurs, en utilisant la sémantique de* PIL *($\mapsto_{bs}$), nous calculons un ensemble de contraintes $\mathcal{C}\pi_p(t)$ tel que l'évaluation de $\pi_p(\lceil t \rceil)$ mène à ϵ. L'équivalence entre ces deux ensembles de contraintes est alors montrée en utilisant le prouveur automatique du premier ordre Zenon [Dol]. Ce dernier est capable de produire un script de preuve Coq, noté* Π*, certifiant que la compilation du motif p est bien correcte.*

PIL, qui sont supposés résoudre des problèmes de filtrage.

Nous considérons à nouveau le langage PIL décrit figure 3.3 page 24. Ce langage est juste assez expressif pour décrire les procédures de filtrage qui nous intéressent. Il est très proche du fragment de langage hôte dans lesquels il doit être traduit à la fin du processus de compilation. Cette phase de traduction n'est constituée que d'un remplacement syntaxique des différentes constructions de PIL par l'instruction équivalente du langage hôte, de telle manière que la preuve de cette partie du processus de compilation ne présente pas de difficulté majeure.

Dans la suite on considère qu'un programme $\pi \in$ PIL est *bien formé* lorsque chaque expression $\texttt{subterm}_f(\texttt{t}, n)$ est telle que $\texttt{t} \in \langle term \rangle$, $n \in [1..ar(f)]$, et $\texttt{is\_fsym}(\texttt{t}, \ulcorner f \urcorner) \equiv \texttt{true}$. En pratique, on vérifie que chaque expression de la forme $\texttt{subterm}_f(\texttt{t}, n)$ appartient à la partie « then » d'une instruction $\texttt{if}(\texttt{is\_fsym}(\texttt{t}, \lceil f \rceil), \ldots)$. Il faut également que toute variable apparaissant dans une sous-expression soit précédemment initialisée par une construction let.

Exemple 9 *Le programme ci-dessous est bien formé dans l'environnement*

$\Gamma = \{s\}$, $\Delta = \emptyset$, *car l'expression* $\texttt{subterm}_f(\ulcorner s \urcorner, \ulcorner 1 \urcorner)$ *est protégée par la construction* $\texttt{if}(\texttt{is\_fsym}(\ulcorner s \urcorner, \ulcorner f \urcorner), \ldots)$ *avec* $1 \in [1..ar(f)]$, *la variable* $\ulcorner x \urcorner$ *est introduite par un* `let` *et* $\ulcorner s \urcorner$ *est dans* Γ.

$$\begin{array}{l}\texttt{if}(\texttt{is\_fsym}(\ulcorner s \urcorner, \ulcorner f \urcorner),\\ \quad \texttt{let}(\ulcorner x \urcorner, \texttt{subterm}_f(\ulcorner s \urcorner, \ulcorner 1 \urcorner), \texttt{accept}),\\ \quad \texttt{refuse})\end{array}$$

Par contre, le programme :

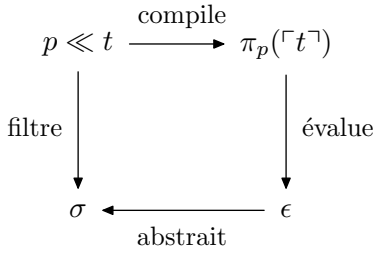
$$\begin{array}{l}\texttt{if}(\texttt{is\_fsym}(\ulcorner s \urcorner, \ulcorner f \urcorner),\\ \quad \texttt{if}(\texttt{eq}(\ulcorner x \urcorner, \texttt{subterm}_g(\ulcorner s \urcorner, \ulcorner 1 \urcorner)),\\ \quad\quad \texttt{accept},\\ \quad\quad \texttt{refuse}),\\ \quad \texttt{refuse})\end{array}$$

n'est pas bien formé dans le même environnement $\Gamma = \{s\}$, $\Delta = \emptyset$ *pour deux raisons :* $\ulcorner x \urcorner$ *n'est pas introduit par une instruction* `let`, *ni dans l'environnement d'exécution, et* $\texttt{subterm}_g$ *n'est pas gardé par une instruction* $\texttt{if}(\texttt{is\_fsym}(\ulcorner s \urcorner, \ulcorner g \urcorner), \ldots)$.

Pour vérifier la validité d'un programme donné, nous avons défini un système de type, présenté figure 5.2 page ci-contre. Un programme π est bien formé dans un environnement d'évaluation si et seulement si on peut construire une dérivation de $\Gamma, \Delta \vdash \pi : wf$ dans ce système. Γ contient les variables initialisées par l'environnement, et Δ contient les termes dont le symbole de tête est connu, ainsi que ce symbole.

Étant donné un problème de filtrage, le fait de savoir si un terme particulier filtre est intéressant. Cependant, ce n'est pas suffisant pour la plupart des applications, et l'on doit aussi calculer un témoin de ce filtrage, c'est à dire une *substitution* qui assigne à chaque variable du motif une valeur.

On introduit pour cela une notion d'*environnement*, modélisant l'état de la mémoire de la machine durant l'évaluation d'un programme. On définit aussi une fonction Φ qui permet de passer d'un environnement concret à une substitution algébrique. Étant donnés une signature $\mathcal{F}$ et un ensemble de variables $\mathcal{X}$, on définit le morphisme Φ des environnements dans les substitutions par $\Phi(\epsilon) = \sigma$ avec : $\sigma = \{x_i \mapsto t_i \mid \epsilon(\lceil x_i \rceil) = \lceil t_i \rceil, x_i \in \mathcal{X} \text{ et } t_i \in \mathcal{T}(\mathcal{F})\}$. Pour prouver la correction du programme π_p correspondant à la compilation d'un certain motif p, on veut s'assurer que pour tout terme t, le diagramme suivant commute :



$$\frac{}{\Gamma, \Delta \vdash \ulcorner b \urcorner : wf \ (b \in \mathbb{B})} \qquad \frac{}{\Gamma, \Delta \vdash \ulcorner x \urcorner : wf} \text{ si } x \in \Gamma$$

$$\frac{}{\Gamma, \Delta \vdash \texttt{accept} : wf} \qquad \frac{}{\Gamma, \Delta \vdash \texttt{refuse} : wf}$$

$$\frac{}{\Gamma, \Delta \vdash \ulcorner t \urcorner : wf \ (t \in \mathcal{T}(\mathcal{F}))} \qquad \frac{\Gamma, \Delta \vdash \texttt{t}_1 : wf \quad \Gamma, \Delta \vdash \texttt{t}_2 : wf}{\Gamma, \Delta \vdash \texttt{eq}(\texttt{t}_1, \texttt{t}_2) : wf}$$

$$\frac{\Gamma, \Delta \vdash \texttt{t} : wf}{\Gamma, \Delta \vdash \texttt{subterm}_f(\texttt{t}, i) : wf} \text{ si } (\texttt{t}, f) \in \Delta \text{ et } i \in [1..ar(f)]$$

$$\frac{\Gamma, \Delta \vdash \texttt{t} : wf \quad \Gamma :: v, \Delta \vdash i : wf}{\Gamma, \Delta \vdash \texttt{let}(v, \texttt{t}, i) : wf}$$

$$\frac{\Gamma, \Delta \vdash \texttt{is\_fsym}(\texttt{t}, \lceil f \rceil) : wf \quad \Gamma, \Delta :: (\texttt{t}, \lceil f \rceil) \vdash i_1 : wf \quad \Gamma, \Delta \vdash i_2 : wf}{\Gamma, \Delta \vdash \texttt{if}(\texttt{is\_fsym}(\texttt{t}, \lceil f \rceil), i_1, i_2) : wf}$$

$$\frac{\Gamma, \Delta \vdash e : wf \quad \Gamma, \Delta \vdash i_1 : wf \quad \Gamma, \Delta \vdash i_2 : wf}{\Gamma, \Delta \vdash \texttt{if}(e, i_1, i_2) : wf} \text{ si } e \neq \texttt{is\_fsym}(\texttt{t}, \lceil f \rceil)$$

FIGURE 5.2 – *Système de type pour vérifier la validité*

On utilise une sémantique à grands pas *à la Kahn* [Kah87] pour exprimer le comportement des programmes `PIL` et leur mécanisme d'évaluation. La relation de réduction de cette sémantique à grands pas est exprimée sur des couples notés $\langle \epsilon, i \rangle$, constitués d'un environnement et d'une instruction du langage `PIL`. La relation de réduction pour la sémantique à grands pas est présentée figure 5.3 page suivante.

Nous avons désormais tous les outils nécessaires pour définir la correction du processus de validation, et montrer comment réduire cette propriété de correction à la validité d'une formule logique du premier ordre. Étant donné un programme π_p dans le langage `PIL` ainsi qu'un motif p, on définira tout d'abord ce que signifie pour π_p d'être une *compilation correcte* de p. Intuitivement on peut dire que cela signifie que l'exécution de π_p sur une entrée $\ulcorner t \urcorner$ représentant un terme t se terminera dans l'état `accept` seulement si le motif p filtre le terme t. Inversement, lorsque p ne filtre pas t, le programme devrait aller dans l'état `refuse`.

Définition 7 (compilation valide) *Étant donné un ancrage formel* $\lceil \rceil$*, un*

$$\frac{}{\langle \epsilon, \texttt{accept} \rangle \mapsto_{bs} \langle \epsilon, \texttt{accept} \rangle} \quad (accept)$$

$$\frac{}{\langle \epsilon, \texttt{refuse} \rangle \mapsto_{bs} \langle \epsilon, \texttt{refuse} \rangle} \quad (refuse)$$

$$\frac{\langle \epsilon, i_1 \rangle \mapsto_{bs} \langle \epsilon', i \rangle}{\langle \epsilon, \texttt{if}(e, i_1, i_2) \rangle \mapsto_{bs} \langle \epsilon', i \rangle} \text{ si } \epsilon(e) \equiv \texttt{true} \quad (iftrue)$$

$$\frac{\langle \epsilon, i_2 \rangle \mapsto_{bs} \langle \epsilon', i \rangle}{\langle \epsilon, \texttt{if}(e, i_1, i_2) \rangle \mapsto_{bs} \langle \epsilon', i \rangle} \text{ si } \epsilon(e) \equiv \texttt{false} \quad (iffalse)$$

$$\frac{\langle \epsilon[x \leftarrow \lceil t \rceil], i_1 \rangle \mapsto_{bs} \langle \epsilon', i \rangle}{\langle \epsilon, \texttt{let}(x, u, i_1) \rangle \mapsto_{bs} \langle \epsilon', i \rangle} \text{ si } \epsilon(u) \equiv \lceil t \rceil \quad (let)$$

FIGURE 5.3 – *Sémantique à grands pas pour* PIL

programme bien formé π_p est une compilation valide *de p lorsque l'on a :*

$$\begin{aligned} &\forall \epsilon, \epsilon' \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}), \\ &\quad \langle \epsilon, \pi_p(\lceil t \rceil) \rangle \mapsto_{bs} \langle \epsilon', \texttt{accept} \rangle \Rightarrow \Phi(\epsilon')(p) = t \quad (sound_{OK}) \\ &\quad \langle \epsilon, \pi_p(\lceil t \rceil) \rangle \mapsto_{bs} \langle \epsilon', \texttt{refuse} \rangle \Rightarrow p \nprec t \quad (sound_{KO}) \end{aligned}$$

Définition 8 (compilation complète) *Étant donné un ancrage formel $\lceil \rceil$, un programme bien formé π_p est une compilation* complète *de p lorsque l'on a :*

$$\begin{aligned} &\forall \epsilon \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}), \\ &\quad p \ll t \Rightarrow \exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_p(\lceil t \rceil) \rangle \mapsto_{bs} \langle \epsilon', \texttt{accept} \rangle \wedge \Phi(\epsilon')(p) = t \quad (complete_{OK}) \\ &\quad p \prec\!\!\prec t \Rightarrow \exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_p(\lceil t \rceil) \rangle \mapsto_{bs} \langle \epsilon', \texttt{refuse} \rangle \quad (complete_{KO}) \end{aligned}$$

Définition 9 (compilation correcte) *Une compilation d'un motif p en un programme π_p est dite* correcte *lorsqu'elle est à la fois* valide *et* complète.

Théorème 1 *Étant donné un ancrage formel $\lceil \rceil$, un motif $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et un programme bien formé π_p, on a :*

$$\begin{gathered} \pi_p \textit{ est une compilation correcte de } p \\ \Longleftrightarrow \\ \forall \epsilon, \epsilon' \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}), \\ \langle \epsilon, \pi_p(\lceil t \rceil) \rangle \mapsto_{bs} \langle \epsilon', \texttt{accept} \rangle \Leftrightarrow \Phi(\epsilon')(p) = t \end{gathered}$$

Propriété 1 *Pour tout environnement $\epsilon \in \mathcal{E}nv$, la dérivation d'une instruction $i \in \langle instr \rangle$ bien formée dans l'environnement Γ, Δ conduit à l'un des états* `accept` *ou* `refuse`*, et la réduction est unique.*

Propriété 2 *Étant donnés un ancrage formel $\lceil \rceil$ et un programme π_p bien formé, on a (reliant les définitions 7 et 8 page ci-contre) : $\forall \epsilon \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F})$, $(sound_{OK}) \Rightarrow (complete_{KO})$ et $(complete_{OK}) \Rightarrow (sound_{KO})$*

Le théorème 1 page précédente est le résultat qui permet de prouver qu'un programme π_p est une compilation correcte d'un motif p. Cependant, l'équivalence entre $\langle \epsilon, \pi_p(\lceil t \rceil) \rangle \mapsto_{bs} \langle \epsilon', \texttt{accept} \rangle$ et $\Phi(\epsilon')(p) = t$ est difficile à prouver car la sémantique à grands pas doit être modélisée et utilisée dans la preuve.

Pour résoudre ce problème, on utilise une forme très simple d'interprétation abstraite (parce que l'évaluation symbolique pourra être faite ici sans approximation) pour dériver statiquement un ensemble de contraintes caractérisant le comportement du programme π_p, dans l'esprit de [CC77, Riv04, GN04]. Pour cela, étant donné un programme π_p, on calcule un ensemble de contraintes $\mathcal{C}\pi_p$ tel que pour prouver que le programme π_p est une compilation correcte du motif p, il est suffisant de montrer que pour tout t, « t satisfait $\mathcal{C}\pi_p$ » si et seulement si « il existe un environnement ϵ tel que $\Phi(\epsilon)(p) = t$ ».

Théorème 2 *Étant donnés un ancrage formel $\lceil \rceil$, un motif $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et un programme bien formé π_p, on a :*

$$\pi_p \textit{ est une compilation correcte de } p$$
$$\Longleftrightarrow$$
$$\forall t \in \mathcal{T}(\mathcal{F}), \mathcal{C}\pi_p(t) \Leftrightarrow \exists \epsilon' \in \mathcal{E}nv, \Phi(\epsilon')(p) = t$$

Ce théorème peut être utilisé pour prouver la correction de la compilation d'un motif. Comme illustré par la figure 5.1 page 41, étant donné un motif p, une condition sur un terme t notée $\mathcal{C}p, \sigma$ peut être extraite. En général, cette condition est de la forme $\exists \sigma, \sigma(p) = t$ mais en utilisant un algorithme de filtrage, la solution σ peut être instanciée par $\{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$ où $t_1, \ldots, t_k$ correspondent aux sous-termes d'un sujet t. Lorsqu'elle est satisfaite, cette condition assure que p filtre t.

De manière similaire, étant donné un programme π_p, la contrainte $\mathcal{C}\pi_p$ peut être calculée. Par application du théorème 2 on sait que l'on peut prouver l'équivalence entre ces deux conditions lorsque le programme π_p est une compilation *correcte* de p. C'est cette preuve qui peut être traitée par un prouveur de théorème automatique, pour fournir une preuve formelle Π.

Les travaux de cette section ont été présentés dans [KMR05b]. Ils sont également implantés et intégrés au compilateur Tom, ce qui les rend utili-

sables en pratique pour s'assurer que l'utilisation de la construction `%match` n'introduit pas d'erreur.

À RETENIR :

Nous avons introduit les notions de compilation valide, complète, et correcte. En donnant une sémantique à grands pas $\mapsto_{bs}$ au langage intermédiaire `PIL`, nous avons pu établir un premier théorème permettant de caractériser une compilation correcte. Ce résultat a été utilisé pour établir un second théorème, directement utilisable : le programme π_p est une compilation correcte du motif p si on arrive à montrer que les contraintes $\mathcal{C}\pi_p$, extraites en utilisant la sémantique $\mapsto_{bs}$ introduite figure 5.3 page 44, sont équivalentes à l'existence d'un environnement ϵ tel que sa substitution associée $\sigma = \Phi(\epsilon)$ vérifie $\sigma(p) = t$, et ceci pour tout terme t. En pratique, les contraintes sont extraites par le compilateur TOM et l'équivalence est prouvée par l'outil Zenon [Dol].

5.2 Sûreté de l'optimiseur

Le travail décrit dans cette section a été réalisé avec Émilie BALLAND, dans le cadre de son stage de master.

Il n'est pas facile d'avoir un compilateur entièrement certifié ou capable de fournir des certificats. Il est néanmoins important de disposer d'un compilateur dans lequel on peut avoir *confiance*. Cette assurance dans les transformations faites par le compilateur peut s'obtenir en testant intensivement le comportement et la qualité du code produit. Nous pouvons également agir en amont en utilisant des méthodes formelles pour concevoir et implanter le compilateur. L'utilisation de règles de réécriture permet de décrire facilement des algorithmes de traduction, directement exécutables. Un autre avantage est de pouvoir raisonner sur les propriétés des transformations ainsi décrites.

Dans la section précédente nous avons présenté une approche permettant de certifier formellement la compilation d'un ensemble de motifs, et ceci indépendamment de l'algorithme de compilation utilisé. Cela signifie que des optimisations peuvent être faites par l'algorithme de compilation. Il existe de nombreux algorithmes de compilation du filtrage [Car84, Aug85, Grä91, FM01]. La plupart d'entre eux effectuent les optimisations au moment de la compilation ; l'idée principale consistant à regrouper les motifs en fonction de leur symbole de tête, à construire un automate dont les transitions dépendent des symboles se trouvant dans le sujet, et à utiliser une instruction du type `switch/case` pour implanter efficacement la fonction de transition de l'automate ainsi produit. Il existe principalement deux approches pour

[KMR05b] Claude Kirchner, Pierre-Etienne Moreau, and Antoine Reilles. Formal Validation of Pattern Matching Code. In Pedro Barahone and Amy Felty, editors, *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 187–197. ACM, July 2005.

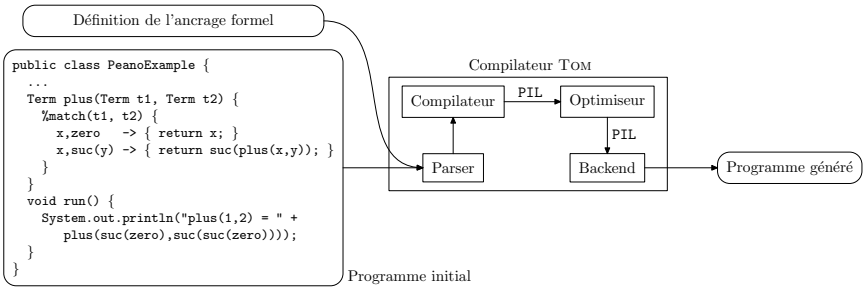


FIGURE 5.4 – *Architecture générale de* TOM *: le compilateur génère un programme* `PIL` *intermédiaire correspondant aux motifs. Dans un second temps, le programme* `PIL` *est optimisé avant d'être traduit dans le code cible désiré.*

construire un automate de filtrage : l'une, optimale en temps mais pas en espace, repose sur les arbres de décision [Car84, Grä91], l'autre, optimale en espace mais pas en temps d'exécution, repose sur les automates avec *backtracking* [Aug85]. Ces deux approches illustrent le compromis *inévitable* entre temps et espace. Pour un ensemble de motifs syntaxiques, linéaires, il n'existe pas d'automate de filtrage qui soit à la fois linéaire en temps et en taille [SRR95].

Dans le cadre de TOM, nous avons choisi une approche différente et peu utilisée. Comme illustré figure 5.4, cela consiste à séparer clairement la phase de compilation du filtrage de la phase d'optimisation du code généré ; l'avantage principal étant de rendre les algorithmes plus simples, et donc plus faciles à comprendre et à maintenir. En contrepartie, la post-optimisation est moins facile à réaliser. Cette approche présente un deuxième avantage qui est déterminant dans notre situation : l'optimisation ne dépend pas de la présence d'une instruction de type `switch/case`. L'utilisateur n'est donc pas forcé de représenter les symboles de fonction par des entiers machines ou des caractères. Le concept d'îlot formel n'est pas affaibli et continue à reposer seulement sur la présence d'un prédicat `is_fsym` pour savoir si un terme commence un symbole donné. Cette remarque est importante lorsqu'on combine filtrage et langage à objets : en effet, un objet peut être vu comme un terme dont le symbole de tête est la classe de l'objet. En utilisant le prédicat `instanceof` pour définir `is_fsym` il n'est pas nécessaire d'associer un entier à chaque classe, par l'intermédiaire d'une table de hachage par exemple, pour être capable de filtrer des objets en mémoire.

Nous considérons à nouveau le langage intermédiaire `PIL` présenté figure 3.3 page 24, auquel nous ajoutons une instruction `hostcode`(`variable`$*$) paramétrée par une liste de variables. Cette instruction correspond à une abstraction du code hôte qui doit être exécuté lorsqu'un motif filtre. La liste de variables correspond aux variables instanciées par TOM qui sont utilisées

code TOM :

```
... code Java ...
...
%match(Term t) {
  f(a) ⇒ { print(...); }
  g(x)⇒ { print(... x...); }
  f(b) ⇒ { print(...); }
}
...
... code Java ...
```

code PIL généré :

```
hostcode(...);
if(is_fsym(t, f), let(t1, subterm_f(t, 1),
  if(is_fsym(t1, f), hostcode(), nop)),
  nop) ;
if(is_fsym(t, g), let(t1, subterm_g(t, 1),
  let(x, t1, hostcode(x)))
  nop) ;
if(is_fsym(t, f), let(t1, subterm_f(t, 1),
  if(is_fsym(t1, b), hostcode(), nop)),
  nop) ;
hostcode(...);
```

FIGURE 5.5 – *La colonne de gauche montre un programme* TOM *qui contient trois motifs :* $f(a), g(x), f(b)$*, où* x *est une variable. Considérons le deuxième motif. Lorsque celui-ci filtre le sujet* t*, cela signifie que* t *commence par le symbole* g *et que la variable* x *est instanciée par son sous-terme immédiat. La colonne de droite montre le code* PIL *qui est généré par* TOM*. On peut remarquer que ce code n'est pas optimal : le prédicat* `is_fsym`(t, f) *est évalué deux fois par exemple.*

dans le code hôte. Cette information sert à l'optimiseur pour détecter les variables non utilisées par exemple. Nous considérons également une variante de sa sémantique à grands pas présentée figure 5.3 page 44. La relation de réduction s'applique sur un triplet composé d'un environnement ϵ, d'une liste de morceaux de codes hôtes et leur environnement δ, ainsi que de l'instruction considérée i :

$$\langle \epsilon, \delta, i \rangle \mapsto_{bs} \delta', \text{ avec } \epsilon \in \mathcal{E}nv, \delta, \delta' \in [\mathcal{E}nv, \langle instr \rangle]^*, \text{ et } i \in \langle instr \rangle$$

Comme l'illustre la figure 5.5, les programmes PIL générés par TOM sont essentiellement composés de `if` imbriqués et de `let`. Les optimisations vont consister à détecter les expressions booléennes ayant des valeurs identiques ou opposées à certains point d'exécution du programme, afin de pouvoir permuter ou fusionner des instructions. Pour cela, nous introduisons la notion d'*équivalence* ou d'*incompatibilité* pour les expressions. `eval` est une fonction qui évalue statiquement en `true` ou `false` une expression e dans un environnement ϵ donné. Étant donnée une liste δ, l'évaluation d'un programme $\pi \in$ PIL résulte en une liste δ'. Au cours cette évaluation de π, l'évaluation de l'expression e se fait dans un environnement ϵ'. Nous appelons Φ la fonction qui associe un tel environnement ϵ' à une sous-expression e de π : $\epsilon' = \Phi(\pi, e, \epsilon, \delta)$.

On dira alors que deux expressions e_1 et e_2 sont π-*équivalentes*, noté $e_1 \sim_\pi$

e_2, si pour tout environnement de départ ϵ, δ, avec $\epsilon_1 = \Phi(\pi, e_1, \epsilon, \delta)$, $\epsilon_2 = \Phi(\pi, e_2, \epsilon, \delta)$, on a bien $\texttt{eval}(\epsilon_1, e_1) = \texttt{eval}(\epsilon_2, e_2)$. De façon similaire, on dira que e_1 et e_2 sont π-*incompatibles*, noté $e_1 \perp_\pi e_2$, lorsque $\texttt{eval}(\epsilon_1, e_1) \neq \texttt{eval}(\epsilon_2, e_2)$.

Une optimisation est une transformation qui préserve la sémantique d'un programme et qui réduit la taille du code (en taille ou en temps). On considère des règles d'optimisation de la forme $i_1 \mapsto\!\!\!\!\!\rightarrow i_2$ si c. Leur application réécrit un programme π en un programme π' si il existe une position ω et une substitution σ telle que $\sigma(i_1) = \pi_{|\omega}$, $\pi' = \pi[\sigma(i_2)]_\omega$ et $\sigma(c)$ est vérifiée. Lorsque c est de la forme $e_1 \sim e_2$ (respectivement $e_1 \perp e_2$), on dit que $\sigma(c)$ est vérifiée lorsque $\sigma(e_1) \sim_{\pi_{|\omega}} \sigma(e_2)$ (respectivement $\sigma(e_1) \perp_{\pi_{|\omega}} \sigma(e_2)$).

L'optimiseur de Tom est constitué d'une première série de règles, relativement classiques, qui réduisent le nombre d'affectations dans un programme. Cela correspond à la propagation des constantes, l'élimination et l'*inlining* de variables, ainsi qu'à la fusion de `let` :

ConstProp	$\texttt{let}(v, t, i)$	$\mapsto\!\!\!\!\!\rightarrow$	$i[v/t]$ si $t \in \mathcal{T}(\mathcal{F})$
DeadVarElim	$\texttt{let}(v, t, i)$	$\mapsto\!\!\!\!\!\rightarrow$	i si $\texttt{use}(v, i) = 0$
Inlining	$\texttt{let}(v, t, i)$	$\mapsto\!\!\!\!\!\rightarrow$	$i[v/t]$ si $\texttt{use}(v, i) = 1$
LetFusion	$\texttt{let}(v_1, t_1, i_1); \texttt{let}(v_2, t_2, i_2)$	$\mapsto\!\!\!\!\!\rightarrow$	$\texttt{let}(v_1, t_2, i_1; i_2[v_2/v_1])$
		si	$t_1 \sim t_2$

La notation $i[v/t]$ correspond au remplacement de v par t dans i, $\texttt{use}(v, i)$ dénote la borne supérieure du nombre de fois où la variable v est utilisée dans l'instruction i.

La deuxième série d'optimisations consiste à réduire le nombre de tests qui seront exécutés pendant le filtrage. Pour cela, les tests avec des conditions équivalentes sont fusionnés, les tests avec des conditions incompatibles peuvent être intervertis ou entrelacés. Le lecteur intéressé par des détails est invité à se reporter à [BM05, BM06]. Nous en avons déduit le système de réécriture suivant :

IfFusion	$\texttt{if}(c_1, i_1, i_1'); \texttt{if}(c_2, i_2, i_2')$	$\mapsto\!\!\!\!\!\rightarrow$	$\texttt{if}(c_1, i_1; i_2, i_1'; i_2')$ si $c_1 \sim c_2$
IfInter_1	$\texttt{if}(c_1, i_1, i_1'); \texttt{if}(c_2, i_2, \texttt{nop})$	$\mapsto\!\!\!\!\!\rightarrow$	$\texttt{if}(c_1, i_1, i_1'; \texttt{if}(c_2, i_2, \texttt{nop}))$
		si	$c_1 \perp c_2$
IfInter_2	$\texttt{if}(c_1, i_1, \texttt{nop}); \texttt{if}(c_2, i_2, i_2')$	$\mapsto\!\!\!\!\!\rightarrow$	$\texttt{if}(c_2, i_2, \texttt{if}(c_1, i_1, \texttt{nop}); i_2')$
		si	$c_1 \perp c_2$
IfSwapping	$\texttt{if}(c_1, i_1, \texttt{nop}); \texttt{if}(c_2, i_2, \texttt{nop})$	$\mapsto\!\!\!\!\!\rightarrow$	$\texttt{if}(c_2, i_2, \texttt{nop}); \texttt{if}(c_1, i_1, \texttt{nop})$
		si	$c_1 \perp c_2$

Afin de garantir la préservation de la taille du code `PIL`, nous avons restreint l'application des règles IfInter à des cas où une des branches est réduite à `nop`, cela évite que du code soit dupliqué. Toutes ces règles diminuent le nombre de tests qui seront effectués à l'exécution, à l'exception de IfSwapping

qui ne sert qu'à permuter des `if` dans le but de rendre voisins deux tests pouvant fusionner ou s'entrelacer.

Étant donnés π_1 et π_2 deux programmes `PIL` bien formés, on dit qu'ils sont *sémantiquement équivalents*, noté $\pi_1 \sim \pi_2$, lorsque :

$$\forall \epsilon, \delta, \exists \delta' \text{ tel que } \langle \epsilon, \delta, \pi_1 \rangle \mapsto'_{bs_\delta} \text{ et } \langle \epsilon, \delta, \pi_2 \rangle \mapsto'_{bs_\delta}$$

Une règle de transformation r est dite *correcte* lorsque son application à tout programme π bien formé produit un programme π' sémantiquement équivalent ($\pi \sim \pi'$).

Nous avons utilisé ces deux définitions pour montrer que chacune des règles précédentes est *correcte*. Pour cela, deux conditions doivent être vérifiées :

1. π' est bien formé,
2. $\forall \epsilon, \delta$, les dérivations de π et π' mènent au même état δ'.

Le système de règles présenté à été naturellement implanté en Tom et Java, puis intégré au compilateur lui-même pour optimiser les programmes compilés par le compilateur Tom. Nous avons évalué l'impact d'une telle approche en mesurant, pour quelques programmes représentatifs, le gain apporté par cette phase d'optimisation. Nous avons également comparé la vitesse des programmes produits avec d'autres implantations de référence telles que le sont Elan et Caml.

	`Fib.`	`Erat.`	`Langton`	`Gomoku`	`Nspk`	`Structure`
`Tom Java`	21.3 *s*	174.0 *s*	15.7 *s*	70.0 *s*	1.7 *s*	12.3 *s*
`avec optimiseur`	20.0 *s*	2.8 *s*	1.4 *s*	30.4 *s*	1.2 *s*	11.3 *s*

- `Fibonacci (Fib.)` calcule 500 fois le 18$^{\text{e}}$ nombre de la suite de Fibonacci en utilisant une représentation de Peano. Sur cet exemple, l'optimiseur a un impact faible parce que le temps passé dans le filtrage est négligeable par rapport au temps passé à allouer des successeurs et à gérer la mémoire.
- `Eratosthène (Erat.)` calcule les nombres premiers jusqu'à 1000 en utilisant du filtrage associatif pour implanter le crible d'Eratosthène. Le gain provient essentiellement de la règle `Inlining` qui permet d'éviter l'instanciation de variables tant que les conditions d'application ne sont pas vérifiées.
- `Langton` est un programme qui calcule la 1000$^{\text{e}}$ itération d'un automate cellulaire en utilisant le filtrage pour implanter la fonction de transition. Cet exemple est intéressant parce qu'il contient plus de 100 motifs (clos). Partant d'une compilation naïve, *one-to-one*, l'optimiseur effectue une transformation de programme telle qu'une paire (position, symbole) ne soit jamais testée plus d'une fois. Cette propriété qui caractérise les automates déterministes ne peut malheureusement pas se généraliser à n'importe quel programme.

- `Gomoku` essaie de placer cinq pions alignés sur une grille 15×15 en utilisant du filtrage associatif pour rechercher des séquences. Cet exemple contient plus de 40 motifs.
- `Nspk` modélise et permet de vérifier des propriétes sur le protocole à clé publique de Needham-Schroeder.
- `Structure` est un prouveur pour le calcul des structures [KMR05a].

	`Fibonnacci`	`Eratosthne`	`Langton`
`Tom Java Optimized`	20.0 *s*	2.8 *s*	1.4 *s*
`Tom C Optimized`	0.95 *s*	0.36 *s*	0.84 *s*
`OCaml`	0.44 *s*	0.7 *s*	1.36 *s*
`ELAN`	0.77 *s*	0.8 *s*	1.26 *s*

L'ensemble de ces mesures ont été faites sur un PowerMac 2 GHz. Elles montrent que l'approche suivie permet d'obtenir des performances comparables à celles des meilleures implantations actuelles. Rappelons à cette occasion que Tom n'est pas restreint à un langage unique. Le fait que la structure de données soit un paramètre du système nous empêche d'utiliser des optimisations classiques reposant sur l'utilisation d'exceptions, de `goto` ou de `switch` par exemple. Ces travaux ont été implantés et présentés dans [BM06].

À retenir :

Les transformations de programmes présentées dans cette section constituent des optimisations qui améliorent l'efficacité d'un programme `PIL`, sans en augmenter la taille. Appliquées au code produit par Tom, comme dans [FM01], nous obtenons une implantation du filtrage dont la taille est linéaire par rapport à la taille et au nombre de motifs. Le défi de notre approche était d'implanter efficacement le filtrage sans utiliser l'instruction `switch/case`, ce qui permet d'exploiter pleinement le principe d'ancrage formel. Les règles présentées sont terminantes pour une stratégie donnée et préservent la sémantique des programmes transformés. L'optimiseur est implanté en Tom et améliore sensiblement l'efficacité des programmes produits, les rendant compétitifs par rapport aux meilleures implantations.

5.3 Formes canoniques

Le travail décrit dans cette section a été réalisé avec Antoine Reilles dans le cadre de sa thèse.

[BM06] Emilie Balland and Pierre-Etienne Moreau. Optimizing pattern matching compilation by program transformation. In Jean-Marie Favre, Reiko Heckel, and Tom Mens, editors, *3rd Workshop on Software Evolution through Transformations (SeTra'06)*. Electronic Communications of EASST, 2006. ISSN 1863-2122.

La confiance que l'on peut avoir dans un programme dépend bien évidemment des preuves formelles qui ont pu être faites ainsi que des jeux de tests qui existent. Mais dans certains cas, ils existe des morceaux de codes qui ne sont ni prouvés ni testés. Il faut alors avoir confiance dans le programmeur qui les a écrits. Pour aider ce programmeur à faire moins d'erreurs, nous lui avons fourni un langage de haut niveau, fondé sur la réécriture. Comme nous l'avons vu section 4.2, nous avons développé un générateur de structures de données typées. Cependant, le système de type utilisé ne permet pas d'avoir un contrôle fin sur la « forme » des données manipulées. Il est pourtant courant d'avoir besoin de spécifier une relation d'équivalence particulière entre les objets, et de travailler en faisant abstraction de cette équivalence.

Exemple 10 *Un exemple d'une telle relation d'équivalence, pour laquelle on veut uniquement considérer un certain représentant canonique est la représentation des nombres rationnels. Ces derniers sont composés d'un numérateur et d'un dénominateur, et on veut généralement considérer le représentant canonique ayant le plus petit dénominateur positif, car cela simplifie d'une part l'affichage, et surtout les algorithmes utilisant ces objets. Ainsi, le signe du nombre rationnel est obtenu simplement en étudiant le signe du numérateur.*

Dans le contexte de la réécriture, il est fréquent de considérer les termes modulo une certaine théorie, comme l'associativité, la commutativité, les éléments neutres, l'idempotence, la distributivité de certains symboles par rapport à d'autres ou des équations spécifiques au domaine [JM92]. Afin d'implanter un système de *réécriture normalisée* [Mar96], dans lequel les règles de réécriture ne travaillent que sur des formes normales d'un système de réécriture terminant et confluent donné, il faut disposer d'un système permettant de calculer ces formes normales. Il est ainsi utile de lier l'implantation de ce système normalisant avec l'implantation de la structure de données représentant les termes. Cela permet non seulement d'obtenir une grande efficacité, mais aussi de s'assurer que le processus d'application des règles de réécriture ne pourra en aucun cas créer des termes qui ne sont pas des formes normales pour ce système, facilitant ainsi l'implantation du moteur de réécriture ainsi que sa lisibilité.

Pour faciliter la définition de structures de données et décrire des invariants, nous avons défini le langage Gom [Rei06] dont la syntaxe et les mécanismes sont proches de ceux présentés section 4.2. Considérons les lois de De Morgan comme une théorie équationnelle pour les booléens. Ces lois sont décrites par les équations $\overline{A \vee B} = \overline{A} \wedge \overline{B}$ et $\overline{A \wedge B} = \overline{A} \vee \overline{B}$. On peut orienter ces équations pour obtenir un système de réécriture confluent et terminant, qui permet alors d'implanter un système de normalisation, après l'application duquel seuls les atomes peuvent être arguments d'une négation. On peut aussi ajouter une règle afin de supprimer les doubles négations. On

obtient alors le système :

$$\begin{array}{lcl} \overline{A \vee B} & \rightarrow & \overline{A} \wedge \overline{B} \\ \overline{A \wedge B} & \rightarrow & \overline{A} \vee \overline{B} \\ \overline{\overline{A}} & \rightarrow & A \end{array}$$

Le mécanisme d'*invariant* de GOM permet de définir des actions arbitraires à exécuter avant (ou en lieu et place de) la fonction originale de création d'un opérateur. Ces actions peuvent être décrites par n'importe quelle construction Java ou TOM. Ces travaux sont à rapprocher de la notion de *type privé* de Caml introduite par Pierre WEIS [LDG+04]. Cette approche permet de forcer l'utilisateur à définir et à utiliser des fonctions de construction explicites pour les objets d'un type algébrique tout en préservant la possibilité d'utiliser le filtrage.

Exemple 11 *Description en* GOM *d'une structure de booléens ainsi que des règles de normalisation correspondant aux lois de De Morgan :*

```
module Boolean
abstract syntax
Bool = | True()
       | False()
       | Not(b:Bool)
       | And(lhs:Bool,rhs:Bool)
       | Or(lhs:Bool,rhs:Bool)
module Boolean:rules() {
  not(not(x)) -> x
  not(and(l,r)) -> or(not(l),not(r))
  not(or(l,r)) -> and(not(l),not(r))
}
```

La construction `module Boolean:rules()` *permet de définir un système de réécriture (confluent et terminant) qui sera appliqué à chaque fois qu'une réduction sera possible. Cela correspond exactement à la notion de « règle non nommée » d'*Elan*. Par conséquent, les termes du module* `Boolean` *sont* toujours *en forme normale par rapport à ce système. Il n'existe pas de moyen, même en programmant directement en* Java*, de créer un terme qui ne serait pas en forme normale.*

En pratique, la construction `rules` *est compilée vers une construction de plus bas niveau, le* hook *de construction :*

```
not:make(arg) {
  %match(arg) {
    not(x)   -> { return `x; }
    and(l,r) -> { return `or(not(l),not(r)); }
    or(l,r)  -> { return `and(not(l),not(r)); }
```

```
  }
}
```

La construction `not:make(arg)` *décrit ce qui doit être fait lors de la construction d'un* `not`*. Ici,* TOM *est utilisé pour analyser* `arg` *par filtrage et implanter les règles de simplification. Lorsque l'exécution de cette construction se termine sans retourner de valeur, lors de la construction de* `not(True())` *par exemple, la fonction de création de l'opérateur (celle allouant de la mémoire) est utilisée.*

L'idée clé pour simuler la notion de type privé en Java a consisté à générer le code correspondant aux *invariants* dans des fonctions de construction statiques et à rendre privé (`private`) le constructeur de la classe. Ainsi, pour la classe `not` par exemple, seule la méthode `public static Bool make(Bool arg)` peut être appelée pour créer une négation. Le constructeur de classe `private not() {}` est quant à lui privé, et donc inaccessible en dehors de la classe. C'est ce mécanisme de protection de Java qui force ainsi l'évaluation des fonctions de normalisation lors de la construction.

La possibilité de définir des règles de réécriture et des invariants de construction est un point fort de GOM, utilisé dans de nombreuses applications. Dans [CMR04], nous définissons un outil permettant de simuler un protocole de communication et de rechercher des failles éventuelles. Pour cela, un espace d'états est exploré, chaque état contenant une liste de messages déjà échangés. Pour accélérer les recherches, nous mémorisons les états déjà visités en utilisant un mécanisme de « cache ». Une optimisation importante a consisté à considérer des classes d'équivalences, regroupant des états correspondant aux mêmes messages échangés, mais pas nécessairement dans le même ordre. Pour cela, nous avons considéré des formes canoniques d'états, revenant à trier les listes de messages. Cet invariant est maintenu par la définition d'un *hook* associé à la fonction de création des listes de messages.

Dans [KMR05a] nous décrivons l'implantation d'un prouveur automatique, efficace et sûr, pour le système BV du calcul des structures [Gug02]. Nous avons modifié les règles originales du calcul et promu certaines règles au titre d'invariant de la structure de données, permettant une implémentation plus simple et plus efficace des règles du calcul. Les structures dans le système BV sont générées par :

[CMR04] Horatiu Cirstea, Pierre-Etienne Moreau, and Antoine Reilles. Rule Based Programming in Java for Protocol Verification. In Narcisso Marti-Oliet, editor, *Proceedings of the 5th International Workshop on Rewriting Logic and its Applications, WRLA'2004 (Barcelona, Spain)*, volume 117, pages 209–227, Barcelona (Spain), April 2004. Electronic Notes in Theoretical Computer Science.

[KMR05a] Ozan Kahramanoğulları, Pierre-Etienne Moreau, and Antoine Reilles. Imple-

$$S ::= \circ \mid a \mid \langle \underbrace{S; \ldots; S}_{>0} \rangle \mid [\underbrace{S, \ldots, S}_{>0}] \mid (\underbrace{S, \ldots, S}_{>0}) \mid \overline{S}$$

où a est un atome, $\circ$ est l'*unité*. On appelle $\langle S; \ldots; S \rangle$ une *structure seq*, $[S, \ldots, S]$ une *structure par*, et $(S, \ldots, S)$ une *structure copar*. $\overline{S}$ est la négation de la structure S. Au cours du calcul, nous avons besoin de maintenir les formules en formes canoniques. Nous utilisons pour cela des règles d'aplatissement et nous trions les sous-termes de *par* et *copar* pour pouvoir utiliser du filtrage associatif à la place du filtrage associatif-commutatif :

$$\begin{aligned}
&\texttt{par(concPar(x))} \to \texttt{x}\\
&\texttt{cop(concCop(x))} \to \texttt{x}\\
&\texttt{seq(concSeq(x))} \to \texttt{x}\\
&\texttt{par(concPar}(\texttt{a}_1, \ldots, \texttt{a}_\texttt{i}, \texttt{par(concPar}(\texttt{x}_1, \ldots, \texttt{x}_\texttt{n})), \texttt{b}_1, \ldots, \texttt{b}_\texttt{j}))\\
&\qquad\qquad \to \texttt{par(concPar}(\texttt{a}_1, \ldots, \texttt{a}_\texttt{i}, \texttt{x}_1, \ldots, \texttt{x}_\texttt{n}, \texttt{b}_1, \ldots, \texttt{b}_\texttt{j}))\\
&\texttt{cop(concCop}(\texttt{a}_1, \ldots, \texttt{a}_\texttt{i}, \texttt{cop(concCop}(\texttt{x}_1, \ldots, \texttt{x}_\texttt{n})), \texttt{b}_1, \ldots, \texttt{b}_\texttt{j}))\\
&\qquad\qquad \to \texttt{cop(concCop}(\texttt{a}_1, \ldots, \texttt{a}_\texttt{i}, \texttt{x}_1, \ldots, \texttt{x}_\texttt{n}, \texttt{b}_1, \ldots, \texttt{b}_\texttt{j}))\\
&\texttt{seq(concSeq}(\texttt{a}_1, \ldots, \texttt{a}_\texttt{i}, \texttt{seq(concSeq}(\texttt{x}_1, \ldots, \texttt{x}_\texttt{n})), \texttt{b}_1, \ldots, \texttt{b}_\texttt{j}))\\
&\qquad\qquad \to \texttt{seq(concSeq}(\texttt{a}_1, \ldots, \texttt{a}_\texttt{i}, \texttt{x}_1, \ldots, \texttt{x}_\texttt{n}, \texttt{b}_1, \ldots, \texttt{b}_\texttt{j}))\\
&\texttt{concPar}(\ldots, \texttt{x}_\texttt{i}, \ldots, \texttt{x}_\texttt{j}, \ldots) \to \texttt{concPar}(\ldots, \texttt{x}_\texttt{j}, \ldots, \texttt{x}_\texttt{i}, \ldots) \text{ if } \texttt{x}_\texttt{j} < \texttt{x}_\texttt{i}\\
&\texttt{concCop}(\ldots, \texttt{x}_\texttt{i}, \ldots, \texttt{x}_\texttt{j}, \ldots) \to \texttt{concCop}(\ldots, \texttt{x}_\texttt{j}, \ldots, \texttt{x}_\texttt{i}, \ldots) \text{ if } \texttt{x}_\texttt{j} < \texttt{x}_\texttt{i}
\end{aligned}$$

La difficulté consiste à maintenir ces invariants, alors même qu'une règle de déduction, appliquée à n'importe quelle position de l'arbre de preuve, peut venir briser ces invariants. L'utilisation de *hooks* a été essentielle pour assurer la correction de notre prouveur. Cela a permis de séparer complètement la partie effectuant la recherche de preuve de la partie maintenant les termes en forme canonique.

Lorsque des *invariants* sont définis, c'est à l'utilisateur de s'assurer que le système de normalisation défini est bien confluent et terminant, car ces propriétés ne seront pas garanties par le compilateur Gom. D'autre part, la combinaison d'invariants pour différentes théories équationnelles dans une même signature doit être faite manuellement par l'utilisateur, la combinaison de systèmes de réécriture ne préservant pas nécessairement les propriétés de confluence et de terminaison. Une extension de ce travail serait de fournir un langage de plus haut niveau étendant Gom et permettant d'exprimer de manière abstraite les différents invariants associés aux opérateurs. C'est en partie ce qui est fait dans le cadre du langage Moca [BHW07] : une extension du langage des types inductifs de Caml permettant d'obtenir des fonctions

menting Deep Inference in Tom. In Paola Bruscoli, François Lamarche, and Charles Stewart, editors, *Structures and Deduction – the Quest for the Essence of Proofs (satellite workshop of ICALP 2005), Technical Report ISSN 1430-211X, Technische Universität Dresden*, pages 158–172, Lisbon, Portugal, 2005.

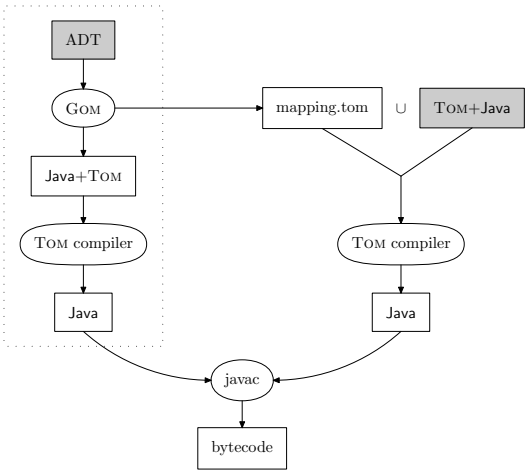


FIGURE 5.6 – *Interactions entre* TOM *et* GOM *: à partir d'une description de structure de données et de ses invariants (décrits par des* invariants *utilisant le langage* TOM*), le compilateur* GOM *produit un ensemble de fichiers* Java *intégrant éventuellement des constructions* TOM *(partie gauche de la figure). Le compilateur* TOM *sera alors appelé pour* dissoudre *ces constructions. Le compilateur génère également l'ancrage formel (*`mapping.tom`*) utilisé par l'application écrite en* TOM *(partie droite de la figure).*

de normalisation pour un type inductif annoté avec des théories qui sont associées aux constructeurs.

On peut considérer GOM comme un composant réutilisable, conçu pour générer des structures de données à la base d'outils plus complexes, de la même manière que la bibliothèque des ATerms et APIGEN ont été utilisés comme base pour Asf+Sdf [dJO04]. Cependant, comme l'illustre la figure 5.6, GOM est parfaitement intégré à l'environnement de développement TOM.

À RETENIR :

Le filtrage permet de rechercher de l'information dans un terme qui a une certaine forme. Dans ce contexte, il est important que deux termes correspondant à la même information soient représentés de façon canonique. Nous avons présenté GOM, un générateur de structures de données typées, qui permet de décrire des invariants à maintenir. Ces invariants sont écrits en TOM+Java et sont tissés dans le code généré. En rendant les constructeurs Java privés, les termes ne peuvent être construits que par une *usine*, qui assure que les invariants sont bien exécutés. Ce travail correspond à l'intégration des types privés de Caml dans un environnement Java. C'est une étape essentielle qui permet d'éviter un grand nombre d'erreurs. C'est également une clé qui permettra d'assurer chapitre 6 que le filtrage effectué par TOM est bien associatif avec élément neutre.

5.4 Point de vue

Les travaux décrits dans ce chapitre ont eu pour but de donner des indicateurs de confiance dans le compilateur utilisé ainsi que dans le code produit. L'utilisation de formes canoniques réduit les risques d'erreurs en simplifiant la tâche du programmeur, qui n'a plus à s'assurer que les termes manipulés sont dans des formes convenables. L'utilisation du filtrage, éventuellement équationnel, introduit un haut niveau d'abstraction qui réduit également le risque d'erreurs de la part du programmeur.

Il reste cependant à avoir confiance dans le compilateur qu'on utilise. Bien que d'apparence simple, la compilation du filtrage n'a rien de trivial. Dans le cas syntaxique, la présence de motifs partageant des constructeurs communs ainsi que des disjonctions permet de générer du code optimisé, n'inspectant qu'un nombre réduit de fois les différentes positions d'un terme. Les algorithmes de compilation effectuant ce type d'optimisation ne sont pas faciles à implanter, à maintenir, et à faire évoluer. Nous avons présenté une méthode permettant de certifier formellement que le code généré a bien la même sémantique que les patterns écrits dans le programme originel. Cette approche pourrait se généraliser et être utilisée pour certifier des programmes compilés par Caml par exemple. Il convient cependant de noter quelques limitations dans l'approche présentée dans la mesure où nous ne l'avons pas généralisée au cas équationnel. La compilation de tels problèmes de filtrage engendre du code récursif, avec des boucles. Il ne suffit plus de montrer la correction du code, il faut également montrer la complétude, assurant que l'ensemble des filtres a bien été calculé, ce qui est un problème difficile. Récemment, nous avons étudié une nouvelle approche consistant à traduire le code `PIL` généré par Tom dans le langage Why ; l'outil Why étant ensuite utilisé pour générer des conditions qui sont vérifiées par le prouveur Ergo. Cette approche prometteuse a permis de vérifier la correction du code généré dans le cas du filtrage syntaxique, en présence d'anti-patterns linéaires. Des travaux sont en cours pour essayer de généraliser ces résultats au cas équationnel.

En attendant que ces travaux aboutissent, nous avons fait en sorte que les algorithmes du compilateur Tom soient les plus faciles à comprendre, à implanter, et à faire évoluer. En séparant les phases de compilation de la phase d'optimisation, et en montrant des propriétés sur l'optimiseur, nous réduisons considérablement la complexité des algorithmes et le risque d'erreur. Récemment, le noyau du compilateur a été ré-écrit en se basant sur un système de règles de propagation et de simplification de contraintes, tel que celui présenté page 64. En utilisant les règles et les stratégies, l'implantation devient ainsi fidèle à la présentation formelle des algorithmes.

Contexte :

Filtrer p vers t c'est trouver une substitution σ telle que $\sigma(p) = t$. Réécrire, c'est chercher une position, filtrer, puis remplacer le terme filtré par un autre terme. C'est un mécanisme simple et élémentaire. Il existe cependant de nombreuses variantes, avec des variables non-linéaires, des variables d'ordre supérieur, des variables d'extension, des disjonctions, des contraintes, des théories équationnelles, et même des négations.

Dans ce chapitre nous nous intéressons au filtrage, non-linéaire, en présence de négations et d'opérateurs associatifs avec élément neutre, et nous essayons de répondre aux questions suivantes.

Questions :

- Peut-on améliorer l'expressivité du filtrage ?
- Qu'est ce qu'un anti-pattern ?
- Peut-on dire que TOM implante du filtrage associatif avec élément neutre ?
- Comment contrôler filtrage et transformation ?

Contributions :

Nous introduisons les anti-patterns, correspondant à la notion de complément, en donnant une sémantique précise ainsi qu'un algorithme, permettant de résoudre les problèmes d'anti-filtrage. Cet algorithme est montré correct, complet, et unitaire dans le cas syntaxique.

Lorsqu'on manipule des listes, il est fréquent d'utiliser une notation variadique, plus commode qu'une notation « peigne droit » à base de *cons* et *nil*. La deuxième contribution est de formaliser l'équivalence entre ces deux notations et de montrer que les formes normales modulo associativité et élément neutre sont mises en relation avec les formes normales des termes variadiques correspondant. Le calcul des formes normales modulo aplatissement et suppression des vides correspond alors au calcul des formes normales pour l'associativité et élément neutre. Cela permet de raisonner sur des algèbres de termes associatives en utilisant comme représentation concrète des objets variadiques, tels que ceux utilisés par TOM.

La troisième contribution, essentielle dans l'environnement TOM, consiste à définir un ensemble minimal d'opérateurs de stratégie, dont la combinaison permet de décrire des parcours et des transformations arbitraires. Le langage proposé autorise des définitions récursives et rend explicite la notion de position dans un terme, permettant ainsi de décrire des stratégies d'exploration non-déterministe. Les stratégies sont réifiées au niveau des termes, permettant de filtrer, de construire, et même de transformer dynamiquement une stratégie. Enfin, une stratégie étant un terme, il est possible d'appliquer une stratégie sur une stratégie. L'ensemble est implanté et intégré à TOM, ce qui rend le langage de stratégie facilement utilisable en présence des toutes les constructions présentées dans ce document.

6

Vers plus d'expressivité

FILTRER **v.** – de *filtre* **1◇** Faire passer à travers un filtre. **2◇** Soumettre à un contrôle, à une vérification, à un tri.

FILTRE **n.m.** – lat. médiév. *filtrum* **1◇** COUR. Appareil (tissu ou réseau, passoire) à travers lequel on fait passer un liquide pour le débarrasser des particules solides qui s'y trouvent. **2◇** SC. TECHN. Corps poreux ou percé de trous, appareil servant à débarrasser un fluide des particules en suspension.

ANT(I)- ◇ Élément, du grec *anti* « en face de, contre », signifiant en composition : **1◇** Qui est situé en face de, à l'opposé de. **2◇** Qui s'oppose à, qui lutte contre les effets de. **3◇** Qui est l'opposé de, le contraire de.

MODULO **prép.** – latin *modulo* ◇ MATH. Suivant la relation d'équivalence (indiquée par le symbole chiffré qui suit).

6.1 Anti-pattern

Le travail décrit dans cette section a été réalisé avec Claude KIRCHNER et Radu KOPETZ, dans le cadre de sa thèse.

Dans cette partie nous présentons une extension de la notion de terme, appelée *anti-terme* ou *anti-pattern*, qui permet d'exprimer l'idée de *complément*. Considérons par exemple le terme $g(x)$ où x est une variable, ce terme non clos représente l'ensemble des termes clos dont le symbole de tête est g. La notion d'anti-terme permet de considérer $g(\neg a)$ par exemple, qui représente l'ensemble des termes clos dont le symbole de tête est g, mais dont l'argument *n'est pas* un a. Cette notation se généralise récursivement aux sous-termes, autorisant des négations multiples, et s'étend aux théories équationnelles.

Étant donnés un ensemble de variables $\mathcal{X}$ et un ensemble de symboles $\mathcal{F}$, la syntaxe d'un *anti-terme* est $\mathcal{AT} ::= \mathcal{X} \mid f(\mathcal{AT}, \ldots, \mathcal{AT}) \mid \neg\mathcal{AT}$, avec $f \in \mathcal{F}$ respectant son arité. L'ensemble des anti-termes est noté $\mathcal{AT}(\mathcal{F}, \mathcal{X})$, l'ensemble des anti-termes *clos* est noté $\mathcal{AT}(\mathcal{F})$. On peut remarquer que tout terme est un anti-terme : $\mathcal{T}(\mathcal{F}, \mathcal{X}) \subset \mathcal{AT}(\mathcal{F}, \mathcal{X})$.

L'opérateur $\neg$ n'agit pas exactement comme un lieur pour les variables qui apparaissent dessous. En effet, $f(x, \neg x)$ et $f(x, \neg y)$ ne représentent les mêmes ensembles de termes : le premier est constitué des termes commençant par f mais ayant des sous-termes différents, alors que le deuxième est vide. Pour un anti-pattern q donné, l'ensemble des *variables libres*, noté $\mathcal{FV}ar(q)$, correspond aux variables qui apparaissent à une position ne se trouvant pas sous un $\neg$. Le complémentaire, l'ensemble des variables non-libres, est noté $\mathcal{NFV}ar(q)$. Par exemple, nous avons $\mathcal{FV}ar(\neg q) = \emptyset$ et $\mathcal{FV}ar(f(x, \neg x)) = \{x\}$. Contrairement aux termes, l'application d'une substitution n'instancie que les variables libres. Pour un anti-pattern q donné, on appelle *grounding substitution* une substitution qui instancie toutes les variables libres de q par des *termes* clos. On note $\mathcal{GS}(q)$ l'ensemble de ces substitutions. La *sémantique close* d'un anti-terme correspond à l'ensemble des *termes clos* représentés. Une présentation détaillée est faite dans [KKM07, KKM08].

Définition 10 (sémantique close) *La* sémantique close *d'un anti-terme $q \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$ est définie récursivement par :*

$$[\![q[\neg q']_\omega]\!]_g = [\![q[z]_\omega]\!]_g \backslash [\![q[q']_\omega]\!]_g$$

où z est une variable fraîche et pour tout $\omega' < \omega$, on a $\mathcal{S}ymb(q_{|\omega'}) \neq \neg$.

Pour un terme $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, on a simplement $[\![t]\!]_g = \{\sigma(t) \mid \sigma \in \mathcal{GS}(t)\}$. En particulier $[\![x]\!]_g = \mathcal{T}(\mathcal{F})$. Cette définition, relativement simple présentée ainsi, correspond à l'idée qu'on se faisait d'un anti-pattern. Elle a également de bonne propriétés. Par exemple, on a : $\forall t \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$, $[\![\neg\neg t]\!]_g = [\![t]\!]_g$.

Exemple 12 *Dans les exemples suivants, x et z sont des variables, a, b sont des constantes, f et g sont des symboles respectivement d'arité 2 et 1.*

- $[\![\neg a]\!]_g = [\![z]\!]_g \backslash [\![a]\!]_g = \mathcal{T}(\mathcal{F}) \backslash \{a\}$, *qui représente tout sauf a,*
- $[\![\neg x]\!]_g = [\![z]\!]_g \backslash [\![x]\!]_g = \mathcal{T}(\mathcal{F}) \backslash \mathcal{T}(\mathcal{F}) = \emptyset$,
- $[\![\neg\neg x]\!]_g = [\![z]\!]_g \backslash [\![\neg x]\!]_g = [\![z]\!]_g \backslash ([\![z']\!]_g \backslash [\![x]\!]_g) = \mathcal{T}(\mathcal{F}) \backslash (\mathcal{T}(\mathcal{F}) \backslash \mathcal{T}(\mathcal{F})) = \mathcal{T}(\mathcal{F})$, *qui est l'ensemble des termes,*
- $[\![\neg g(x)]\!]_g = [\![z]\!]_g \backslash [\![g(x)]\!]_g = \mathcal{T}(\mathcal{F}) \backslash \{g(\sigma(x)) \mid \sigma \in \mathcal{GS}(g(x))\}$, *qui correspond à l'ensemble des termes privé de ceux commençant par g,*
- $[\![g(\neg x)]\!]_g = [\![g(z)]\!]_g \backslash [\![g(x)]\!]_g = \emptyset$,
- $[\![\neg g(\neg x)]\!]_g = [\![z]\!]_g \backslash [\![g(\neg x)]\!]_g = \mathcal{T}(\mathcal{F}) \backslash \emptyset = \mathcal{T}(\mathcal{F})$,

[KKM07] Claude Kirchner, Radu Kopetz, and Pierre-Etienne Moreau. Anti-pattern matching. In Rocco De Nicola, editor, *16th European Symposium on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 110–124, Braga, Portugal, 2007. Springer.

[KKM08] Claude Kirchner, Radu Kopetz, and Pierre-Etienne Moreau. Anti-pattern matching modulo. In Carlos Martín-Vide, editor, *Proceedings of the 2nd International Conference on Language and Automata Theory and Applications*, Tarragona, Spain, 2008.

– *l'utilisation d'une double négation permet de décrire l'ensemble des termes ne commencent pas par g, à moins qu'il s'agisse de $g(a)$:*

$$\begin{aligned}
[\![\neg g(\neg a)]\!]_g &= [\![z]\!]_g \backslash [\![g(\neg a)]\!]_g = [\![z]\!]_g \backslash ([\![g(z')]\!]_g \backslash [\![g(a)]\!]_g) \\
&= \mathcal{T}(\mathcal{F}) \backslash ([\![g(z')]\!]_g \backslash \{g(a)\}) \\
&= \mathcal{T}(\mathcal{F}) \backslash (\{g(\sigma(z')) \mid \sigma \in \mathcal{GS}(g(z'))\} \backslash \{g(a)\}) \\
&= \mathcal{T}(\mathcal{F}) \backslash \{g(z) \mid z \in \mathcal{T}(\mathcal{F}, \mathcal{X})\} \ \cup \ \{g(a)\},
\end{aligned}$$

– $$\begin{aligned}
[\![f(a, \neg b)]\!]_g &= [\![f(a, z)]\!]_g \backslash [\![f(a, b)]\!]_g \\
&= \{f(a, \sigma(z)) \mid \sigma \in \mathcal{GS}(f(a, z))\} \backslash \{f(a, b)\},
\end{aligned}$$

– $$\begin{aligned}
[\![\neg f(x, x)]\!]_g &= [\![z]\!]_g \backslash [\![f(x, x)]\!]_g \\
&= \mathcal{T}(\mathcal{F}) \backslash \{f(\sigma(x), \sigma(x)) \mid \sigma \in \mathcal{GS}(f(x, x))\}
\end{aligned}$$

la non-linéarité de x permet de ne pas retenir les termes commençant par f et ayant deux sous-termes identiques,

– $$\begin{aligned}
[\![f(x, \neg x)]\!]_g &= [\![f(x, z)]\!]_g \backslash [\![f(x, x)]\!]_g \\
&= \{f(\sigma(x), \sigma(z)) \mid \sigma \in \mathcal{GS}(f(x, z))\} \backslash \\
&\quad \{f(\sigma(x), \sigma(x)) \mid \sigma \in \mathcal{GS}(f(x, x))\} \\
&= f(a, b), f(a, c), f(b, c), \ldots
\end{aligned}$$

La notion de sémantique close est une contribution essentielle qui permet de définir formellement quelles sont les solutions d'un problème de filtrage. Toutes ces notions s'étendent, avec subtilité, aux symboles ayant des propriétés équationnelles telles que l'associativité, la commutativité, ou un élément neutre par exemple.
Étant donnés une théorie équationnelle $\mathcal{E}$ et deux ensembles A et B, on a :

– $t \in_{\mathcal{E}} A \ \Leftrightarrow \exists t' \in A$ tel que $t =_{\mathcal{E}} t'$;
– $A \subseteq_{\mathcal{E}} B \ \Leftrightarrow \forall t \in A$ on a $t \in_{\mathcal{E}} B$;
– $A =_{\mathcal{E}} B \ \Leftrightarrow A \subseteq_{\mathcal{E}} B$ et $B \subseteq_{\mathcal{E}} A$.

Définition 11 (sémantique close modulo) *Pour une théorie équationnelle $\mathcal{E}$ donnée, la sémantique close modulo $\mathcal{E}$ d'un terme $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ se définit naturellement par $[\![t]\!]_{g\mathcal{E}} = \{t' \mid t' \in_{\mathcal{E}} [\![t]\!]_g\}$. Pour un anti-terme $q \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$, la sémantique close modulo est définie récursivement de la façon suivante :*

$$[\![q[\neg q']_\omega]\!]_{g\mathcal{E}} = \begin{cases} [\![q[z]_\omega]\!]_{g\mathcal{E}} \backslash [\![q[q']_\omega]\!]_{g\mathcal{E}}, & \textit{si } \mathcal{FV}ar(q[\neg q']_\omega) = \emptyset \\ \bigcup_{\sigma \in \mathcal{GS}(q[\neg q']_\omega)} [\![\sigma(q[\neg q']_\omega)]\!]_{g\mathcal{E}}, & \textit{sinon} \end{cases}$$

où z est une variable fraîche et pour tout $\omega' < \omega$, on a $\mathcal{S}ymb(q_{|\omega'}) \neq \neg$.

Il est intéressant de noter que cette définition est une généralisation de la définition 10, parfaitement compatible dans le cas où $\mathcal{E}$ se ramène à la théorie vide. Dans un premier temps, nous pensions pouvoir simplement étendre la définition 10 au cas équationnel, mais cela n'était pas suffisant. En effet, lorsque f est un symbole associatif avec élément neutre, l'anti-terme $f(x, f(\neg a, y))$ correspond intuitivement à une liste contenant au moins un

élément différent de a, comme $f(b, f(a, c))$ par exemple. L'application directe de la définition 10 nous aurait conduit à $[\![f(x, f(z, y))]\!]_{g_{\mathcal{AU}}} \setminus [\![f(x, f(a, y))]\!]_{g_{\mathcal{AU}}}$, qui ne contient pas le terme $f(b, f(a, c))$. Cela vient du fait qu'avec cette approche, il est possible d'appliquer les axiomes $\mathcal{AU}$ de manière différente sur le contexte $f(x, f(\_, y))$ que nous voulions conserver. C'est précisément ce problème qui est traité par le deuxième cas introduit dans la définition 11.

Définition 12 (solutions d'une équation d'anti-filtrage) *Étant donnés $q \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$ et $t \in \mathcal{T}(\mathcal{F})$, les* solutions d'une équation d'anti-filtrage *$q \preccurlyeq_{\mathcal{E}} t$ sont définies par :*

$$Sol(q \preccurlyeq_{\mathcal{E}} t) = \{\sigma \mid t \in [\![\sigma(q)]\!]_{g_{\mathcal{E}}},\ \textit{avec } \sigma \in \mathcal{G}S(q)\}.$$

Exemple 13 *Dans les exemples suivants, x et y sont des variables, a, b, c des constantes, f est un symbole binaire, associatif avec élément neutre, g et h sont des symboles libres (sans théorie associée), respectivement unaire et binaire.*

- $Sol(h(\neg a, x) \preccurlyeq h(b, c)) = \{x \mapsto c\}$,
- $Sol(h(x, \neg g(x)) \preccurlyeq h(a, g(b))) = \{x \mapsto a\}$,
- $Sol(h(x, \neg g(x)) \preccurlyeq h(a, g(a))) = \emptyset$.

Dans le cas de la théorie associative avec élément neutre ($\mathcal{AU}$), on a :

- $Sol(f(x, f(\neg a, y)) \preccurlyeq_{\mathcal{AU}} f(b, f(a, f(c, d))) = \{x \mapsto f(b, a), y \mapsto d\}$,
- $Sol(f(x, f(\neg a, y)) \preccurlyeq_{\mathcal{AU}} f(a, f(a, a)) = \emptyset$.

L'exemple suivant filtre lorsqu'il n'y a pas de a sous un f :

- $Sol(\neg f(x, f(a, y)) \preccurlyeq_{\mathcal{AU}} f(b, f(a, f(c, d))) = \emptyset$,
- $Sol(\neg f(x, f(a, y)) \preccurlyeq_{\mathcal{AU}} f(b, f(b, f(c, d))) = \Sigma$.

La combinaison des deux exemples précédents $\neg f(x, f(\neg a, y))$, filtre naturellement vers les termes commençant par un f et ne contenant que des a immédiatement dessous :

- $Sol(\neg f(x, f(\neg a, y)) \preccurlyeq_{\mathcal{AU}} f(a, f(b, a)) = \emptyset$,
- $Sol(\neg f(x, f(\neg a, y)) \preccurlyeq_{\mathcal{AU}} f(a, f(a, a)) = \Sigma$.

La non-linéarité permet d'exprimer la contrainte all-different *sur les sous-termes de f : $Sol(\neg f(x, x) \preccurlyeq_{\mathcal{AU}} f(a, f(b, f(a, b))) = \emptyset$.*

Les définitions 10, 11 et 12 constituent des fondations théoriques décrivant clairement ce que sont les *anti-termes* et quelles sont les *solutions* d'un problème de filtrage impliquant des anti-termes. Dans la suite, nous donnons un algorithme permettant de résoudre effectivement un problème d'anti-filtrage. Pour cela, nous considérons la règle suivante :

$$\begin{array}{lllll} \mathsf{ElimAnti} & q[\neg q']_\omega \preccurlyeq_{\mathcal{E}} t & \twoheadrightarrow & \exists z\ q[z]_\omega \preccurlyeq_{\mathcal{E}} t \wedge \\ & & & \forall x \in \mathcal{FV}ar(q')\ not(q[q']_\omega \preccurlyeq_{\mathcal{E}} t) \\ & & \mathsf{si} & z \text{ est frais et } \forall\ \omega' < \omega,\ \mathcal{S}ymb(q_{|\omega'}) \neq \neg \end{array}$$

Cette règle élimine les négations $\neg$ en transformant chaque problème d'anti-filtrage en deux problèmes ayant chacun un symbole $\neg$ en moins. L'application de cette règle termine et conserve les solutions du problème initial. C'est un moyen systématique de transformer un problème de filtrage sur les anti-termes en un problème équationnel pour lequel il existe des algorithmes de résolution connus, comme ceux de disunification par exemple [CL90].

Proposition 2 *La règle* ElimAnti *est correcte et complète modulo $\mathcal{E}$: aucune solution n'est perdue ou introduite par application de cette règle.*

Proposition 3 *Un problème de filtrage sur les anti-termes peut toujours être transformé, en un nombre fini d'étapes, en un problème équationnel équivalent.*

Cette approche est intéressante mais n'est pas très efficace en pratique. En effet, l'application de la règle ElimAnti peut produire 2^n problèmes équationnels, où n correspond au nombre de négations dans le problème initial. Par ailleurs, les algorithmes de disunification sont trop généraux et permettent de résoudre des problèmes plus complexes que ceux résultant de l'application de ElimAnti. De ce fait, leur application est non optimale dans les cas qui nous intéressent, en particulier dans les cas de la théorie vide ou des théories associatives avec élément neutre par exemple. Comme l'illustre l'ensemble de règles $\mathcal{AU}$-AntiMatching suivant, où la règle ElimAnti doit être appliquée en priorité, des algorithmes plus efficaces peuvent être dérivés pour résoudre des problèmes d'anti-filtrage. Partant d'un algorithme de filtrage $\mathcal{AU}$ classique, nous l'avons enrichi par quelques règles permettant de traiter les anti-patterns :

ElimAnti	$q[\neg q']_\omega \ll_{\mathcal{AU}} t$	$\mapsto\!\!\!\!\rightarrow$	$\exists z\ q[z]_\omega \ll_{\mathcal{AU}} t \wedge$ $\forall x \in \mathcal{FV}ar(q')\ not(q[q']_\omega \ll_{\mathcal{AU}} t)$
		si	z est frais et $\forall\ \omega' < \omega,\ \mathcal{S}ymb(q_{\|\omega'}) \neq \neg$
QTransform	$\forall \bar{y}\ not(\mathsf{D})$	$\mapsto\!\!\!\!\rightarrow$	$not(\exists \bar{y}\ \mathsf{D})$
NotOr	$not(\mathsf{D}_1 \vee \mathsf{D}_2)$	$\mapsto\!\!\!\!\rightarrow$	$not(\mathsf{D}_1) \wedge not(\mathsf{D}_2)$
NotTrue	$not(\top)$	$\mapsto\!\!\!\!\rightarrow$	$\bot$
NotFalse	$not(\bot)$	$\mapsto\!\!\!\!\rightarrow$	$\top$

Règles provenant du filtrage $\mathcal{AU}$:

Mutate	$f(p_1, p_2) \ll_{\mathcal{AU}} f(t_1, t_2)$	$\mapsto\!\!\!\!\rightarrow$	$(p_1 \ll_{\mathcal{AU}} t_1 \wedge p_2 \ll_{\mathcal{AU}} t_2) \vee$ $\exists x(p_2 \ll_{\mathcal{AU}} f(x, t_2) \wedge f(p_1, x) \ll_{\mathcal{AU}} t_1) \vee$ $\exists x(p_1 \ll_{\mathcal{AU}} f(t_1, x) \wedge f(x, p_2) \ll_{\mathcal{AU}} t_2)$
SClash_1^+	$f(p_1, p_2) \ll_{\mathcal{AU}} a$	$\mapsto\!\!\!\!\rightarrow$	$(p_1 \ll_{\mathcal{AU}} e_f \wedge p_2 \ll_{\mathcal{AU}} a) \vee$ $(p_1 \ll_{\mathcal{AU}} a \wedge p_2 \ll_{\mathcal{AU}} e_f)$
SClash_2^+	$a \ll_{\mathcal{AU}} f(p_1, p_2)$	$\mapsto\!\!\!\!\rightarrow$	$(e_f \ll_{\mathcal{AU}} p_1 \wedge a \ll_{\mathcal{AU}} p_2) \vee$ $(a \ll_{\mathcal{AU}} p_1 \wedge e_f \ll_{\mathcal{AU}} p_2)$
CstClash	$a \ll_{\mathcal{AU}} b$	$\mapsto\!\!\!\!\rightarrow$	$\bot$ si $a \neq b$
Replt	$z \ll_{\mathcal{AU}} t \wedge \mathsf{S}$	$\mapsto\!\!\!\!\rightarrow$	$z \ll_{\mathcal{AU}} t \wedge \{z \mapsto t\}\mathsf{S}$ si $z \in \mathcal{FV}ar(\mathsf{S})$

Règles utilitaires :

Delete	$p \prec\!\!<_{\mathcal{AU}} p$	$\mapsto\!\!\!\!\rightarrow$	$\top$
Exists_1	$\exists z(\mathsf{D}[z \prec\!\!<_{\mathcal{AU}} t])$	$\mapsto\!\!\!\!\rightarrow$	$\mathsf{D}[\top]$ si $z \notin \mathcal{V}ar(\mathsf{D}[\top])$
Exists_2	$\exists z(\mathsf{S}_1 \vee \mathsf{S}_2)$	$\mapsto\!\!\!\!\rightarrow$	$\exists z(\mathsf{S}_1) \vee \exists z(\mathsf{S}_2)$
DistribAnd	$\mathsf{S}_1 \wedge (\mathsf{S}_2 \vee \mathsf{S}_3)$	$\mapsto\!\!\!\!\rightarrow$	$(\mathsf{S}_1 \wedge \mathsf{S}_2) \vee (\mathsf{S}_1 \wedge \mathsf{S}_3)$
$\mathsf{PropagClash}_1$	$\mathsf{S} \wedge \bot$	$\mapsto\!\!\!\!\rightarrow$	$\bot$
$\mathsf{PropagClash}_2$	$\mathsf{S} \vee \bot$	$\mapsto\!\!\!\!\rightarrow$	S
$\mathsf{PropagSuccess}_1$	$\mathsf{S} \wedge \top$	$\mapsto\!\!\!\!\rightarrow$	S
$\mathsf{PropagSuccess}_2$	$\mathsf{S} \vee \top$	$\mapsto\!\!\!\!\rightarrow$	$\top$

Proposition 4 *L'application des règles* $\mathcal{AU}$-AntiMatching *est correcte et complète.*

Théorème 3 *Les formes normales de* $\mathcal{AU}$-AntiMatching *sont des problèmes de filtrage* $\mathcal{AU}$ *en forme résolue.*

Cet algorithme est utilisable en pratique mais il faut noter que la règle ElimAnti produit un nombre exponentiel de sous-problèmes à résoudre. Nous avons considéré une sous-classe de motifs, appelée $Pure\mathcal{FV}ars$, correspondant aux anti-termes tels qu'à toute position, une variable libre dans un sous-terme ne peut pas apparaître sous un $\neg$ dans un autre sous-terme. Plus formellement, étant donné $q \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$, on a $q \in Pure\mathcal{FV}ars$ si et seulement si $\forall i, j$ avec $i \neq j$, et pour un terme de la forme $C[f(t_1, \ldots, t_i, \ldots, t_j, \ldots, t_n)]$, on a $\mathcal{FV}ar(t_i) \cap \mathcal{NFV}ar(t_j) = \emptyset$. Par exemple, $f(x, x) \in Pure\mathcal{FV}ars$, $f(\neg x, \neg x) \in Pure\mathcal{FV}ars$, mais $f(x, \neg x) \notin Pure\mathcal{FV}ars$. Pour résoudre les problèmes correspondant à cette classe de motifs, la règle ElimAnti peut être remplacée par la règle suivante, sans plus aucune priorité particulière :

$$\mathsf{EfficientElimAnti} \quad \neg q \prec\!\!<_{\mathcal{AU}} t \quad \mapsto\!\!\!\!\rightarrow \quad \forall x \in \mathcal{FV}ar(q)\ not(q \prec\!\!<_{\mathcal{AU}} t)$$

La correction de cet algorithme a également été établie. Pour résoudre un problème général, il suffit de déterminer si le motif appartient à la classe $Pure\mathcal{FV}ars$ et de choisir l'algorithme à appliquer en conséquence : $\mathcal{AU}$-Matching ou $\mathcal{AU}$-Matching dans lequel la règle ElimAnti est remplacée par la règle EfficientElimAnti.

Nous conjecturons qu'un même algorithme pourrait être utilisé dans tous les cas : il s'agit du deuxième algorithme dans lequel la règle EfficientElimAnti serait modifiée de telle sorte que seules les variables respectant la condition $\mathcal{FV}ar(t_i) \cap \mathcal{NFV}ar(t_j) = \emptyset$ (de la définition de $Pure\mathcal{FV}ars$) seraient quantifiées universellement ; l'algorithme resterait correct et complet. Par exemple, lors de l'application de EfficientElimAnti à $f(x, \neg x)$, la variable x ne serait plus quantifiée universellement. Cet algorithme a été implanté et expérimenté sans faire apparaître de contre-exemple. Nous travaillons sur la preuve formelle de sa correction.

Exemple 14 *La notion d'anti-pattern a été entièrement intégrée au langage* Tom. *Le symbole* `!` *est utilisé pour représenter un complément. Considérons par exemple l'extrait de programme suivant :*

```
%match(s,s) {
  list(_*,x,_*), !list(_*,x,_*,x,_*) -> {
    System.out.println(x);
  }
}
```

La règle se déclenche pour tous les éléments de la liste `s` *qui n'apparaissent pas plus d'une fois. Si* `s` *correspond au terme* `list(1,2,1,3,2,1,5)`*, le programme précédent n'affiche que* `3` *et* `5`*.*

Comme dans les langages fonctionnels, notons que la virgule qui sépare les deux motifs a le même sens qu'une virgule séparant les sous-termes dans un motif. Ainsi, le premier motif sélectionne un élément de `s` *et le deuxième vérifie que cet élément n'apparaît pas au moins deux fois.*

Grace aux travaux de Radu Kopetz, les membres gauches des règles de Tom ne sont plus restreints à de simples *patterns*. Des conjonctions et des disjonctions de contraintes de filtrage peuvent également être utilisées. Cette extension rend plus naturelle l'utilisation d'anti-patterns. Récemment, nous avons modélisé les politiques de sécurité de Bell et LaPadula [BL73] ainsi que celle de McLean [McL88], afin d'étudier la fuite éventuelle d'information. Dans cette application nous utilisons une règle permettant de trouver des *lectures* qui seraient faites de manière *implicite* :

```
%strategy makeExplicit() extends Identity() {
  visit State {
    s@state(_*,read(s1,o1,_),_*,read(s2,o2,_),_*,write(s1,o2,_),_*)
    && !state(_*,read(s2,o1,_),_*) << s -> {
      return `state(read(s2,o1,implicit()),s);
    }
    ...
  }
}
```

Dans cet exemple, l'anti-pattern `!state(_*,read(s2,o1),_*)` vérifie que l'état courant `s` ne contient pas d'accès en lecture du sujet `s2` sur l'objet `o1`, ceci avec `s1`, `s2`, `o1` et `o2` vérifiant les propriétés caractérisées par le premier filtre. À savoir `s1` accédant en lecture à `o1`, `s2` accédant en lecture à `o2` et `s1` écrivant dans `o2`. L'anti-pattern exprime plus qu'une simple non-appartenance à la liste `s` dans la mesure où le troisième argument de `read(s2,o1,_)` est une variable (anonyme).

À RETENIR :

Les anti-patterns sont une extension des termes où l'opérateur $\neg$ dénote un complément. Nous définissons la sémantique close d'un anti-pattern q, notée $[\![q]\!]_g$, comme étant l'ensemble des termes clos défini par $[\![q[\neg q']_\omega]\!]_g = [\![q[z]_\omega]\!]_g \backslash [\![q[q']_\omega]\!]_g$. La résolution d'un problème d'anti-filtrage peut se faire en transformant les équations de filtrage impliquant un anti-pattern en une conjonction de problèmes de disunification, pour lesquelles il existe des algorithmes de résolution connus. Cette approche est étendue au cas équationnel et nous proposons un algorithme plus efficace pour la sous-classe $PureF\mathcal{V}ars$, correspondant aux anti-patterns tels qu'à toute position, une variable libre dans un sous-terme ne peut pas apparaître sous un $\neg$ dans un autre sous-terme

6.2 Filtrage modulo

Le travail décrit dans cette section a été réalisé avec Antoine REILLES, dans le cadre de sa thèse.

GOM permet de définir des opérateurs variadiques, utiles pour représenter les listes. TOM offre un filtrage dit *associatif*, permettant de rechercher et de filtrer des éléments dans ces structures. Nous avons naturellement cherché à savoir si le filtrage offert par TOM correspondait bien au filtrage « classique » sur les termes modulo les axiomes d'associativité et de neutralité. La réponse est *oui*, sous certaines conditions.

Afin d'apporter des précisions à cette réponse, commençons par définir quelques notions. Un opérateur binaire $f : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ est dit *associatif* s'il satisfait la proposition :

$$\forall x, y, z \in \mathcal{T}, f(f(x, y), z) = f(x, f(y, z))$$

La constante $e \in \mathcal{T}$ est *neutre à gauche* pour l'opérateur binaire $f : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ si elle satisfait la proposition : $\forall x \in \mathcal{T}, f(e, x) = x$. Elle est dite *neutre à droite* si elle satisfait $\forall x \in \mathcal{T}, f(x, e) = x$. La constante e est élément *neutre* pour l'opérateur binaire f si elle est neutre à gauche et à droite pour cet opérateur.

Une *signature associative* est composée d'un ensemble de symboles de fonction $\mathcal{F}$ d'arité fixe et d'une liste de couples (f, e), avec $f \in \mathcal{F}$ binaire et $e \in \mathcal{F}$ constante, tels que f est associatif et e est son élément neutre. On notera $\mathcal{F}_{\mathcal{AU}}$ cette signature, et $\mathcal{T}(\mathcal{F}_{\mathcal{AU}})$ l'ensemble des termes engendrés par cette signature. On notera $\mathcal{AU}$ la théorie équationnelle engendrée par les équations d'associativité et de neutralité de ces opérateurs. On peut alors définir une relation d'équivalence $=_{\mathcal{AU}}$ sur les termes de $\mathcal{T}(\mathcal{F}_{\mathcal{AU}})$ telle que les termes égaux modulo associativité et élément neutre sont égaux.

Un *opérateur variadique* est un symbole de fonction pour lequel la fonction arité n'est pas définie. Il peut prendre un nombre quelconque d'arguments. On peut voir la définition d'un opérateur variadique f_v comme la définition d'une famille d'opérateurs algébriques $\{f_n \mid n \in \mathbb{N}, ar(f_n) = n\}$. Une

signature variadique est composée d'un ensemble de symboles de fonctions $\mathcal{F}$ d'arité fixe, ainsi que d'un ensemble de symboles de fonction variadiques. On notera $\mathcal{F}_v$ cette signature, et $\mathcal{T}(\mathcal{F}_v)$ l'ensemble des termes engendrés par cette signature.

On peut définir des notions similaires à l'associativité et à l'élimination des éléments neutres pour les opérateurs variadiques. Soit f_v un opérateur variadique. On dit qu'il satisfait l'équation d'*aplatissement* si :

$$\forall t_1, \ldots, t_n, \quad f_v(t_1, \ldots, t_i, \ldots, t_n) = f_v(t_1, \ldots, t_{i-1}, {t'}_1, \ldots, {t'}_{n'}, t_{i+1}, \ldots, t_n), \\ \text{avec } t_i = f_v({t'}_1, \ldots, {t'}_{n'}).$$

f_v satisfait l'équation de *suppression des vides* si :

$$\forall t_1, \ldots, t_n, \quad f_v(t_1, \ldots, t_{i-1}, f_v(), t_{i+1}, \ldots, t_n) = f_v(t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n)$$
$$\forall t \text{ tel que } \forall v_1, \ldots, v_k, t \neq f_v(v_1, \ldots, v_k), \text{ on a : } f_v(t) = t.$$

On peut alors définir une relation d'équivalence $=_v$ sur les termes de $\mathcal{T}(\mathcal{F}_v)$ comme l'équivalence modulo aplatissement et suppression des vides.

Les équations de l'associativité et élément neutre pour les termes contenant des symboles associatifs ou neutres définissent une notion de classe d'équivalence, dans laquelle on peut choisir un représentant canonique. On peut faire de même pour les termes contenant des symboles variadiques. Dans les deux cas, il s'agit d'orienter les équations d'une part d'associativité et éléments neutres (système de réécriture appelé R), et d'autre part d'aplatissement et élimination des vides (système de réécriture appelé R_v).

La figure 6.1 montre comment passer d'un terme *associatif* à un terme *variadique* : il suffit de pourvoir interpréter les termes de $\mathcal{T}(\mathcal{F}_{\mathcal{AU}})$. Pour cela on se donne une fonction de représentation variadique $[\![\ ]\!]$ et une fonction auxiliaire $[\ ]_{aux}$:

$$[\![\ ]\!] \begin{cases} [\![e]\!] & \rightarrow & f_v() \\ [\![f(x,y)]\!] & \rightarrow & f_v([\![x]\!], [y]_{aux}) \\ [\![g(t_1, \ldots, t_n)]\!] & \rightarrow & g([\![t_1]\!], \ldots, [\![t_n]\!]), \forall g \in \mathcal{F} \end{cases}$$

$$[\ ]_{aux} \begin{cases} [f(x,y)]_{aux} & \rightarrow & [\![x]\!], [y]_{aux} \\ [e]_{aux} & \rightarrow & f_v() \\ [g(t_1, \ldots, t_n)]_{aux} & \rightarrow & g([\![t_1]\!], \ldots, [\![t_n]\!]), \forall g \in \mathcal{F}. \end{cases}$$

Théorème 4 *La fonction de représentation variadique $[\![\ ]\!]$ est injective :*

$$\forall t_1, t_2 \in \mathcal{T}(\mathcal{F}_{\mathcal{AU}}), [\![t_1]\!] = [\![t_2]\!] \Rightarrow t_1 = t_2.$$

D'un point de vue un peu plus formel on peut définir les fonctions *peigne* (application orientée de l'axiome d'associativité), *supprA* (élimination des neutres), *aplat* (aplatissement des symboles variadiques), *supprV* (élimination des vides), et on peut montrer des propriétés de commutation.

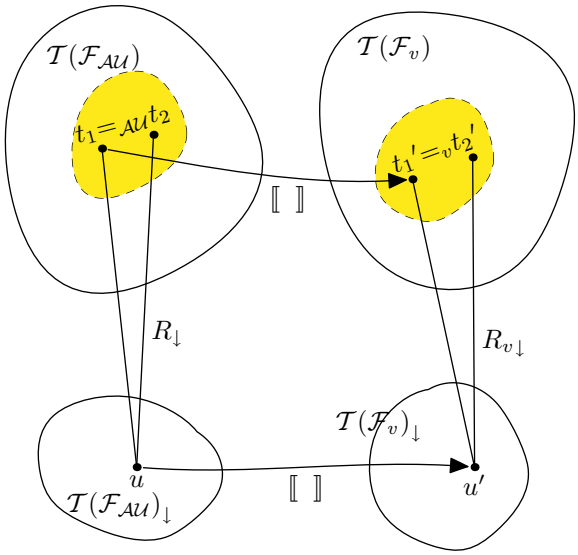


FIGURE 6.1 – *Représentation informelle des liens entre termes associatifs et termes variadiques. Deux termes t_1 et t_2 sont égaux modulo $\mathcal{AU}$ si les formes normales par rapport à R sont identiques. Il en est de même pour deux termes variadiques t_1' et t_2', en utilisant le système R_v. Le point important ici est que la fonction de représentation $[\![\ ]\!]$ met bien en correspondance les formes normales u et u'.*

Théorème 5 (commutations) *Les fonctions peigne et aplat commutent avec la fonction de représentation variadique :*

$$\forall t \in \mathcal{T}(\mathcal{F}_{\mathcal{AU}}), aplat([\![t]\!]) = [\![peigne(t)]\!].$$

Les fonctions supprA et supprV commutent avec la fonction de représentation variadique :

$$\forall t \in \mathcal{T}(\mathcal{F}_{\mathcal{AU}}), supprV([\![t]\!]) = [\![supprA(t)]\!].$$

On a alors la commutation de la normalisation associative avec élément neutre, et de la normalisation par aplatissement et élimination des éléments vides avec la fonction de représentation variadique :

$$\forall t \in \mathcal{T}(\mathcal{F}_{\mathcal{AU}}), supprV(aplat([\![t]\!])) = [\![supprA(peigne(t))]\!].$$

On peut résumer ce théorème par le diagramme suivant :

$$\begin{array}{ccc} t & \xrightarrow{[\![\,]\!]} & [\![t]\!] \\ \downarrow peigne & & \downarrow aplat \\ peigne(t) & \xrightarrow{[\![\,]\!]} & aplat([\![t]\!]) \\ \downarrow supprA & & \downarrow supprV \\ supprA(peigne(t)) & \xrightarrow{[\![\,]\!]} & supprV(aplat([\![t]\!])) \end{array}$$

Théorème 6 (complétude $supprA \circ peigne$**)**

$$\forall t_1, t_2 \in \mathcal{T}(\mathcal{F}_{\mathcal{AU}}), t_1 =_{\mathcal{AU}} t_2 \Rightarrow supprA(peigne(t_1)) = supprA(peigne(t_2)).$$

Théorème 7 (correction $supprA \circ peigne$**)**

$$\forall t \in \mathcal{T}(\mathcal{F}_{\mathcal{AU}}), t =_{\mathcal{AU}} supprA(peigne(t)).$$

Théorème 8 (complétude $supprV \circ aplat$**)**

$$\forall t_1, t_2 \in \mathcal{T}(\mathcal{F}_{\mathcal{AU}}), t_1 =_{\mathcal{AU}} t_2 \Rightarrow supprV(aplat([\![t_1]\!])) = supprV(aplat([\![t_2]\!])).$$

Théorème 9 (correction $supprV \circ aplat$**)**

$$\forall t_1 \in \mathcal{T}(\mathcal{F}_{\mathcal{AU}}), \exists t_2 \in \mathcal{T}(\mathcal{F}_{\mathcal{AU}}), supprV(aplat([\![t_1]\!])) = [\![t_2]\!] \wedge t_1 =_{\mathcal{AU}} t_2.$$

L'utilisation de ces théorèmes nous garantit que les termes de $\mathcal{T}(\mathcal{F}_{\mathcal{AU}})$ d'une même classe d'équivalence sont projetés par la normalisation associative et élément neutre vers un représentant canonique unique. Cette normalisation est implémentée par la combinaison de fonctions $supprA \circ peigne$. Les représentations variadiques de ces termes sont quant à elles projetées par aplatissement et élimination des vides vers un représentant canonique dans $\mathcal{T}(\mathcal{F}_v)$ unique. Cette normalisation est implémentée par $supprV \circ aplat$. Ce représentant est par ailleurs égal à la représentation variadique du représentant canonique des termes initiaux.

Dans cette section, nous avons montré les liens entre une algèbre de termes comprenant des symboles associatifs ainsi que des éléments neutres pour ces symboles associatifs et une algèbre de termes comprenant des symboles variadiques.

Les formes normales modulo associativité et élément neutre sont alors mises en relation avec les formes normales des termes variadiques correspondant. Ainsi, le calcul des formes normales modulo aplatissement et suppression des vides correspond au calcul des formes normales pour l'associativité et élément neutre. On peut alors raisonner sur des algèbres de termes associatives en utilisant comme représentation concrète des objets variadiques, tels que ceux générés par Gom.

La certification des corrélations entre termes associatifs et termes variadiques permet alors de donner une base formelle pour la preuve d'algorithmes manipulant des termes modulo associativité et élément neutre, implantés en utilisant des structures associatives.

À RETENIR :

Un terme $f_v(t_1, \ldots, t_n)$, où f_v est un opérateur variadique, peut être vu comme un opérateur associatif avec élément neutre. Nous avons défini la fonction de représentation ($[\![\ ]\!]$) d'un terme associatif en un terme variadique ainsi que des fonctions de mise en forme canonique par aplatissement et élimination des neutres. Nous avons également montré des propriétés d'injection, de commutation, de correction et de complétude, garantissant que les termes $\mathcal{AU}$ et que les termes variadiques ont bien des représentants canoniques identiques, à $[\![\ ]\!]$ près.

6.3 Stratégies

Le travail décrit dans cette section a été réalisé en partie avec Antoine REILLES, dans le cadre de sa thèse.

Les notions de filtrage et de règle de réécriture sont particulièrement bien adaptées pour décrire une transformation élémentaire à effectuer. Lorsque la transformation devient plus complexe, bien que possible, il n'est pas raisonnable de se limiter seulement à ces deux notions pour décrire et contrôler l'ordre et le lieu d'application des règles. En programmation fonctionnelle par exemple, les calculs élémentaires sont effectués par des fonctions ; le contrôle est décrit par des fonctions prenant d'autres fonctions en argument. Dans les langages de programmation basés exclusivement sur la réécriture, cette fonctionnalité d'ordre supérieur est souvent absente. Pour répondre à ce besoin, la notion de *langage de stratégie* a été développée. La conception et l'implantation de tels langages est un sujet qui m'intéresse et sur lequel je travaille depuis le début de ma thèse.

Partant de l'expérience acquise au cours du projet Elan [Bor98], et en étudiant les travaux faits depuis sur Maude [Cla98, MOMV04], Stratego [VB98, VBT98], et JJTraveler [Vis01], j'ai essayé de concevoir un nouveau langage de stratégie qui soit à la fois expressif, possible à implanter sous forme de librairie Java, et compatible avec le cadre des îlots formels. En terme de fonctionnalité nos contraintes sont assez fortes parce que nous voulons pouvoir décrire aussi bien des parcours d'arbre tels que *leftmost-innermost*, que des stratégies d'exploration comme *breadth-first search*. Le premier type de parcours nécessite la présence d'opérateurs de congruence et de récursion permettant de descendre de manière générique dans les sous-termes. Le deuxième type d'exploration, propre à Elan, existant sous une forme assez limitée dans Maude, et inexistant dans la plupart des autres langages basés sur la réécriture, nécessite d'avoir des opérateurs de stratégie retournant plusieurs résultats ou

Combinateur	Sémantique
`Identity`	ne fait rien, mais n'échoue pas
`Fail`	échoue toujours
v_1 `;` v_2	applique v_1, puis v_2. Échoue si l'un des deux échoue
v_1 `<+` v_2	applique v_1, puis v_2 si v_1 échoue
`All(`v`)`	applique v à tous les fils directs
`One(`v`)`	applique v à un des fils tel que cela réussisse
`Not(`v`)`	applique v. Échoue si v s'applique, renvoie l'identité sinon
`Omega(`i,v`)`	applique v au i-ème fils s'il existe, échoue sinon

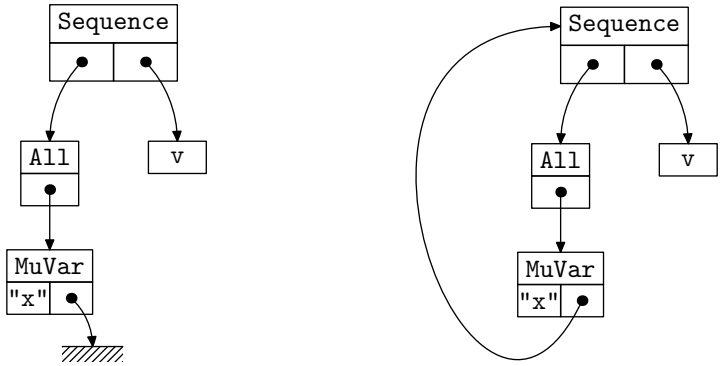
FIGURE 6.2 – *Les opérateurs élémentaires peuvent être combinés pour décrire des stratégies plus complexes. Ainsi, la répétition peut se définir par* `repeat`(v) = $(v$`;repeat(`v`))` `<+ Identity` *ou encore par* `repeat`(v) = $\mu x.(v$`;`x`)` `<+ Identity` *en utilisant l'opérateur de récursion* μ*. L'application* bottom-up *d'une stratégie se définie par* `bottom-up`(v) = μx.`All(`x`)` `;` v*. Il est important de noter que* `One` *et* `All` *sont des opérateurs génériques qui permettent de « descendre » sous n'importe quel symbole de fonction.*

un multi-ensemble de résultats. Combiner ces deux types de recherche pour calculer toutes les réductions possibles d'un terme (à toutes les positions, et pour tous les filtres modulo $\mathcal{AU}$) constituait un véritable défi. Nous sommes partis du jeu de combinateurs décrit figure 6.2, inspiré d'Elan et de Stratego, ainsi que des techniques d'implantation décrites dans [Vis01]. Notre première contribution a été de créer un ancrage formel pour ces combinateurs et de rendre l'opérateur de récursion μ explicite. Ainsi, toute expression de stratégie peut s'exprimer directement en utilisant le « ‘ » de TOM. Cette construction étant restreinte à une notation préfixée, `Sequence` correspond au « ; », `mu(MuVar("x")` correspond à la construction μx :

```
Strategy BottomUp(Strategy v) {
  return ‘mu(MuVar("x"), Sequence(All(MuVar("x")),v));
}
```

Cette approche illustre la puissance et l'expressivité du concept d'îlot formel : toute stratégie est un objet du langage qui est vu comme un *terme*. Le premier avantage est de permettre le passage de stratégie en paramètre. Le second est de pouvoir construire dynamiquement une stratégie, filtrer vers une stratégie, ou encore appliquer une stratégie sur une stratégie.

Rendre l'opérateur de récursion μ explicite signifie que la construction `mu(MuVar("x")` peut être utilisée pour dénoter un appel récursif. À l'exécution, une stratégie est un « graphe » cyclique qui est parcouru. Une difficulté de ce travail a été de permettre au programmeur d'utiliser une notation telle que `‘mu(MuVar("x")` pour dénoter un appel récursif. Le « terme » construit



(A) *Avant μ-expansion : arbre représentant* BottomUp

(B) *Après μ-expansion : graphe représentant* BottomUp

FIGURE 6.3 – *Graphes de stratégies avec le combinateur* `MuVar`

par le « ‘ » devant alors être transformé *just in time* en un « graphe » comme illustré figure 6.3. Une stratégie étant un terme, cette transformation, appelée *μ-expansion*, est naturellement écrite en TOM en utilisant une stratégie *amorce* directement écrite en Java.

Nous avons jusqu'ici présenté un langage de stratégies permettant de décrire des traversées d'arbre génériques. Ces stratégies s'expriment à l'aide d'opérateurs élémentaires tels que `All`, `One` et `Omega`, permettant d'accéder de manière générique aux sous-termes, ainsi qu'un opérateur de récursion μ. Pour rendre le langage de stratégies utile, il faut encore pouvoir définir des stratégies élémentaires décrivant les transformations ou actions à effectuer lors de la traversée.

Une première possibilité consiste à hériter de certaines classes de la bibliothèque et à écrire le code directement en Java, comme illustré ci-dessous :

```
public static class Rule extends TFwd {
  public Rule() {
    super(new Fail());
  }
  public T visit_T(T arg) throws jjtraveler.VisitFailure {
    %match(T arg) {
      f(x,x) -> { return ‘g(x); }
    }
  }
}
```

Cette approche rend cependant fastidieuse l'écriture de stratégies élémentaires, ce qui va à l'encontre de notre volonté de rendre les méthodes formelles faciles à utiliser. Nous avons donc étendu le langage TOM afin qu'il

génère lui-même l'enrobage « administratif ». La définition de la stratégie élémentaire correspondant à la règle $f(x,x) \rightarrow g(x)$ devient alors :

```
%strategy Rule() extends Fail() {
  visit T {
    f(x,x) -> g(x)
  }
}
```

Dans cet exemple, `Rule()` est le nom de la stratégie. Il peut être combiné avec d'autres stratégies, comme `Innermost(Rule())`, pour créer des stratégies plus complexes. L'application d'une stratégie se fait en appliquant la méthode `.visit(...)` comme illustré ci-dessous :

```
Term subject = `g(f(a,a));
Term result = `Innermost(Rule()).visit(subject);
```

Le langage de stratégie que nous venons de présenter est une première réponse au besoin de contrôle inhérent à la programmation par règle. L'ensemble de constructeurs proposé est facilement extensible (il faut en général moins de 20 lignes de Java pour implanter une nouvelle brique) et permet de décrire simplement en Java la plupart des stratégies de *parcours*. Notre deuxième contribution a été d'étendre ce langage pour permettre la description de stratégies d'*exploration*.

En Elan (ou en Prolog), l'application d'une règle peut produire plusieurs résultats, chacun étant calculé de manière paresseuse en fonction des besoins. En Tom, le modèle de calcul est différent car l'application d'une règle ne retourne pas de résultat mais exécute une *action*. Cette action peut bien entendu retourner un résultat ou calculer un ensemble de résultats, en ajoutant un résultat dans un ensemble par exemple, mais elle pourrait faire tout autre chose. Cette différence subtile de modèle de calcul rend difficile, voire impossible, la gestion implicite d'un ensemble de résultats par le compilateur Tom. Ce choix a cependant été fait parce qu'il offre plus de souplesse aux utilisateurs et rend l'implantation du compilateur Tom plus simple.

Afin de pouvoir effectuer des calculs non-déterministes, et ainsi décrire des stratégies d'exploration, nous proposons une solution relativement simple, mais particulièrement élégante et expressive : celle-ci consiste à rendre utilisable une notation largement utilisée dans la littérature, à savoir $t_{|\omega}$ ou $t[u]_\omega$ par exemple.

Dans cette notation, ω représente une position dans un terme. Cette position étant le résultat d'un calcul ou d'une recherche. Nous avons donc étendu Tom pour que cette notion de position puisse être explicitement représentée (pensez à une liste d'entiers par exemple). Nous avons également étendu le langage de stratégie, c.-à-d. chaque combinateur élémentaire, pour que la *position* où la stratégie s'applique soit implicitement calculée. Ainsi,

dans une stratégie décrite à l'aide de `%strategy`, il est possible d'accéder à la position ω où cette stratégie est appliquée.

Exemple 15 *Considérons à nouveau la définition de la stratégie* `Rule()`*, dans laquelle nous remplaçons la partie droite de* `f(x,x) -> g(x)` *par l'action* `Java` *suivante :* `{ print(getPosition()); return `g(x); }`*. L'application de* `Innermost(Rule())` *à $f(a,a)$ retourne bien $g(a)$, mais affiche également* `[0]`*, correspondant à la représentation de la position racine, où est appliquée la règle.*

En appliquant cette même stratégie au terme $f(g(a), f(a,a))$, le résultat est $g(g(a))$ (après deux réductions) et deux positions sont affichées : `[2]` *et* `[0]`*, correspondant aux applications sur $f(a,a)$ et sur le terme résultant de cette première réduction, à savoir $f(g(a), g(a))$.*

Une position `p` est un objet `Java` qui peut être affiché, mémorisé dans une collection, ou encore utilisé pour construire une stratégie. Ainsi, dans la bibliothèque de base, nous avons prédéfini (dans la classe `Position`) deux méthodes (`getSubterm` et `getReplace`) qui retournent des stratégies dont l'application correspond à l'accès au sous-terme et à la greffe. L'expression `p.getSubterm().visit(t)` calcule $t_{|p}$, alors que `p.getReplace(u).visit(t)` retourne $t[u]_p$. La manière dont sont implantées ces méthodes n'a pas à être connue des utilisateurs. Nous les détaillons seulement pour montrer l'intérêt de pouvoir manipuler explicitement les positions et les stratégies comme des objets :

```
public Strategy getReplace(final Object u) {
  Strategy s = new AbstractStrategy() {
               public Object visit(Object x) {
                 return u;
               }
             };
  return getOmega(s);
}
```

L'expression `getOmega(s)` retourne une stratégie qui applique son paramètre `s` à la position courante. Ainsi, l'appel de la méthode `getOmega(s)` sur la position `1.2`, retourne la stratégie `Omega(1,Omega(2,s))`, dont l'application descend à la position `1.2` avant d'appliquer `s`. De manière générale, la stratégie retournée par `getOmega` est une simple imbrication de combinateurs élémentaires `Omega`. Dans notre exemple, le paramètre `s` est la stratégie élémentaire correspondant à $x \to u$.

L'encodage de la stratégie retournée par `p.getSubterm()` est un peu plus subtil. En effet, l'application de la stratégie `p.getOmega(s)` à un terme `t` retourne $\mathtt{t}[\mathtt{s.visit}(\mathtt{t}_{|\mathtt{p}})]_{\mathtt{p}}$. Dans notre cas, la difficulté vient du fait qu'il faut retourner $\mathtt{t}_{|\mathtt{p}}$ et d'une certaine façon se *débarrasser* du contexte. Pour cela, nous utilisons l'encodage suivant :

```
public Strategy getSubterm() {
  return new AbstractStrategy() {
    public Object visit(Object x) {
      final Object[] ref = new Object[1];
      Strategy s = new AbstractStrategy() {
                    public Object visit(Object y) {
                      ref[0] = y;
                      return y;
                  };
      Object tmp = getOmega(s).visit(subject);
      return ref[0];
    }
  }
}
```

L'idée consiste à créer une stratégie `s` qui mémorise dans la variable `ref[0]` (par effet de bord) le terme sur lequel elle est appliquée. La variable `tmp` contient bien `subject`, mais l'évaluation de `getOmega(s).visit(subject)` a mémorisé dans `ref[0]` le sous-terme $\texttt{subject}_{|\texttt{this}}$. La méthode `getSubterm` n'a plus alors qu'à retourner la stratégie correspondant à $x \rightarrow \texttt{ref}[0]$.

Exemple 16 *Dans cet exemple nous montrons comment l'utilisation des* positions *permet d'encoder des calculs non-déterministes et de collecter un ensemble de résultats. Considérons le système de réécriture $\mathcal{R}$ défini par :*

$$\mathcal{R} = \left\{ \begin{array}{lcl} g(x) & \rightarrow & g(s(x)) \\ g(s(x)) & \rightarrow & g(g(x)) \end{array} \right.$$

On peut définir une stratégie `Collect`*, paramétrée par une* `Collection`*, qui mémorise dans cette structure les successeurs du terme sur lequel est appliquée une règle de $\mathcal{R}$.*

```
%strategy Collect(t:T,c:Collection) extends Identity() {
  visit T {
    g(x) -> {
      c.add(getPosition().getReplace(`g(s(x))).visit(t));
    }
    g(s(x)) -> {
      c.add(getPosition().getReplace(`g(g(x))).visit(t));
    }
  }
```

Étant donné un terme `t`*, l'expression* `` `BottomUp(Collect(t,c)).visit(t) `` *calcule l'ensemble des termes atteignables à partir de* `t`*, en une étape de réécriture. Il suffit par exemple d'itérer ce processus pour parcourir l'espace de*

recherche. Dans cet exemple, le terme `t` *est passé en premier argument de* `Collect` *pour conserver une référence vers le terme initial, dans le contexte duquel la notion de position a un sens. L'objet* `c` *est une collection, qui peut être implantée aussi bien par une* `LinkedList` *que par un* `HashSet`*, permettant ainsi de calculer un multi-ensemble ou un ensemble de résultats.*

La notion de stratégie est un ingrédient essentiel du langage Tom, dont l'utilisation est très simple en pratique. C'est un moyen de séparer clairement les règles de transformation (introduites par `%strategy`) du contrôle, spécifiant de quelle façon les transformations doivent être effectuées. Il existe pour cela une bibliothèque de stratégies prédéfinies (`Repeat`, `Sequence`, `TopDown`, etc.). L'utilisation de stratégies est une vraie originalité par rapport aux langages fonctionnels où la congruence doit être encodée explicitement par appels récursifs, pour chaque constructeur du type inductif considéré. L'utilisation de paramètres dans les stratégies apporte une partie de la puissance des langages fonctionnels à Java. En effet, le passage de stratégie en paramètre est un moyen de simuler le passage de fonction en paramètre. `TopDown` est un exemple de stratégie paramétrée. Si on considère les constructeurs de liste *cons* et *nil*, la stratégie `map`(s) peut se définir par `map`$(s) = \mu x.$`nil <+ cons`(s, x), où `nil` et `cons` sont des stratégies de congruence (c.-à-d. `nil` appliquée à autre chose que *nil* échoue, `cons(s1,s2)` appliquée à $cons(t_1, t_2)$ retourne $cons($`s1`$[t_1],$ `s2`$[t_2])$, et échoue dans les autres cas).

À retenir :

> Partant d'un ensemble d'opérateurs élémentaires (`Sequence`, `Choice`, `All`, `One`, ...), nous avons permis l'utilisation d'un opérateur de récursion μ, et fait en sorte que l'application de chaque stratégie connaisse son contexte, c.-à-d. sa position ω dans le terme global. Ces deux extensions permettent de définir de nombreuses stratégies de parcours, dont *innermost* et *outermost*, ainsi que des stratégies d'exploration dont *breadth-first search* par exemple. Le calcul de ω permet de manipuler explicitement la notion de position, comme on a l'habitude de le faire en écrivant $t_{|\omega}$ ou $t[u]_\omega$ par exemple. La définition d'un ancrage formel nous permet de voir une stratégie comme un terme et par conséquent de pouvoir filtrer vers une stratégie, appliquer une stratégie sur une stratégie, ou encore modifier dynamiquement une stratégie ; c'est ce qui a été utilisé pour transformer une expression de stratégie en un graphe cyclique.

6.4 Point de vue

Les trois contributions de ce chapitre ont pour point commun d'étendre le formalisme de programmation par règles pour le rendre plus expressif et plus facilement utilisable. Elles ont aussi pour point commun d'avoir été initiées par des rencontres, des questions ou des défis.

L'idée d'anti-pattern vient d'une double rencontre. Tout d'abord, dans le cadre d'un projet commun avec Ilog, Hassan Aït-Kaci nous a suggéré de regarder ce que donnerait un formalisme à base de règles et de contraintes. Cela nous a amené à vouloir augmenter l'expressivité du filtrage. Un peu au même moment, Luigi Liquori concevait le langage Snake, qui possédait un symbole de négation dans les motifs. Nous nous sommes lancés le défi de donner une sémantique précise à ce symbole dans le cadre des motifs non-linéaires. Ce fut beaucoup plus difficile que prévu puisqu'il nous a fallu plus d'un an pour réussir à présenter des définitions précises, intuitives et prenant en compte la non-linéarité des variables. Proposer des algorithmes à la fois efficaces et montrés corrects fut particulièrement difficile. Comme beaucoup de nouveautés en programmation, il nous a fallu un peu de temps pour comprendre l'intérêt de ce nouveau type de construction. Combinée avec le filtrage équationnel, la notion d'anti-pattern permet d'exprimer facilement et clairement l'absence d'une famille d'objets, ou l'absence d'éléments identiques dans une liste par exemple.

Lorsque j'ai conçu Tom, la notation variadique m'a semblé être la plus naturelle pour décrire des manipulations de listes. Claude Kirchner m'a alors demandé de lui expliquer quelle était la sémantique précise du filtrage effectué par Tom. J'ai dans un premier temps répondu qu'il s'agissait de filtrage associatif avec élément neutre. Cette réponse ne l'ayant pas convaincu, nous avons entrepris de nous en convaincre. Il nous a fallu, là aussi, plus d'un an pour bien comprendre les liens entre la notation variadique et la notation binaire classique. Ce travail nous a permis de bien comprendre sous quelles conditions le filtrage de Tom correspondait effectivement au filtrage associatif avec élément neutre. Ce travail est évidemment en lien étroit avec l'étude des formes canoniques présentées section 5.3.

En s'inspirant fortement d'Elan, Eelco Visser a présenté un langage à la fois simple, élégant et expressif. En 2001, Joost Visser a présenté dans [Vis01] le lien entre les constructions de Stratego et le *visitor design pattern*. Pendant des années je me suis demandé comment exploiter ce résultat dans le cadre de Tom. Avec Jurgen Vinju, au cours d'un séminaire d'*extreme programming*, nous avons essayé de conjuguer les travaux de Joost Visser avec Tom, sans succès. Ce n'est que quelque temps plus tard, en travaillant sur le générateur de structures de données typées que le puzzle s'est enfin formé. L'intégration semble maintenant évidente et naturelle.

C'est en nous attaquant à la simulation d'un système chimique que nous avons eu besoin de non-déterminisme pour calculer l'ensemble des réactifs produits par la réaction. En introduisant la notion de position explicite, nous avons rendu le langage de stratégie de Tom bien plus expressif que ne le sont les autres langages existants.

Contexte :

Les travaux présentés dans ce document font tous l'objet d'implantations et d'une intégration dans le langage TOM. Celui-ci, et son environnement de programmation associé, forment un ensemble cohérent, fiable, utilisable en pratique. Dans ce chapitre, nous retraçons rapidement l'évolution du système et nous essayons de faire le point sur les applications actuelles, d'identifier les constructions qui mériteraient d'être améliorées, et d'entrevoir les perspectives d'évolution.

7

Terre en Vue

FIABLE **adj.** – de *se fier* **1◇ TECHN.** Se dit d'un matériel dans lequel on peut avoir confiance, qui fonctionne bien.

VITE **adj.** et **adv. I◇ Adj.** Rapide. **II◇ Adv. 1◇** En parcourant un grand espace en peu de temps. **2◇** En peu de temps. **3◇** Au bout d'une courte durée.

7.1 Parcours

C'est avec Marian VITTEK et Christophe RINGEISSEN, au cours de l'année 2000, que nous avons défini les premières fonctionnalités du langage TOM. Le premier prototype de compilateur a été écrit en C et n'était capable que de générer des programmes écrits en C. Le premier programme écrit en TOM est donné dans l'annexe 8.1. En 2001, le compilateur a été étendu pour pouvoir également générer des programmes écrits en Java. Cette amorce a ensuite été utilisée pour réécrire complètement le compilateur en TOM+Java. Ce fut le premier *bootstrap*. Le code source comportait alors moins de 800 lignes.

Au fil du temps, le projet s'est développé, devenant peu à peu un véritable environnement de programmation intégrant compilateur, générateur de structures de données, générateur d'ancrages formels [KM08], générateur de termes de preuve, bibliothèque de *runtime*, plugin `Eclipse` [GMR04], support syntaxique pour `emacs` et `vim`, manuels d'utilisation, tutoriels, exemples, forge de développement, site web, etc.

[KM08] Radu Kopetz and Pierre-Etienne Moreau. Software quality improvement via pattern matching. In José Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, Budapest, Hungary, 2008. Springer-Verlag.

[GMR04] Julien Guyon, Pierre-Etienne Moreau, and Antoine Reilles. An Integrated Development Environment for Pattern Matching Programming. In Brian Barry and Oege de Moor, editors, *Proceedings of the 2nd eclipse Technology eXchange workshop, eTX'2004 (Barcelona, Spain)*, Barcelona (Spain), April 2004. Electronic Notes in Theoretical Computer Science.

Le langage a également évolué, intégrant successivement les notions de stratégies, de formes canoniques, de filtrage $\mathcal{AU}$, d'anti-patterns, etc. L'annexe 8.2 retrace les grandes lignes de cette évolution.

Un plus grand nombre de personnes se sont alors impliquées pratiquement dans le développement et l'utilisation du système, rendant la gestion des développements plus complexe, mais également plus intéressante.

Aujourd'hui, l'environnement est particulièrement stable et robuste. Tom est passé de l'état de prototype à l'état de système fiable et diffusable. Il est utilisé par différentes équipes : centres de recherche, universités, industriels. Les principales applications sont le développement d'outils de preuve et de transformation (optimisation de requêtes ou manipulation de données XML par exemple) ainsi que l'enseignement. Un des succès les plus encourageants est sans doute l'utilisation de Tom faite par Business Objects. Après plusieurs mois d'expérimentation et de comparaison avec les autres outils existants, Tom a été retenu pour écrire et mettre en production un de leurs logiciels. Ce code sera commercialisé à partir de 2010. Depuis cette première mise en œuvre, une deuxième équipe de Business Objects, située à Vancouver, utilise également Tom pour écrire un autre composant du logiciel.

Dans le cadre de nos activités de recherche, Tom est utilisé presque quotidiennement, que ce soit pour prototyper des idées ou pour développer des outils à vocation de diffusion. Cette interaction entre conception, développement et confrontation nous a permis d'identifier un ensemble de constructions pertinent, réduit et utile. Certaines constructions, dont les stratégies, sont directement inspirées de travaux théoriques, tels que ceux menés sur le calcul de réécriture [CK01]. D'autres ont trouvé leur source dans l'étude de problèmes plus pratiques. C'est en essayant de réaliser un outil de vérification pour le protocole de Needham-Schroeder [CMR04], ainsi qu'un prouveur pour le calcul des structures [KMR05a] que nous avons eu besoin de développer un langage de stratégie permettant d'explorer un espace de recherche, ainsi qu'un moyen de calculer efficacement des formes canoniques pour des objets équivalents. C'est en voulant calculer de manière exhaustive les composés produits par des réactions de combustion dans un moteur [AIK06] que nous avons eu l'idée de rendre la position ω explicite.

Le formalisme d'ancrage formel de Tom a été initialement conçu pour « voir » des structures arborescentes comme des termes. Étudiant Mgs [GM01], nous avons essayé d'utiliser les ancrages pour filtrer vers des structures plus complexes, telles que des grilles en 2 dimensions par exemple. Ces expérimentations nous ont amenés à étendre et rendre plus générique le formalisme d'ancrage. C'est également en voulant utiliser Tom comme un *backend* pour Mgs que nous avons été conduits à développer un *backend* Caml pour Tom. Cela nous a amenés à revoir et à améliorer le langage intermédiaire `PIL`.

Récemment, nous avons développé un ancrage formel pour le *bytecode* de Java. Cet ancrage permet de « voir » toute classe Java comme un terme pouvant être transformé par application de règles et de stratégies. Nous avons

implanté un *class loader* qui garantit, par transformation des appels de méthodes, qu'une certaine politique d'accès aux fichiers est respectée [BMR07]. Dans le cadre d'une coopération avec l'équipe LANDE de Rennes et l'équipe CASSIS de Besançon, nous utilisons TOM pour réaliser un outil de complétion d'automate d'arbre. L'objectif étant de réaliser des analyses d'atteignabilité sur des programmes Java exprimés sous forme de systèmes de réécriture.

Parmi les applications majeures développées dans l'équipe à ce jour, citons Lemu qui est un assistant à la preuve permettant de construire des preuves dans le calcul des séquents extensible. Un point clé est le vérificateur de preuve. L'utilisation de TOM, et plus particulièrement du filtrage associatif permet de rendre ce dernier très petit (quelques centaines de lignes), réduisant le risque d'erreurs. Lemu utilise la plupart des constructions offertes par TOM, dont les stratégies, les anti-patterns, les positions explicites ω, ainsi que les derniers développements liés à la réécriture de graphes [BM08].

Une autre application majeure écrite en TOM est certainement le compilateur lui-même. Les sources du logiciel sont organisées en cinq parties :

- l'*amorce*, écrite entièrement en Java, correspond au résultat de la compilation des sources. C'est elle qui permet de compiler le compilateur TOM, écrit essentiellement en TOM,
- les *tests*, écrits en TOM, permettent de s'assurer rapidement (en quelques minutes) que le logiciel fonctionne bien,
- les *exemples*, illustrent divers aspects du langage. Incluant de nombreux tests, ils servent également à vérifier le bon fonctionnement du compilateur,
- les *applications* sont des programmes écrits par des proches (des membres de l'équipe par exemple), qui ne sont pas nécessairement impliqués dans le développement du compilateur,
- les *sources*, correspondent aux compilateurs TOM et GOM, ainsi qu'aux bibliothèques de *runtime*, incluant les *ancrages* et la bibliothèque de stratégies.

Comme illustré table 7.1, le logiciel est relativement « compact » ; ceci est dû en partie à l'expressivité du langage. Le code généré représente quant à lui près de 500 000 lignes (dont 380 000 pour les structures de données) et plus de 3 000 classes.

7.2 Point et perspectives

La conception du langage TOM a été guidée par l'objectif d'en faire un langage expressif, offrant un minimum de concepts, et reposant sur des bases

[BMR07] Emilie Balland, Pierre-Etienne Moreau, and Antoine Reilles. Bytecode rewriting in tom. In *Second Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 07)*, volume 190 of *Electronic Notes in Theoretical Computer Science*, Braga/Portugal, 2007.

	nombre de lignes		nombre de fichiers
amorce (Java)	54 093	29,5%	184
tests	12 471	7%	93
exemples	50 916	28%	299
applications	23 529	13%	82
sources (Tom)	32 552	18%	90
sources (Java)	8 715	5%	73
ancrages	1 034	0,5%	62
total	183 310	100%	883

Table 7.1: Taille du code composant le système

théoriques solides, bien établies. Dans la mesure du possible, les constructions offertes devaient pouvoir se généraliser ; les cas particuliers devant rester exceptionnels. La construction `%match` est un exemple qui illustre cette approche : sa sémantique a su résister aux différentes évolutions du langage et se généraliser au filtrage équationnel, aux anti-patterns, aux stratégies ainsi qu'aux extensions récentes, filtrage de graphes et contraintes de filtrage. L'implantation du système a été guidée par une stratégie visant à réduire autant que possible les coûts de développement, de maintenance, et à maximiser le rapport entre l'effort d'implantation et l'expressivité des constructions offertes dans le langage. Nous avons souvent privilégié des solutions partielles, mais simples, plutôt que des solutions plus complètes, mais complexes. Pour mesurer le gain d'expressivité offert par une construction donnée, nous nous efforçons de l'utiliser pour écrire le compilateur lui-même et nous évaluons son intérêt. À l'image de Louis Majorelle, ébéniste, qui expérimenta ses créations en s'installant dans la « Villa Majorelle », pour laquelle il produisit lui-même une partie de l'équipement, des meubles et des ferronneries. Nous utilisons également la plupart des constructions du langage que nous concevons. Le compilateur est d'une certaine façon notre « Villa Majorelle », nous le mettons à l'épreuve quotidiennement. Il est intéressant de constater que depuis quelques années, le nombre de lignes composant le compilateur est presque constant, alors que le nombre de constructions augmente régulièrement.

Savoir si la stratégie de développement adoptée est bonne est une question que je me suis longtemps posée. S'agissant d'un logiciel destiné à être développé dans un milieu *académique* et à servir de support pour effectuer de la recherche, je suis convaincu que la stratégie adoptée a été essentielle, voire la meilleure.

L'immersion dans un langage hôte comme Java permet de se concen-

trer sur notre métier spécifique (le filtrage, la réécriture, les stratégies) et nous évite d'avoir à consacrer de l'énergie au développement de bibliothèques complexes, telles que la gestion d'entrées-sorties, de *threads*, d'interfaces graphiques, de calculs arithmétiques, etc. L'utilisation de Tom pour écrire le compilateur permet d'expérimenter le langage et les outils. Cela introduit également un style de programmation fonctionnel, sans effet de bords, permettant de décomposer plus facilement le compilateur en phases indépendantes. Ce découpage est essentiel pour aider les différents membres impliqués (principalement des chercheurs) à expérimenter de nouvelles idées, de nouveaux algorithmes, ou à ajouter de nouvelles phases complètement indépendantes, telles que des outils de preuve de terminaison par exemple. L'utilisation de la notion de terme, comme structure de donnée principale, c.-à-d. des arbres sans effet de bord, permet d'intégrer plus facilement de nouveaux développeurs (stagiaires, doctorants) tout en facilitant le déboguage : il suffit de comprendre la structure de l'arbre de syntaxe abstraite (AST) pour pouvoir ajouter de nouveaux traitements. Dans notre domaine, et plus particulièrement dans le cas du compilateur, les AST sont souvent composés de quelques dizaines ou centaines de constructeurs, ce qui rend leur assimilation aisée. J'ai été surpris de constater qu'en plusieurs années de développement, nous n'avons que très rarement cherché un *bug* plus de quelques minutes. Nous n'avons qu'exceptionnellement dépassé l'heure de recherche ; j'ai dû m'y résoudre. En pratique, les problèmes rencontrés sont principalement dûs à des cas oubliés ou des morceaux d'arbres mal construits. L'affichage du terme (c.-à-d. de la mémoire), combiné avec une recherche dans les sources (`grep`) permet dans la plupart des cas de trouver l'origine du problème, qui se résume bien souvent à un membre gauche ou droit de règle mal formé. La réécriture a pour particularité d'utiliser la même syntaxe pour allouer de la mémoire (un membre droit) et pour la représenter (le terme qui est réécrit), ce qui assure une certaine « traçabilité ». Une des conséquences de cette approche est que Tom a pu être développé rapidement, à plusieurs, offrir des constructions puissantes, atteindre un bon niveau de maturité, de fiabilité, tout en continuant à évoluer en fonction des travaux de recherche.

Pour être complet, il faut aussi reconnaître que l'approche suivie a des implications parfois négatives. Le compilateur actuel peut générer du code pour différents langages hôtes. Les premières étapes de *parsing* et de traduction des motifs en automates de filtrage sont entièrement communes. Seule la traduction du langage intermédiaire `PIL` vers le langage cible est spécifique, et correspond à du code distinct dans l'implantation. Cette approche permet de s'adapter rapidement à de nouveaux langages ou aux évolutions syntaxiques d'un langage, comme l'introduction de *generics* dans Java par exemple. En contrepartie, l'analyse statique et la détection d'erreurs est rendue plus difficile, voire impossible. Le langage hôte n'étant pas analysé, une partie des erreurs de typage sont remontées par le compilateur du langage hôte. Pour que les messages, et leurs numéros de ligne associés, aient un

sens, le compilateur Tom doit générer les actions écrites en code hôte à leur emplacement originel (même fichier, même numéro de ligne). Cela ajoute des contraintes sur les algorithmes de compilation utilisés et nous empêche d'effectuer certaines optimisations. Pour modéliser les données, le concept d'ancrage formel est essentiel dans notre démarche. C'est lui qui permet de s'adapter à différentes structures de données, écrites dans différents langages. Dans sa version courante, parce qu'initialement conçu pour pouvoir être utilisé avec le langage C, il ne prend pas en compte la notion de sous-typage. Les modèles de données objets, où la notion d'héritage est présente, ne sont pris en compte que partiellement. Ce qui nous amène à introduire des opérateurs de *coercion* explicites, ou bien à utiliser des types moins précis pour les codomaines des opérateurs. En amont, le générateur Gom ne permet pas non plus de définir des inclusions de sortes, et rend inévitable l'utilisation d'opérateurs de *coercion*. Pour les mêmes raisons, la notion de surcharge n'a pas été initialement prise en compte. Ces différentes limitations, sans être bloquantes, sont à juste titre mal comprises par les utilisateurs.

Nous avons toujours pour objectif de rendre les méthodes formelles plus facilement utilisables et intégrables dans des projets existants. Notre démarche pourra être considérée comme un réel succès le jour où des notions d'arbre, de filtrage, de règle et de stratégie seront intégrées à des langages généralistes comme Java, C#, Python, ou VisualBasic par exemple. Les travaux récents autour de Java 7, C# 3 et F# me donnent bon espoir. Tom est une première étape, une plateforme d'expérimentation, qui permet de montrer à quoi pourraient ressembler ces nouveaux langages. Pour rendre son utilisation plus aisée, nous pensons qu'il faut continuer à améliorer son expressivité et son intégration dans les langages existants. Voici donc une liste de points qui mériteraient d'être étudiés.

Signatures et types plus riches. Le mécanisme de modélisation de données est actuellement restreint aux signatures algébriques, multi-sortées. Pour mieux prendre en compte le modèle de données des langages objets (héritage, surcharge et types génériques), il nous parait important d'intégrer les notions de sous-sortes, de surcharges d'opérateurs, et de sortes paramétrées, directement dans le mécanisme d'ancrage formel. Cela aura des implications sur les algorithmes d'inférence de types et de filtrage. Une des difficultés est la prise en compte des opérateurs associatifs en présence de sous-sortes.

Contraintes de filtrage et extension aux termes associatifs-commutatifs. Pour rendre les membres gauches de règles plus expressifs, nous avons introduit la notion d'anti-terme. Nous avons également introduit la notion de « contrainte de filtrage », qui permet de combiner plusieurs problèmes partageant des variables. Les membres gauches de règles ne sont alors plus de simples motifs, mais des « conditions », c.-à-d. des conjonctions ou des disjonctions de contraintes de filtrage et d'anti-filtrage. Dans le cadre de la thèse de Radu Kopetz, le compilateur a été réécrit pour rendre la définition des

algorithmes de filtrage modulaires, en fonction des théories équationnelles considérées. Il nous semble intéressant d'ajouter, peut-être pas dans toute sa généralité, la notion de contrainte de filtrage pour des termes composés de symboles *associatifs* et *commutatifs*. La restriction à des termes « larges », mais ayant peu de symboles imbriqués, devrait permettre de trouver des algorithmes de filtrage efficaces et de simuler la notion de « règle métier », habituellement implantée par un algorithme de type Rete [For74, For82]. La modularité du compilateur devrait également permettre de combiner filtrage de termes et filtrage sur des bases importantes d'objets (c.-à-d. la *working memory*).

Analyse de programmes. Un domaine d'application privilégié de Tom est l'analyse statique de programmes. Nous avons mis en place des outils permettant de définir facilement des arbres de syntaxe abstraite (Gom) et de les construire directement à partir d'outils standard d'analyse syntaxique, tels que Antlr [Par07]. Les opérations d'analyse et de transformation de programmes peuvent alors se décrire en utilisant des constructions de filtrage et des stratégies. Cependant, il est fréquent que des informations représentant la dynamique du programme soient nécessaires. En particulier, le graphe de flot de contrôle par exemple. C'est pourquoi nous avons commencé à étudier comment représenter des graphes, comment les parcourir et les transformer tout en restant dans l'environnement Tom. Une solution originale et prometteuse consiste à représenter les graphes par des termes contenant des constantes particulières : des *pointeurs* vers un autre sous-arbre [BM08]. Ces travaux sont décrits dans les annexes de ce manuscrit.

Un de nos objectifs est de créer des outils d'analyse de programmes plus puissants, capables de prendre en compte des combinaisons de langages, comme c'est souvent le cas dans les applications Web mélangeant JavaScript, XML et HTML par exemple. Nous pensons qu'un axe de recherche intéressant serait l'étude d'algorithmes de filtrage s'appliquant sur des structures de données compactes telles que les BDD (*Binary Decision Diagram*). En effet, une approche prometteuse consiste à analyser les programmes en collectant un ensemble d'informations correspondant à des relations entre entités [LBN05, Bey06] (variables, fonctions, instructions, modules, etc.). Pour être capable de les traiter efficacement, ces informations sont représentées par des BDD. Combiner de telles structures de données avec le filtrage, la réécriture et les stratégies devrait permettre d'analyser plus facilement les programmes où l'étude de l'arbre abstrait et du flot de contrôle ne suffit pas.

Meilleure intégration. L'approche îlot formel permet de développer plus rapidement de nouveaux langages et de s'adapter aux évolutions de ceux-ci.

[BM08] Emilie Balland and Pierre-Etienne Moreau. Term-graph rewriting via explicit paths. In *Proceedings of the 19th Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 2008. Accepté pour publication.

En revanche, le langage ainsi construit peut souffrir de quelques inconvénients par rapport aux langages classiques : une syntaxe moins uniforme, des messages d'erreurs moins précis, des environnements de développement moins complets. Par exemple, dans le cadre de Tom et Java, certaines erreurs de typage ne sont détectées que par le compilateur Java. Dans l'environnement `Eclipse`, la complétion de noms de fonctions et le *refactoring* ne sont plus possibles. Pour faire de Tom un langage destiné à être utilisé dans un milieu industriel, nous pensons qu'il est important de corriger ces défauts. Pour cela, nous étudions deux approches possibles. La première consiste à développer des versions spécialisées pour quelques langages, dont Java, incluant un *parseur* dédié et un outil d'analyse sémantique. Cette approche est une conséquence naturelle de notre démarche : après une expérimentation intéressante et prometteuse, nous voulons prendre en compte les attentes et les besoins des utilisateurs. Les difficultés liées à l'analyse sémantique des langages hôtes s'intègrant dans un projet plus large visant à développer un ensemble de composants permettant de construire des outils d'analyse de programmes. La deuxième solution envisagée consiste à pousser à l'extrême l'idée d'îlot formel, en intégrant complètement ces îlots, via une API, dans la syntaxe du langage hôte. La syntaxe pourrait ressembler à `if(match("f(x) << s")) { make("g(x)"); }`, où `match` et `make` seraient des méthodes de l'interface Tom, compilées en utilisant une approche par *aspects* par exemple. Une autre possibilité que nous étudions est la réalisation d'une *fluent interface* [FP06]. Dans les deux cas, la difficulté consiste à concevoir une interface facile à utiliser.

C'est ici que s'achève ce premier périple. Huit années de recherche ont été présentées, la terre est en vue.

8

Annexes

8.1 Premier programme Tom

Conçu pour le langage C au départ, sa syntaxe est différente de celle actuelle, on reconnaît cependant bien l'idée d'ancrage formel (introduite par %GET_FUN_SYM, %GET_SUBTERM et %sym) qui est présente depuis le début.

```
%// This is an example on computing n-th fib number on
%// peano arithmetics
%{
#define ZERO 0
#define SUC 1
#define PLUS 2
#define FIB 3

struct term {
  int fs;
  int arity;
  struct term **subt;
};
%}

%GET_FUN_SYM<struct term *>(x) (x->fs)
%GET_SUBTERM<struct term *>(x,n) (x->subt[n])

%sym struct term *zero % ZERO
%sym struct term *suc(struct term *) % SUC
%sym struct term *plus(struct term *, struct term *) % PLUS
%sym struct term *fib(struct term *) % FIB

%var struct term *x, *y, *z;
```

```
%rule plus(x, zero) %--> return(x);
%rule plus(x, suc(y)) %--> return(suc(plus(x,y)));

%rule fib(zero) %--> return(suc(zero));
%rule fib(suc(zero)) %--> return(suc(zero));
%rule fib(suc(suc(x))) %--> return(plus(fib(x),fib(suc(x))));
%%
struct term *suc(struct term *x) {
       struct term *res;
       res = malloc(sizeof(struct term));
       res->fs = SUC;
       res->arity = 1;
       res->subt = malloc(sizeof(struct term *));
       res->subt[0] = x;
       return(res);
}

static void symbolicFib(int n) {
  int i;
  struct term *t;
  t = zero;
  for(i=0; i<n; i++) t = suc(t);
}

int main() {
  symbolicFib(10);
}
```

8.2 Extrait du ChangeLog

```
2001-01-25  Pierre-Etienne Moreau  <moreau@loria.fr>
  * Setup of Tom under automake
2001-02-14  Pierre-Etienne Moreau  <moreau@loria.fr>
  * src/jtom/Tom.java: first bootstrap
2001-02-20  Pierre-Etienne Moreau  <moreau@loria.fr>
  * test/Jfib1Test.java: JUnit is now used
2001-03-02  Pierre-Etienne Moreau  <moreau@loria.fr>
  * new syntax for type and symbol declarations
    %typeterm  (implement, get_fun_sym(t), get_subterm(t,n), ...)
    %typelist  (implement, get_element(l,n), get_size(l))
    %typearray (implement, get_head(l), get_tail(l), is_empty(l))
    %op        (fsym, make(t1,...,tn))
    %oplist    (fsym, make_empty(), make_add(l,e))
    %oparray   (fsym, make_empty(size), make_add(l,e,index))
```

```
2001-03-20 * parse and check list-pattern
2001-03-26 * compilation of list-matching (typearray representation)
...
2001-08-24 * first Eiffel program automatically compiled
...
2002-10-27 * new generic methods: map, maptree and traversal
2002-11-19 * version 0.7: bootstrap with ApiGen (6 types)
2002-11-21 * version 1.0beta, new stable version based on ApiGen
2003-01-23 * release 1.1 under GPL license
2003-04-23 * release 1.2 under GPL license
2003-08-22 * release 1.3
2003-09-23 * release 1.4
2003-12-10 * version 1.5 released
2004-06-09 * version 2.0 released
2004-12-20 * version 2.1 released
2005-07-29 * version 2.2 released
2006-04-28 * version 2.3 released
2006-10-04 * version 2.4 released
2007-07-09 * version 2.5 released
2008-04-xx * version 2.6 released
...
```

Bibliographie

[AG93] Andrew W. Appel et Marcelo J. R. Gonçalves : Hash-consing garbage collection. Rapport technique CS-TR-412-93, Princeton, février 1993.

[AIK06] Oana Andrei, Liliana Ibanescu et Hélène Kirchner : Non-intrusive formal methods and strategic rewriting for a chemical application. *In* Kokichi Futatsugi, Jean-Pierre Jouannaud et José Meseguer, éditeurs : *Essays Dedicated to Joseph A. Goguen*, volume 4060 de *Lecture Notes in Computer Science*, pages 194–215. Springer, 2006.

[AK90] Hassan Aït-Kaci : The WAM : a (real) tutorial. Technical report 5, Digital Systems Research Center, Paris (France), janvier 1990.

[ASVO05] Tiago Alves, Paulo Silva, Joost Visser et José Nuno Oliveira : Strategic Term Rewriting And Its Application To A VDM-SL to SQL Conversion. *In Formal Methods 2005*, volume 3582 de *LNCS*, pages 399–414, 2005.

[Aug85] Lennart Augustsson : Compiling pattern matching. *In Proceedings of the conference on Functional Programming Languages and Computer Architecture*, pages 368–381. Springer-Verlag, 1985.

[BDL06] Sandrine Blazy, Zaynah Dargaye et Xavier Leroy : Formal verification of a C compiler front-end. *In Formal Methods 2006*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[Bey06] Dirk Beyer : Relational programming with crocopat. *In ICSE '06 : Proceedings of the 28th International Conference on Software Engineering*, pages 807–810, New York, NY, USA, 2006. ACM.

[BHW07] Frédéric Blanqui, Thérèse Hardin et Pierre Weis : On the implementation of construction functions for non-free concrete data types. *In* Rocco De Nicola, éditeur : *16th European Symposium on Programming*, volume 4421 de *Lecture Notes in Computer Science*, pages 95–109. Springer, 2007.

[BJMR98] Peter Borovanský, Salma Jamoussi, Pierre-Etienne Moreau et Christophe Ringeissen : Handling ELAN rewrite programs via an exchange format. *In* Claude Kirchner

et Hélène KIRCHNER, éditeurs : *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications, WRLA'98 (Pont-à-Mousson, France)*, volume 15, Pont-à-Mousson (France), septembre 1998. Electronic Notes in Theoretical Computer Science.

[BK98] Jan A. BERGSTRA et Paul KLINT : The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.

[BKM06] Emilie BALLAND, Claude KIRCHNER et Pierre-Etienne MOREAU : Formal Islands. *In* Michael JOHNSON et Varmo VENE, éditeurs : *11th International Conference on Algebraic Methodology and Software Technology*, volume 4019 de *LNCS*, pages 51–65, Kuressaare, Estonia, juillet 2006. Springer-Verlag.

[BKN87] Dan BENANAV, Deepak KAPUR et Paliath NARENDRAN : Complexity of matching problems. *Journal of Symbolic Computation*, 3(1-2):203–216, 1987.

[BL73] Daniel BELL et Len LAPADULA : Secure Computer Systems : a Mathematical Model. Rapport technique MTR-2547 (Vol. II), MITRE Corp., Bedford, MA, May 1973.

[BM05] Emilie BALLAND et Pierre-Etienne MOREAU : Optimizing pattern matching by program transformation. Rapport technique, INRIA-LORIA, 2005. `http://hal.inria.fr/inria-00000763`.

[BM06] Emilie BALLAND et Pierre-Etienne MOREAU : Optimizing pattern matching compilation by program transformation. *In* Jean-Marie FAVRE, Reiko HECKEL et Tom MENS, éditeurs : *3rd Workshop on Software Evolution through Transformations (SeTra'06)*. Electronic Communications of EASST, 2006. ISSN 1863-2122.

[BM08] Emilie BALLAND et Pierre-Etienne MOREAU : Term-graph rewriting via explicit paths. *In Proceedings of the 19th Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 2008. Accepté pour publication.

[BMR07] Emilie BALLAND, Pierre-Etienne MOREAU et Antoine REILLES : Bytecode rewriting in tom. *In Second Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 07)*, volume 190 de *Electronic Notes in Theoretical Computer Science*, Braga/Portugal, 2007.

[Bor98] Peter BOROVANSKÝ : *Le contrôle de la réécriture : étude et implantation d'un formalisme de stratégies.* Thèse de Docto-

rat d'Université, Université Henri Poincaré - Nancy I, octobre 1998.

[BW88] Hans-Juergen Boehm et Mark Weiser : Garbage collection in an uncooperative environment. *Software : Practice and Experience*, 18(9):807–820, 1988.

[BY92] Robert S. Boyer et Yuan Yu : Automated correctness proofs of machine code programs for a commercial microprocessor. *In* D. Kapur, éditeur : *Proceedings of the 11th International Conference on Automated Deduction*, pages 416–430. Springer-Verlag, 1992.

[Car84] Luca Cardelli : Compiling a functional language. *In Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 208–217, 1984.

[CC77] Patrick Cousot et Radhia Cousot : Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In Conference Records of the 4th ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, Los Angeles, CA, USA, janvier 1977.

[CK01] Horatiu Cirstea et Claude Kirchner : The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, mai 2001.

[CL90] Hubert Comon et Pierre Lescanne : Equational problems and disunification. *In* Claude Kirchner, éditeur : *Unification*, pages 297–352. Academic Press inc., London, 1990.

[Cla98] Manuel Clavel : *Reflection in general logics, rewriting logic, and Maude*. Thèse de doctorat, University of Navarre, Spain, 1998.

[CMR04] Horatiu Cirstea, Pierre-Etienne Moreau et Antoine Reilles : Rule Based Programming in Java for Protocol Verification. *In* Narcisso Marti-Oliet, éditeur : *Proceedings of the 5th International Workshop on Rewriting Logic and its Applications, WRLA'2004 (Barcelona, Spain)*, volume 117, pages 209–227, Barcelona (Spain), avril 2004. Electronic Notes in Theoretical Computer Science.

[Con04] Evelyne Contejean : A certified ac matching algorithm. *In* Vincent van Oostrom, éditeur : *RTA*, volume 3091 de *Lecture Notes in Computer Science*, pages 70–84. Springer, 2004.

[dJO04] Hayco de Jong et Pieter Olivier : Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59(1-2):35–61, 2004.

[Dol] Damien Doligez : Zenon : an automatic theorem prover for first-order logic. Available as part of the Focal system at `http://focal.inria.fr/zenon/`.

[dRE98] Willem-Paul de Roever et Kai Engelhardt : *Data Refinement : Model-Oriented Proof Methods and their Comparison*. Numéro 47 de Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, novembre 1998.

[DWH+90] Alan Demers, Mark Weiser, Barry Hayes, Daniel G. Bobrow et Scott Shenker : Combining generational and conservative garbage collection : Framework and implementations. *In Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery's Special Interest Group on programming languages, pages 261–269, San Francisco, CA, janvier 1990. ACM.

[EOW07] Burak Emir, Martin Odersky et John Williams : Matching objects with patterns. *In* Erik Ernst, éditeur : *ECOOP*, volume 4609 de *Lecture Notes in Computer Science*, pages 273–298. Springer, 2007.

[FM01] Fabrice Le Fessant et Luc Maranget : Optimizing pattern matching. *In Proceedings of the sixth International Conference on Functional Programming*, pages 26–37. ACM Press, 2001.

[For74] Charles Forgy : A network fast routine for production systems. Working paper, Carnegie-Mellon University, 1974.

[For82] Charles Forgy : Rete : A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[FP06] Steve Freeman et Nat Pryce : Evolving an embedded domain-specific language in java. *In OOPSLA '06 : Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 855–865, New York, NY, USA, 2006. ACM.

[GKK+87] Joseph A. Goguen, Claude Kirchner, Hélène Kirchner, Aristide Mégrelis, Jose Meseguer et Timothy Winkler : An introduction to OBJ-3. *In* J.-P. Jouannaud et S. Kaplan, éditeurs : *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 de *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, juillet 1987. Also as internal report CRIN : 88-R-001.

[GM01] Jean-Louis Giavitto et Olivier Michel : MGS : a Rule-Based Programming Language for Complex Objects and Collections. *In* Mark van den Brand et Rakesh Verma, éditeurs : *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.

[GMR04] Julien Guyon, Pierre-Etienne Moreau et Antoine Reilles : An Integrated Development Environment for Pattern Matching Programming. *In* Brian Barry et Oege de Moor, éditeurs : *Proceedings of the 2nd eclipse Technology eXchange workshop, eTX'2004 (Barcelona, Spain)*, Barcelona (Spain), avril 2004. Electronic Notes in Theoretical Computer Science.

[GN04] Sumit Gulwani et George C. Necula : Global value numbering using random interpretation. *In Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'04)*, pages 342–352. ACM, 2004.

[Gog78] Joseph A. Goguen : Some design principles and theory for OBJ-0, a language for expressing and executing algebraic specifications of programs. *In* E. Blum, M. Paul et S. Takasu, éditeurs : *Proceedings of Mathematical Studies of Information Processing*, volume 75. Lecture Notes in Computer Science, 1978.

[Grä91] Albert Gräf : Left-to-right tree pattern matching. *In* R. V. Book, éditeur : *Proceedings 4th Conference on Rewriting Techniques and Applications, Como (Italy)*, volume 488 de *Lecture Notes in Computer Science*, pages 323–334. Springer-Verlag, avril 1991.

[Gug02] Alessio Guglielmi : A system of interaction and structure. Rapport technique WV-02-10, TU Dresden, 2002. To appear in ACM Transactions on Computational Logic.

[Han99] David R. Hanson : Early Experience with ASDL in lcc. *Software - Practice and Experience*, 29(3):417–435, 1999.

[HM01] Gorel Hedin et Eva Magnusson : Jastadd—a java-based system for implementing front ends. *In* D. Parigot et M.G.J. van den Brand, éditeurs : *Proceedings of the 1st International Workshop on Language Descriptions, Tools and Applications*, volume 44, Genova (Italy), april 2001. Electronic Notes in Theoretical Computer Science.

[HO82] Christoph M. Hoffmann et Michael J. O'Donnell : Programming with equations. *ACM Transactions on Programming Languages and Systems*, 4(1):83–112, 1982.

[JM92] Jean-Pierre Jouannaud et Claude Marché : Termination and completion modulo associativity, commutativity and

identity. *Theoretical issues of Design and Implementation of Symbolic Computation Systems*, 104(1):29–51, 1992.

[JO04] Hayco A. de JONG et Pieter A. OLIVIER : Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59, April 2004.

[Jon96] Richard E. JONES : *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, juillet 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[Kah87] Gilles KAHN : Natural semantics. *In 4th Annual Symposium on Theoretical Aspects of Computer Sciences (STACS'87)*, pages 22–39, London, UK, 1987. Springer-Verlag.

[KB70] Donald E. KNUTH et Peter B. BENDIX : Simple word problems in universal algebras. *In* J. LEECH, éditeur : *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

[KKM07] Claude KIRCHNER, Radu KOPETZ et Pierre-Etienne MOREAU : Anti-pattern matching. *In* Rocco De NICOLA, éditeur : *16th European Symposium on Programming*, volume 4421 de *Lecture Notes in Computer Science*, pages 110–124, Braga, Portugal, 2007. Springer.

[KKM08] Claude KIRCHNER, Radu KOPETZ et Pierre-Etienne MOREAU : Anti-pattern matching modulo. *In* Carlos MARTÍN-VIDE, éditeur : *Proceedings of the 2nd International Conference on Language and Automata Theory and Applications*, Tarragona, Spain, 2008.

[Kli93] Paul KLINT : A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[KM95] Hélène KIRCHNER et Pierre-Etienne MOREAU : Prototyping completion with constraints using computational systems. *In* J. HSIANG, éditeur : *Proceedings 6th Conference on Rewriting Techniques and Applications*, volume 914 de *Lecture Notes in Computer Science*, pages 438–443, Kaiserslautern, Germany, 1995. Springer-Verlag.

[KM96] Hélène KIRCHNER et Pierre-Etienne MOREAU : A reflective extension of Elan. *In* José MESEGUER, éditeur : *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), septembre 1996. Electronic Notes in Theoretical Computer Science.

[KM01] Hélène KIRCHNER et Pierre-Etienne MOREAU : Promoting rewriting to a programming language : A compiler for non-

deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(2):207–251, 2001.

[KM08] Radu KOPETZ et Pierre-Etienne MOREAU : Software quality improvement via pattern matching. *In* José FIADEIRO et Paola INVERARDI, éditeurs : *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, Budapest, Hungary, 2008. Springer-Verlag.

[KMR05a] Ozan KAHRAMANOĞULLARI, Pierre-Etienne MOREAU et Antoine REILLES : Implementing Deep Inference in Tom. *In* Paola BRUSCOLI, François LAMARCHE et Charles STEWART, éditeurs : *Structures and Deduction – the Quest for the Essence of Proofs (satellite workshop of ICALP 2005), Technical Report ISSN 1430-211X, Technische Universität Dresden*, pages 158–172, Lisbon, Portugal, 2005.

[KMR05b] Claude KIRCHNER, Pierre-Etienne MOREAU et Antoine REILLES : Formal Validation of Pattern Matching Code. *In* Pedro BARAHONE et Amy FELTY, éditeurs : *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 187–197. ACM, juillet 2005.

[KN86] Deepak KAPUR et Paliath NARENDRAN : NP-completeness of the set unification and matching problems. *In Proc. of the 8th international conference on Automated deduction*, pages 489–495, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[KV01] Tobias KUIPERS et Joost VISSER : Object-oriented tree traversal with JJForester. *In* Mark G. J. van den BRAND et Didier PARIGOT, éditeurs : *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001. Proc. of Workshop on Language Descriptions, Tools and Applications (LDTA).

[LBN05] Claus LEWERENTZ, Dirk BEYER et Andreas NOACK : Efficient relational calculation for software analysis. *IEEE Trans. Softw. Eng.*, 31(2):137–149, 2005.

[LDG+04] Xavier LEROY, Damien DOLIGEZ, Jacques GUARRIGUE, Didier RÉMY et Jérôme VOUILLON : The Objective Caml system, 2004. `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[Ler06] Xavier LEROY : Formal certification of a compiler back-end, or : programming a compiler with a proof assistant. *In 33rd*

symposium Principles of Programming Languages, pages 42–54. ACM, 2006.

[LJVWF02] David Lacey, Neil D. Jones, Eric Van Wyk et Carl Christian Frederiksen : Proving correctness of compiler optimizations by temporal logic. *In Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 283–294. ACM, 2002.

[LM03] Jed Liu et Andrew C. Myers : Jmatch : Iterable abstract pattern matching for java. *In* Verónica Dahl et Philip Wadler, éditeurs : *PADL*, volume 2562 de *Lecture Notes in Computer Science*, pages 110–127. Springer, 2003.

[Mar96] Claude Marché : Normalized rewriting : an alternative to rewriting modulo a set of equations. *Journal of Symbolic Computation*, 21(3):253–288, 1996.

[McL88] John McLean : The algebra of security. *In Proc. IEEE Symposium on Security and Privacy*, pages 2–7. IEEE Computer Society Press, 1988.

[MHS03] Marjan Mernik, Jan Heering et Anthony M. Sloane : When and how to develop domain-specific languages. Rapport technique, CWI, 2003.

[MOMV04] Narciso Martí-Oliet, José Meseguer et Alberto Verdejo : Towards a strategy language for maude. *In* Narcisso Marti-Oliet, éditeur : *Proceedings of the 5th International Workshop on Rewriting Logic and its Applications, WRLA'2004 (Barcelona, Spain)*, volume 117, Barcelona (Spain), avril 2004. Electronic Notes in Theoretical Computer Science.

[Mor73] F. Lockwood Morris : Advice on structuring compilers and proving them correct. *In Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 144–152. ACM, 1973.

[Mor00] Pierre-Etienne Moreau : REM (Reduce Elan Machine) : Core of the new ELAN compiler. *In Proceedings 11th Conference on Rewriting Techniques and Applications*, volume 1833 de *Lecture Notes in Computer Science*, pages 265–269, Norwich, UK, 2000. Springer-Verlag.

[MP67] John McCarthy et James Painter : Correctness of a compiler for arithmetic expressions. *In* J. T. Schwartz, éditeur : *Proceedings Symposium in Applied Mathematics, Vol. 19*, pages 33–41. AMS, 1967.

[MRV01] Pierre-Etienne Moreau, Christophe Ringeissen et Marian Vittek : A Pattern-Matching Compiler. *In* Didier Parigot

et Mark G. J. van den BRAND, éditeurs : *Proceedings of the 1st International Workshop on Language Descriptions, Tools and Applications*, volume 44, Genova (Italy), avril 2001. Electronic Notes in Theoretical Computer Science.

[MRV03] Pierre-Etienne MOREAU, Christophe RINGEISSEN et Marian VITTEK : A Pattern Matching Compiler for Multiple Target Languages. *In* G. HEDIN, éditeur : *12th Conference on Compiler Construction*, volume 2622 de *LNCS*, pages 61–76, Warsaw, Poland, mai 2003. Springer-Verlag.

[MZ04] Pierre-Etienne MOREAU et Olivier ZENDRA : GC^2 : A Generational Conservative Garbage Collector for the ATerm Library. *Journal of Logic and Algebraic Programming*, 59(1–2):5–34, avril 2004.

[Nec97] George C. NECULA : Proof-carrying code. *In Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM, 1997.

[NL98] George C. NECULA et Peter LEE : The design and implementation of a certifying compiler. *SIGPLAN Not.*, 33(5):333–344, 1998.

[ORW95] Dino P. OLIVA, John D. RAMSDELL et Mitchell WAND : The VLISP verified PreScheme compiler. *Lisp Symb. Comput.*, 8(1-2):111–182, 1995.

[OW97] Martin ODERSKY et Philip WADLER : Pizza into Java : Translating theory into practice. *In Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[Par07] Terence PARR : *The Definitive ANTLR Reference : Building Domain-Specific Languages*. The Pragmatic Bookshelf, Raleigh, 2007.

[PSS98] Amir PNUELI, Michael SIEGEL et Eli SINGERMAN : Translation validation. *In Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166. Springer-Verlag, 1998.

[PTW] Jens PALSBERG, Kevin TAO et Wanjun WANG : Java tree builder. Available at `http ://www.cs.purdue.edu/jtb`. Purdue University, Indiana, U.S.A.

[Rei06] Antoine REILLES : Canonical abstract syntax trees. *In* Grit DENKER et Carolyn TALCOTT, éditeurs : *Proceedings of WRLA 2006*, volume 176, pages 165–179, Vienna, Austria, 2006. Electronic Notes in Theoretical Computer Science.

[Riv04] Xavier Rival : Symbolic transfer function-based approaches to certified compilation. *In Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'04)*, pages 1–13. ACM, 2004.

[RL07] Adam Richard et Ondrej Lhoták : Oomatch : pattern matching as dispatch in java. *In* Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes et Guy L. Steele Jr., éditeurs : *OOPSLA Companion*, pages 771–772. ACM, 2007.

[RM99] Martin C. Rinard et Darko Marinov : Credible compilation with pointers. *In Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, juillet 1999.

[Spi01] Diomidis Spinellis : Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1): 91–99, 2001.

[SRR95] R. C. Sekar, R. Ramesh et I. V. Ramakrishnan : Adaptive pattern matching. *SIAM Journal on Computing*, 24(6):1207–1234, 1995.

[Str02] Martin Strecker : Formal verification of a java compiler in Isabelle. *In Proceedings of the 18th International Conference on Automated Deduction*, pages 63–77. Springer-Verlag, 2002.

[VB98] Eelco Visser et Zine-el-Abidine Benaissa : A core language for rewriting. *In* Claude Kirchner et Hélène Kirchner, éditeurs : *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications, WRLA'98*, volume 15, Pont-à-Mousson (France), septembre 1998. Electronic Notes in Theoretical Computer Science.

[VBT98] Eelco Visser, Zine-el-Abidine Benaissa et Andrew Tolmach : Building program optimizers with rewriting strategies. *In Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM, septembre 1998.

[vdBdJKO00] Mark G. J. van den Brand, Hayco A. de Jong, Paul Klint et Pieter Olivier : Efficient annotated terms. *Software-Practice and Experience*, 30:259–291, 2000.

[vdBMR02] Mark. G. J. van den Brand, Pierre-Etienne Moreau et Christophe Ringeissen : The ELAN environment : a rewriting logic environment based on ASF+SDF technology. *In* Mark G. J van den Brand et Ralf Lämmel, éditeurs : *Proceedings of the 2st International Workshop on Language Descriptions, Tools and Applications*, volume 65, Grenoble (France), avril 2002. Electronic Notes in Theoretical Computer Science.

[vdBMV03] Mark. G. J. van den BRAND, Pierre-Etienne MOREAU et Jurgen VINJU : Environments for Term Rewriting Engines for Free ! *In* R. NIEUWENHUIS, éditeur : *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, volume 2706 de *Lecture Notes in Computer Science*, pages 424–435, Valencia (Spain), june 2003. Springer-Verlag.

[vdBMV05] Mark. G. J. van den BRAND, Pierre-Etienne MOREAU et Jurgen VINJU : Generator of Efficient Strongly Typed Abstract Syntax Trees in Java. *IEE Proceedings - Software Engineering*, 152(2):70–78, avril 2005.

[Vis01] Joost VISSER : Visitor combination and traversal control. *In Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'01)*, pages 270–282, New York, NY, USA, 2001. ACM.

[Vit94] Marian VITTEK : ELAN *: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, octobre 1994.

[Wad87] Philip WADLER : Views : a way for pattern matching to cohabit with data abstraction. *In Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 307–313. ACM, 1987.

[WAKS97] Daniel C. WANG, Andrew W. APPEL, Jeff L. KORN et Christopher S. SERRA : The Zephyr Abstract Syntax Description Language. *In Proceedings of the Conference on Domain-Specific Languages*, pages 213–227, 1997.

[Wan95] Mitchell WAND : Compiler correctness for parallel languages. *In Proceedings of the seventh international conference on Functional programming languages and computer architecture (FPCA'95)*, pages 120–134, New York, NY, USA, 1995. ACM.

[War83] David H. D. WARREN : An abstract Prolog instruction set. Rapport technique 309, SRI International, Artificial Intelligence Center, 1983.

[WAS03] Dinghao WU, Andrew W. APPEL et Aaron STUMP : Foundational proof checkers with small witnesses. *In Proceedings of the 5th ACM SIGPLAN international Conference on Principles and Practice of Declarative Programming*, pages 264–274. ACM, 2003.

[Wil92] Paul R. WILSON : Uniprocessor garbage collection techniques. *In* Yves BEKKERS et Jacques COHEN, éditeurs : *Proceedings of International Workshop on Memory Management*, volume 637

de *Lecture Notes in Computer Science*, University of Texas, USA, 16–18 septembre 1992. Springer-Verlag.

[Wil94] Paul R. Wilson : Uniprocessor garbage collection techniques. Rapport technique, University of Texas, janvier 1994. Expanded version of the IWMM92 paper.

[Woh07] Claes Wohlin : An analysis of the most cited articles in software engineering journals - 2000. *Inf. Softw. Technol.*, 49(1):2–11, 2007.

[Zor89] Benjamin G. Zorn : *Comparative Performance Evaluation of Garbage Collection Algorithms*. Thèse de doctorat, University of California at Berkeley, mars 1989. Technical Report UCB/CSD 89/544.