Réécriture et compilation de confiance

Antoine Reilles

Institut National
Polytechnique de Lorraine

Département de formation doctorale en informatique

École doctorale IAEM Lorraine

Réécriture et compilation de confiance

THÈSE

présentée et soutenue publiquement le 27 novembre 2006

pour l'obtention du

Doctorat de l'Institut National Polytechnique de Lorraine

(spécialité informatique)

par

Antoine Reilles

Composition du jury

Rapporteurs :	Pierre Cointe	Professeur, École des Mines de Nantes, France
	Paul Klint	Professeur, Universiteit van Amsterdam, Pays-Bas
Examinateurs :	Claude Kirchner	Directeur de recherche, INRIA, Nancy, France
	Pierre-Étienne Moreau	Chargé de recherche, INRIA, Nancy, France
	Karl Tombre	Professeur, École des Mines de Nancy, INPL, France
	Reinhard Wilhelm	Professeur, Universität des Saarlandes, Allemagne

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

Mis en page avec LaTeX

Table des matières

Remerciements

Je tiens à remercier tout d'abord Guillaume Bonfante et Karl Tombre, qui ont su me montrer que l'informatique pouvait être une science, me communiquer l'envie d'en explorer une parcelle, et de me lancer dans l'aventure de la recherche.

Claude Kirchner et Pierre-Étienne Moreau m'ont encadré pendant ces trois ans de thèse ainsi que durant mon DÉA. Leur patience et leur enthousiasme communicatif n'ont eu d'égal que leur disponibilité. Pierre-Étienne a toujours su trouver les cinq minutes nécessaires pour examiner un problème, qui se terminent quelques heures plus tard.

Je tiens à remercier Damien Doligez pour la patience dont il a fait preuve lorsqu'il tentait de m'expliquer le fonctionnement de Zenon, ainsi que pour tout le temps qu'il a passé à m'expliquer l'utilisation de Coq, ainsi qu'a prouver les théorèmes que je lui proposais.

Je voudrais remercier les personnes qui ont accepté d'être membres de mon jury.

Paul Klint à accepté la tâche d'être rapporteur. Je tiens aussi à le remercier pour ses commentaires et conseils très pertinents, qui m'ont permis d'améliorer ce document, et permettront encore d'améliorer ce travail. Avec Mark van den Brand et l'équipe d'ASF+SDF ils avaient eu l'extrême gentillesse de m'accueillir au CWI pour mon premier mois de thèse, je profite de l'occasion pour les en remercier.

Merci à Pierre Cointe d'avoir bien voulu accepter de rapporter ce document, ainsi que pour tout l'intérêt qu'il a manifesté pour ce travail.

Reinhard Wilhelm et Karl Tombre ont accepté de faire partie de ce jury et de jouer le rôle d'examinateurs. Je leur en suis très reconnaissant.

Je tiens à remercier les auteurs des systèmes TEX, LATEX et METAPOST. Sans ces outils ce document aurait sûrement tout de même vu le jour, mais en me permettant de conserver l'impression de programmer, ils ont permis à ce processus de ne pas être trop douloureux. Merci aussi aux développeurs des projets NetBSD et Kaffe, en particulier Thomas, Quentin et Dalibor pour m'avoir fourni de quoi occuper quelques nuits et dimanches après-midi.

Je ne remercierai jamais assez tous les collègues et amis, Emmanuel, Florent, Germain, Stéphane et Stéphane, pour m'avoir tour à tour poussé à travailler, dévier sur les problèmes informatiques les plus incongrus, examiner la marche du monde ou échafauder les théories les plus absurdes.

Finalement, je remercie profondément ma famille ainsi que Gaëlle pour tout le soutien qu'ils ont pu m'apporter durant ces trois années (et même avant).

Extended Abstract

The subject of this thesis is the compilation of high level languages to lower level ones. More precisely, the focus is on how much we can trust those compilers, and what can be done effectively to increase the level of confidence we can have in those compilers.

This question is motivated by the development of Tom, which is a compiler for matching constructs in languages like Java, C or ML. *The Tom language provides tools and features to help writing safe compilers, by introducing high level constructs and signatures helping to describe formally complex algorithms and abstract syntax tree manipulations.*

A first problem is how much confidence can be placed in the Tom compiler itself: if this compiler produces erroneous code, how could it help writing better compilers? We address this issue by the development of a certification method allowing us to prove at each run of the compiler that the generated code effectively implements the matching constructs to compile.

This leads us to provide a generator of efficient, maximally shared and safe data structure implementations, integrating invariants and their preservation. The invariants, such as requiring lists to be ordered, or neutral elements to be removed are enforced by the implementation. This data structure generator is also a keystone to build safe applications with Tom.

We then focus on providing a higher level language for describing tree traversals and transformations, leading to a simpler and safer implementation of complex document transformations.

Once those pieces are gathered together, we obtain a software environment to describe and implement compilers in a trustworthy way. This environment is not intended to provide fully formally verified compilers and document transformations, but rather provide an effective framework to ease development while raising the confidence level in the result.

The Tom environment

Rewriting and pattern-matching are of general use in mathematics and informatics to describe computation as well as deduction. They are central in systems making the notion of rule an explicit and first class object, like expert and business systems (JRule), programming languages based on equational logic (OBJ), the rewriting calculus (ELAN) or logic (MAUDE), functional, possibly logic, programming (ML, Haskell, ASF+SDF, Curry, Teyjus) and model checkers (Murphi). They are also recognized as crucial components

of proof assistants (Coq, Isabelle) and theorem provers for expressing computation as well as strategies.

Since pattern-matching is directly related to the structure of objects and therefore is a very natural programming language feature, it is a first class citizen of functional languages like ML or Haskell and has been considered recently as a useful add-on facility in object programming languages [OW97]. This is formally backed up by works like [CKL01] and particularly well-suited when describing various transformations of structured entities like, for example, trees/terms, hierarchized objects, and XML documents.

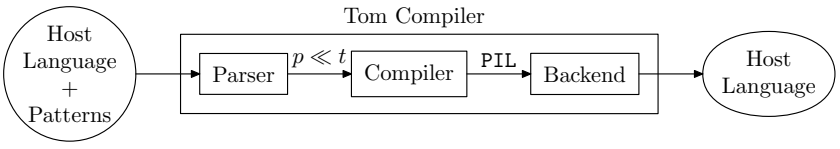
In this context, we are developing the Tom system [MRV03] which provides a generic way to integrate algebraic signatures and matching power in *existing* programming languages like Java, C or ML. For example, when using Java as the host language, and with a signature representing Peano numbers, the sum of two integers can be described in Tom as follows:

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

In this example, given two terms t_1 and t_2 that represent Peano integers, the evaluation of `plus` computes their sum using pattern matching. This definition of `plus` is given in a functional style, but now, after compilation, the `plus` function can be used elsewhere in a Java program to perform addition.

The notion of *mapping* is used to instruct the Tom compiler in how the signature is implemented in the host language. In this example, objects of type `Term` implement this signature. Using those *mappings*, the pattern matching code generated by the compiler is independent of the data structure, permitting indifferently the use of XML or objects for representing terms.

The general architecture of Tom, depicted as follows,



enlightens that a generic matching problem $p \ll t$ is compiled into an intermediate language code (`PIL`) that is supposed to compute a substitution σ iff $\sigma(p) = t$. A detailed description of the language and its architecture is given in Chapter 2.

Certifying pattern matching compilation

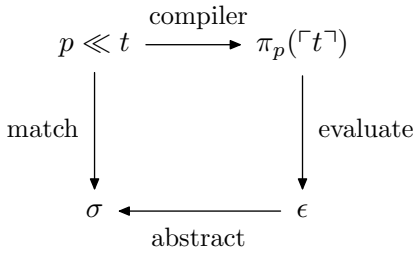
The addition of those high level constructs is intended to provide the user more confidence over the resulting code: the algorithms are expressed in a more algebraic or

mathematical way, and thus are easier to prove correct. Pattern matching is related to the structure of the manipulated objects, and the Tom language provides facility to use pattern matching on arbitrary object of the host language, like objects for Java and structures for C. The data structure independence is obtained by relying to a notion of *mapping*, linking the concrete structure and the abstract one manipulated by the Tom matching constructs.

The first concern is how we can ensure the Tom compiler does not introduce errors. It is common to blindly trust a particular compiler, because it is widely used and tested for example, but that does not provide a high level of confidence. Also, as the compiler is itself a software project, the problem is made more complex: it is not sufficient to prove the compiler correct once, but we also want to preserve this confidence when the compiler evolves.

Thus we describe a formal environment used to prove for each run of the compiler that the code produced for the input matching problems is correct [KMR05b]. This approach was chosen because it allows to treat the compiler as a black box, letting the compiler performing all kind of optimisations. The compiled code is proven *a posteriori*.

The program corresponding to a matching problem is examined, and its evaluation is proven equivalent to the original algebraic matching problem. This corresponds to proving the following diagram commutes, with $p \ll t$ being a matching problem, $\pi_p(\ulcorner t \urcorner)$ the compiled program corresponding to this matching problem, ϵ the environment obtained after the program evaluation and σ the algebraic substitution solution of the matching problem.



This equivalence is shown assuming that the data structure representing the algebraic terms in the concrete implementation is consistent with the terms it has to represent, which is formalized by the notion of *formal anchor*. This also defines the conditions the *mapping* has to satisfy to be used safely.

The definition of the *formal anchor* notion and its use to validate the compilation of patterns matching code is discussed in Chapter 3.

Efficient and safe canonical data structures

Another problem is the safety of the data structures the programmer using Tom can use. Tom itself can manipulate any data structure, provided there is a mapping for this structure. However, it can be tedious to implement such data structure in an efficient way, and then to provide a correct mapping. This is necessary to benefit from

the correctness of compilation proofs from Tom: the data structure has to be a *formal anchor*. This task is made even more difficult when data structure invariants have to be considered.

We formalize the Gom language, which allows for the description of such data structure and invariants, and generates a correct implementation [Rei06]. The invariants are described by the mean of *hooks*, a kind of aspects attached to different operators of the signature. The generated implementation features maximal sharing, and guarantees that the maximal sharing is preserved at run time, and that *hooks* are always called when building terms of the signature. It also allows to implement associative data structures, which can safely be used with the associative matching of Tom.

We show how the Gom language can be used to implement a prover for the system BV of the calculus of structures, which is a proof theoretical formalism generalizing sequent calculus with *deep inference*. Gom is used to represent the data structure and its invariants, and Tom to implement the rules. The implemented deduction system, integrating the invariants and techniques to reduce the impact of non-determinism is proved correct with respect to the original calculus, leading to an efficient and trustworthy prover [KMR05a].

The Gom language is presented in Chapter 4, followed by a description of its use in a prover for the system BV of the calculus of structures. Chapter 5 formally presents the links between the canonical forms as we can obtain with Gom and algebraic structures modulo associativity and neutral element.

Strategies and transformations

Because inference can be performed at any depth of a formula, *deep inference* implies being able to describe traversals over structures, seeking for a place where to apply one of the deduction rules. The need for high level descriptions of tree traversals and tree transformations led to the design and implementation of a generic strategy language, inspired by JJTraveler [Vis01] and Stratego itself inspired in turn by ELAN.

This strategy language provides an easy and efficient way to describe complex tree traversals. The strategies are type preserving, and allow to implement document transformations in an elegant manner. Allowing the user to specify new strategies either by using pattern matching, or by composing existing one bring higher order functions in Java, from `map/reduce` for lists to very complex transformations.

Providing congruence strategies and parametrized strategies along with recursive strategy definitions allow generic and higher order computations to be performed in a type safe manner in Java.

Those high level strategies can also be manipulated as algebraic objects themselves. This allows for a fully dynamical use of strategies in applications, as well as optimisations to be performed at the strategy level, and even just in time compilation of strategy expressions to efficient Java code, using partial evaluation techniques.

This strategy language, its design and use are discussed in Chapter 6.

Roadmap

The combination of Tom, Gom and this strategy language provides an integrated environment for implementing document transformations or compilers. They allow to use high level constructs to describe the transformations and control them, thus providing better security. The validation of the tools allows then to give the same confidence level to the application produced by the Tom and Gom compilers.

We first present a few preliminary notions about rewriting and rewriting based languages in Chapter 1. We then present the Tom language, as it is the main support for this work. The certification of the pattern matching compilation of Tom is presented in Chapter 3. This certification assumes given a formal anchor, and thus the Gom language, which generates such formal anchors is presented in Chapter 4. The strategy language is presented in Chapter 6. We give then some details about the developments methods in the Tom framework, focussed on improving the confidence in the compiler and making the use of Tom and all related tools easy and integrated into a development environment in Chapter 7 and conclude.

Introduction

Le sujet de cette thèse est la compilation de langages de haut niveau vers des langages de niveau moins élevé. Plus précisément, la question est de savoir quel niveau de confiance peut être accordé à ces compilateurs, et ce que l'on peut faire pour élever ce niveau de confiance.

Cette question est motivée par le développement de Tom*, qui est un compilateur pour des constructions de filtrage intégrées à un langage comme* Java*,* C *ou* ML*. Le langage* Tom *fournit des outils et fonctionnalités dont le but est de permettre l'écriture de compilateurs sûrs. Ceci se fait par l'introduction de constructions de haut niveau et de signatures aidant la description formelle d'algorithmes complexes et de manipulations d'arbres de syntaxe abstraits.*

Une première question est de savoir quel niveau de confiance attribuer au compilateur Tom *lui-même : si ce compilateur produit du code faux, comment pourrait-il nous permettre d'écrire de meilleurs compilateurs ? Nous développons pour résoudre ce problème une méthode de certification qui nous permet de prouver formellement à chaque exécution du compilateur que le code qu'il vient de générer implémente effectivement les constructions de filtrage qu'il avait à compiler.*

Cela nous conduit à fournir un générateur d'implémentations de structures de données avec partage maximal, efficaces et sûres, intégrant des invariants ainsi que leur préservation. Ces invariants, comme par exemple imposer aux listes d'être ordonnées ou aux éléments neutres d'être supprimés sont alors assurés par l'implémentation. Ce générateur de structures de données sûres est aussi une pierre angulaire pour le développement d'applications sûres avec Tom*.*

On s'intéresse alors à fournir un langage de haut niveau pour décrire des traversées et des transformations d'arbres, conduisant à une implémentation plus simple et plus sûre de transformations de documents complexes.

Une fois tous ces éléments rassemblés, on obtient un environnement pour décrire et implémenter des compilateurs de manière sûre. Cet environnement n'a pas pour but de fournir des compilateurs et transformations de documents entièrement vérifiés formellement, mais plutôt de fournir un ensemble de méthodes et outils facilitant le développement de telles transformations et augmentant le niveau de confiance dans le résultat de ce développement.

L'environnement Tom

La réécriture et le filtrage sont d'usage courant en mathématiques et informatique, pour décrire aussi bien des calculs que pour la déduction. Leur rôle est central dans les systèmes faisant de la notion de règle un objet explicite, comme les systèmes experts ou de règles métiers (JRule), les langages de programmation basés sur la logique équationnelle (OBJ), le calcul de réécriture (ELAN) ou sa logique (MAUDE), les langages fonctionnels, éventuellement logiques (ML, Haskell, ASF+SDF, Curry, Teyjus) ainsi que les vérificateurs de modèles (Murphi). Ces notions sont aussi des composants cruciaux pour les assistants de preuve (COQ, Isabelle) et les prouveurs de théorèmes, pour exprimer les calculs ou les stratégies.

Comme le filtrage est directement relié à la structure des objets, c'est donc une fonctionnalité naturelle pour un langage de programmation. C'est un citoyen de première classe des langages de programmation fonctionnelle comme ML ou Haskell, et il a récemment été considéré comme un ajout utile pour les langages orientés objet [OW97]. Cet ajout est particulièrement adapté à la description de transformations d'entités structurées variées, comme par exemple des termes ou arbres, des hiérarchie d'objets ou des documents XML.

Nous développons, dans ce contexte, le système Tom [MRV03], qui fournit un moyen générique d'intégrer les signatures algébriques et le filtrage dans les langages de programmation *existants*, comme Java, C ou ML. Par exemple, lorsqu'on utilise Java comme langage hôte, et avec une signature représentant les entiers de Peano, la somme de deux entiers peut être décrite en Tom comme suit :

```
Nat = | Zero()
      | Suc(Nat)
```

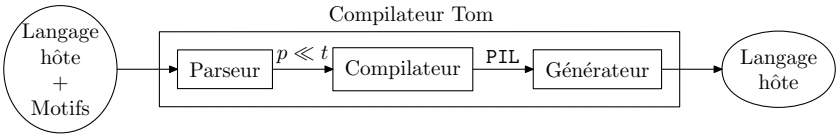
```
Term plus(Term t1, Term t2) {
  %match(Nat t1, Nat t2) {
    x,Zero()   -> { return `x; }
    x,Suc(y)   -> { return `Suc(plus(x,y)); }
  }
}
```

Dans cet exemple, étant donnés deux termes t_1 et t_2 représentant des entiers de Peano, l'évaluation de `plus` calcule leur somme en utilisant du filtrage. Cette définition de `plus` est donnée dans un style fonctionnel, mais après compilation, la fonction `plus` peut être utilisée n'importe où dans un programme Java pour effectuer l'addition.

La notion de *mapping* est utilisée pour expliquer au compilateur Tom comment la signature des termes algébriques est implémentée au niveau du langage hôte. Dans cet exemple, des objets de type `Term` implémentent la signature. En utilisant ces *mappings*, le code effectuant le filtrage généré par le compilateur est indépendant de la représentation de la structure de données, autorisant indifféremment l'utilisation de XML ou d'objets pour représenter les termes. L'indépendance du compilateur de filtrage avec la structure de termes manipulée est importante, car elle permet un développement modulaire des différents composants. On pourra alors faire évoluer ces composants séparément, tout en préservant une unité d'ensemble, les composants et leurs interactions étant bien identifiés. Cette idée de composants modulaires, coopérant pour la constitution d'un environnement

de spécification nous a guidés tout au long de ce travail, en infléchissant les choix de conception.

L'architecture générale de Tom, décrite pour ce qui concerne le filtrage par le schéma suivant,



montre qu'un problème de filtrage générique $p \ll t$ est compilé dans un programme écrit dans le langage intermédiaire (PIL) supposé calculer une substitution σ si et seulement si il existe $\sigma(p) = t$. Le chapitre 2 présente le langage Tom, et utilise ce langage pour motiver les différentes contributions.

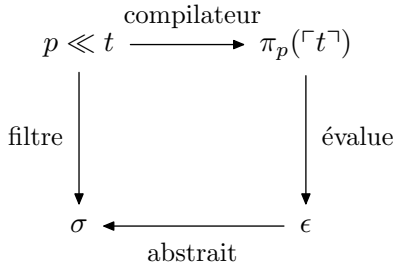
Certification de la compilation des constructions de filtrage

L'ajout de ces constructions de haut niveau a pour but de donner à l'utilisateur plus de confiance dans le code résultant : les algorithmes sont exprimés d'une manière plus formelle, et donc leur correction est plus facile à prouver. Le filtrage fait le lien avec la structure des objets manipulés, et le langage Tom permet d'utiliser le filtrage sur des objets arbitraires du langage hôte, comme des objets en Java ou des structures en C, en s'appuyant sur une notion d'ancrage qui lie la structure concrète des objets et la structure abstraite manipulée par les constructions de filtrage de Tom.

Notre premier souci est de s'assurer que le compilateur Tom lui-même n'introduise pas d'erreurs. Il est courant de se fier aveuglément à un compilateur particulier parce qu'il est largement utilisé et testé, mais cela ne fournit pas un niveau très élevé de confiance. De plus, comme un compilateur est avant tout un objet logiciel, le problème est rendu d'autant plus complexe : il n'est pas suffisant de prouver que le compilateur est correct à un instant t, mais il faut aussi s'assurer que cette confiance est préservée lorsque le compilateur évolue.

On décrit alors un environnement formel utilisé pour prouver, pour chaque exécution du compilateur, que le code produit pour les problèmes de filtrage d'entrée est correct [KMR05b]. Cette approche à été choisie parce qu'elle autorise à traiter le compilateur comme une boîte noire, en le laissant faire toutes sortes d'optimisations. Le code alors compilé et optimisé est prouvé correct *a posteriori*.

Le programme correspondant au problème de filtrage est alors examiné, et son évaluation est prouvée équivalente au problème de filtrage algébrique original. Ceci correspond à prouver que le diagramme suivant commute, avec $p \ll t$ un problème de filtrage, $\pi_p(\ulcorner t \urcorner)$ le programme compilé correspondant à ce problème de filtrage, ϵ l'environnement obtenu après évaluation du programme et σ la substitution algébrique solution du problème de filtrage.



Cette équivalence est montrée en supposant que la structure de données représentant les termes algébriques dans l'implantation concrète est cohérente avec les termes qu'elle doit représenter, ce qui est formalisé par la notion d'« ancrage formel ». Cette notion définit aussi les conditions qu'un ancrage doit satisfaire pour pouvoir être utilisé de manière sûre.

On définit cette notion d'ancrage formel ainsi que son utilisation pour la validation de la compilation de problèmes de filtrage dans le chapitre 3.

Structures de données canoniques, efficaces et sûres

La sûreté des structures de données que le programmeur utilisant Tom peut manipuler est un autre problème. Tom lui-même peut manipuler n'importe quelle structure de données, si elle est munie d'un ancrage. Cependant, il peut être difficile d'implémenter de telles structures de données de manière sûre puis de fournir un ancrage correct. Ceci est pourtant nécessaire pour profiter des preuves de correction de la compilation de Tom : la structure de données doit être un *ancrage formel*. Cette tâche est rendue encore plus difficile lorsque des invariants doivent être considérés au niveau de la structure de données.

On propose et formalise le langage Gom, qui permet la description de telles structures de données et invariants, et génère une implémentation correcte [Rei06]. Les invariants sont décrits par le biais de *hooks*, qui décrivent une forme d'aspects attachés aux différents opérateurs de la signature. L'implémentation générée est caractérisée par le partage maximal, et garantit que ce partage est préservé ainsi que le fait que les *hooks* sont toujours exécutés lors de la création de termes de la signature. Le langage autorise aussi l'implémentation de structures de données associatives qui peuvent être utilisées de manière sûre en conjonction avec le filtrage associatif de Tom.

On montre comment le langage Gom peut être utilisé pour implémenter un prouveur pour le système BV du calcul des structures, qui est un formalisme de théorie des preuves généralisant le calcul des séquents avec la notion d'*inférence profonde*. Gom est utilisé pour représenter la structure de données et ses invariants, tandis que Tom permet d'implanter les règles de déduction. Le système de déduction alors implémenté, intégrant les invariants et des techniques pour réduire l'impact du non-déterminisme, est prouvé correct vis-à-vis du calcul original, conduisant à un prouveur de confiance efficace [KMR05a].

Le chapitre 4 présente le langage Gom, puis décrit son utilisation dans le cas d'un prouveur pour le système BV du calcul des structures. Le chapitre 5 formalise alors les

liens entre les termes variadiques que l'on obtient avec Gom et les structures algébriques modulo associativité et élément neutre.

Stratégies et transformations

Parce que l'inférence peut être faite à n'importe quelle profondeur dans une formule, l'*inférence profonde* impose d'être capable de décrire des traversées à travers les structures de données, à la recherche de positions où appliquer l'une des règles de déduction. Le besoin de descriptions de haut niveau des traversées et transformations d'arbres nous a conduit à la conception et à l'implémentation d'un langage de stratégies générique, inspiré par JJTraveler [Vis01] et Stratego, lui-même inspiré par ELAN.

Ce langage de stratégies nous donne une manière aisée et efficace de décrire des traversées complexes d'arbres. Les stratégies ont la propriété de préserver les types dans la structure de données à transformer, et permettent d'implanter des transformations de documents élégamment. Permettre à l'utilisateur de spécifier de nouvelles stratégies soit en utilisant filtrage et réécriture, soit en composant des stratégies existantes apporte des fonctions d'ordre supérieur à Java, allant de `map/reduce` sur les listes à des transformations d'arbres plus élaborées.

De plus, en fournissant des stratégies de congruence et des stratégies paramétrées, ainsi que la possibilité de définir des stratégies de manière récursive, nous permettons de définir des calculs génériques et d'ordre supérieur préservant la sûreté de typage et Java.

Ces stratégies de haut niveau peuvent aussi être elles-mêmes manipulées comme des objets algébriques. Ceci permet une utilisation entièrement dynamique des stratégies dans les applications, ainsi que de spécifier des optimisations effectuées au niveau des stratégies. Cela autorise aussi une compilation *juste à temps* des expressions de stratégies vers du code Java efficace, en utilisant des techniques d'évaluation partielle.

Le langage de stratégies ainsi que son extension réflexive sont décrits dans le chapitre 6.

La combinaison de Tom, Gom et de notre langage de stratégies fournit un environnement intégré pour l'implantation de transformations de documents ou compilateurs. Elle permet l'utilisation de constructions de haut niveau pour décrire les transformations et les contrôler, fournissant ainsi plus de sûreté. La validation des outils permet alors de donner le même niveau de confiance aux applications produites par les compilateurs Tom et Gom qu'à leurs descriptions au niveau source.

Plan de la thèse

On présente tout d'abord quelques notions préliminaires sur la réécriture et les langages basés sur la réécriture dans le chapitre 1. On présente alors le langage Tom dans le chapitre 2, car il est le support principal de ce travail. La certification de la compilation de constructions de filtrages est alors présentée dans le chapitre 3, ainsi que sa mise en œuvre au sein du compilateur Tom. Cette certification suppose la présence d'ancrages formels, et ainsi le langage Gom, qui permet la génération de tels ancrages

formels est présenté dans le chapitre 4. Le chapitre 5 traite des liens que l'on peut établie entre les structures variadiques générées par le compilateur Gom et les opérateurs algébriques associatifs avec élément neutres. Le langage de stratégies est présenté dans le chapitre 6. On donne alors quelques détails sur les méthodes de développement dans l'environnement Tom, dont l'objectif est d'accroître la confiance dans ce compilateur et de permettre l'utilisation facile et intégrée dans un environnement de développement de Tom et des outils associés dans le chapitre 7. Enfin, la conclusion dresse un bilan et donne des directions dans lesquelles ce travail devrait être poursuivi.

1. Notions préliminaires

> "When I use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean – neither more nor less."
>
> *(Lewis Carroll)*

Ce premier chapitre rassemble quelques notions auxquelles nous ferons appel par la suite. Nous traiterons tout d'abord de la réécriture de termes, puis de ses extensions, qui fournissent un pouvoir expressif permettant la construction d'un langage de programmation basé sur ce principe. Finalement, nous survolerons les différents langages et systèmes basés sur cette idée de réécriture, ainsi que diverses applications de ces langages.

1.1. Réécriture

1.1.1. Termes

Nous donnons dans cette section les notions de base concernant les algèbres de termes du premier ordre.

Définition 1 (Signature)**.** *Une signature $\mathcal{F}$ est un ensemble d'opérateurs (symboles de fonction), chacun étant associé à un entier naturel par la fonction arité, $ar : \mathcal{F} \to \mathbb{N}$. On note par $\mathcal{F}_n$ le sous-ensemble des opérateurs d'arité n. L'ensemble $\mathcal{F}_0$ est appelé ensemble des constantes.*

On peut alors définir les termes construits sur ces opérateurs et un ensemble infini dénombrable de variables $\mathcal{X}$.

Définition 2 (Termes)**.** *Étant donnés un ensemble infini dénombrable de variables $\mathcal{X}$ et une signature $\mathcal{F}$, on définit l'ensemble des termes $\mathcal{T}(\mathcal{F}, \mathcal{X})$ comme le plus petit ensemble tel que :*

- *$\mathcal{X} \subset \mathcal{T}(\mathcal{F}, \mathcal{X})$: toute variable de $\mathcal{X}$ est un terme de $\mathcal{T}(\mathcal{F}, \mathcal{X})$;*
- *pour tous $t_1, \ldots, t_n$ éléments de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ et pour tout opérateur f d'arité n, le terme $f(t_1, \ldots, t_n)$ est un élément de $\mathcal{T}(\mathcal{F}, \mathcal{X})$.*

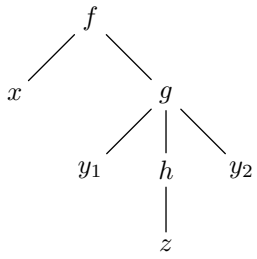
Pour tout terme t de la forme $f(t_1, \ldots, t_n)$, le symbole de tête de t, noté $\mathcal{S}ymb(t)$ est par définition l'opérateur f.

Définition 3 (Variables)**.** *L'ensemble $Var(t)$ des variables d'un terme t est défini de manière inductive :*

- $Var(t) = \emptyset$ *si* $t \in \mathcal{F}_0$,
- $Var(t) = \{t\}$ *si* $t \in \mathcal{X}$
- $Var(t) = \bigcup_{i=1}^{n} Var(t_i)$ *si* $t = f(t_1, \ldots, t_n)$

Un terme t est clos si $Var(t) = \emptyset$. L'ensemble des termes clos est noté $\mathcal{T}(\mathcal{F})$.

On peut considérer un terme comme un arbre dont les feuilles sont des constantes ou des variables et les nœuds sont des opérateurs d'arité non nulle. Par exemple, le terme $f(x, g(y_1, h(z), y_2))$ est représenté par l'arbre :



Chaque nœud dans un arbre peut être identifié par sa position, définie par une suite d'entiers naturels représentant le chemin depuis la racine jusqu'à ce nœud.

Définition 4 (Position)**.** *Une position dans un terme t est représentée par une suite ω d'entiers naturels positifs décrivant le chemin de la racine du terme jusqu'à la racine du sous-terme à cette position, noté $t_{|\omega}$. Un terme u a une occurrence dans t si $u = t_{|\omega}$ pour une position ω dans t.*

On note $t[t']_\omega$ pour indiquer le fait que le terme t a le terme t' à la position ω et on note $t[\omega \hookleftarrow s]$ pour signifier que le sous-terme $t_{|\omega}$ a été remplacé par s dans t. On a ainsi $t[t']_\omega \Leftrightarrow t_{|\omega} = t'$ et $t' = t[\omega \hookleftarrow s] \Rightarrow t'_{|\omega} = s$.

1.1.2. Filtrage

Une substitution est une opération de remplacement, uniquement définie par une fonction des variables vers les termes clos.

Définition 5 (Substitution)**.** *Une substitution σ est une fonction de $\mathcal{X}$ vers $\mathcal{T}(\mathcal{F})$, notée lorsque son domaine est fini $\sigma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$. Cette fonction s'étend de manière unique en un endomorphisme $\sigma' : \mathcal{T}(\mathcal{F}, \mathcal{X}) \to \mathcal{T}(\mathcal{F}, \mathcal{X})$ sur l'algèbre des termes, qui est défini inductivement par :*

- $\sigma'(x) = \begin{cases} \sigma(x) \text{ pour toute variable } x \in \mathrm{Dom}(\sigma) \\ x \text{ sinon ;} \end{cases}$
- $\sigma'(f(t_1, \ldots, t_n)) = f(\sigma'(t_1), \ldots, \sigma'(t_n))$ *pour tout symbole de fonction* $f \in \mathcal{F}_n$.

Si σ est totale, c'est à dire qu'à chaque variable est associé un terme, alors $\sigma'(\mathcal{T}(\mathcal{F}, \mathcal{X})) \subseteq \mathcal{T}(\mathcal{F})$.

Définition 6 (Filtrage). *Étant donnés un motif $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et un terme clos $t \in \mathcal{T}(\mathcal{F})$, p filtre t, noté $p \ll t$, si et seulement s'il existe une substitution σ telle que $\sigma(p) = t$.*

$$p \ll t \Leftrightarrow \exists \sigma, \sigma(p) = t$$

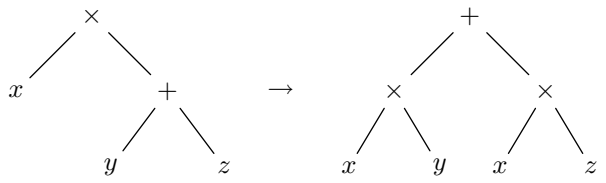
La négation du filtrage sera notée $p \not\ll t$.

1.1.3. Systèmes de réécriture

Une théorie équationnelle peut être définie par un ensemble d'égalités. L'idée centrale de la réécriture est d'imposer une direction dans l'utilisation des égalités, par la définition de règles de réécriture, qui définissent un calcul.

Définition 7 (Règle). *Une règle de réécriture est un couple (l, r) de termes dans $\mathcal{T}(\mathcal{F}, \mathcal{X})$, notée $l \to r$.*

l est le membre gauche de la règle, et r son membre droit. Un exemple de règle de réécriture est la distributivité de la multiplication dans un anneau par rapport à l'addition :



Définition 8 (Système de réécriture). *Un système de réécriture sur les termes est un ensemble de règles de réécriture tel que :*
- *les variables du membre droit de la règle font partie des variables du membre gauche ($Var(r) \subseteq Var(l)$) ;*
- *le membre gauche d'une règle n'est pas une variable ($l \notin \mathcal{X}$).*

Considérons un système de réécriture R. On définit la relation binaire de réécriture $\to_R$ associée à ce système.

Définition 9 (Réécriture). *Un terme $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ se réécrit en t' dans le système R, ce que l'on note $t \to_R t'$ s'il existe :*
- *une règle $l \to r \in R$,*
- *une position ω dans t,*
- *une substitution σ telle que $t_{|\omega} = \sigma(l)$ et $t' = t[\omega \hookleftarrow \sigma(r)]$.*

Le sous-terme $t_{|\omega}$ est appelé *radical*.

Définition 10 (Fermetures). *Étant donnée une relation binaire $\to$, on note :*
- *$\xrightarrow{*}$ la fermeture réflexive et transitive de $\to$;*
- *$\overset{*}{\leftrightarrow}$ la fermeture réflexive, symétrique et transitive de $\to$, qui est alors une relation d'équivalence.*

Soient deux relations binaires $\to_1$ et $\to_2$, on note $\to_1 \circ \to_2$ la composition de ces relations.

On dit que deux termes t_1 et t_2 sont joignables s'il existe un terme v tel que $t_1 \xrightarrow{*} v$ et $t_2 \xrightarrow{*} v$. On note $t_1 \downarrow t_2$ le fait que t_1 et t_2 soient joignables.

Définition 11 (Réductibilité)**.** *Soit une relation binaire $\to$ sur un ensemble T. Un élément t de T est réductible par $\to$ s'il existe t' dans T tel que $t \to t'$, sinon, il est dit irréductible. On appelle* forme normale *de t tout élément t' irréductible de T tel que $t \xrightarrow{*} t'$.*

1.1.4. Confluence et terminaison

Afin de savoir si deux termes appartiennent à une même classe d'équivalence pour la relation d'équivalence induite par la relation $\to$, $\overset{*}{\leftrightarrow}$, on peut envisager de calculer une forme normale pour chacun des deux termes et vérifier si elles sont égales. Cela n'est cependant possible que si pour chaque élément il existe effectivement une forme normale, et que de plus elle est unique.

Définition 12 (Terminaison)**.** *Une relation $\to$ sur un ensemble T est* terminante *si il n'existe pas de suite infinie $(t_i)_{i\geq 1}$ d'éléments de T telle que $t_1 \to t_2 \to \cdots$.*

Les formes normales pour une relation $\to$ existent pour tout terme si la relation *termine*. Lorsqu'une forme normale existe, son unicité est assurée par la confluence, ou par la propriété de Church-Rosser, qui est équivalente.

Définition 13 (Confluence)**.** *Soit une relation binaire $\to$ sur un ensemble T.*

– *$\to$ vérifie la propriété de Church-Rosser si et seulement si*

$$\forall u, v,\ u \overset{*}{\leftrightarrow} v \Rightarrow \exists w,\ (u \xrightarrow{*} w \text{ et } v \xrightarrow{*} w)$$

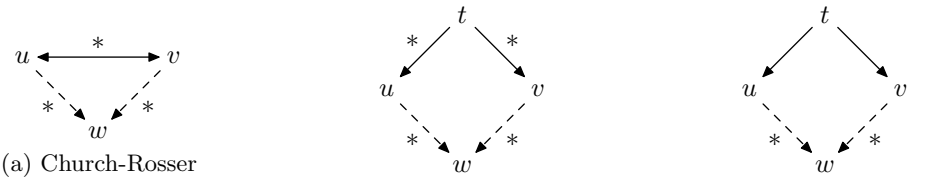
– *$\to$ est confluente si et seulement si*

$$\forall t, u, v,\ (t \xrightarrow{*} u \text{ et } t \xrightarrow{*} v) \Rightarrow \exists w,\ (u \xrightarrow{*} w \text{ et } v \xrightarrow{*} w)$$

– *$\to$ est localement confluente si et seulement si*

$$\forall t, u, v,\ (t \to u \text{ et } t \to v) \Rightarrow \exists w,\ (u \xrightarrow{*} w \text{ et } v \xrightarrow{*} w)$$

On peut représenter simplement ces définitions par des diagrammes de la figure 1.1, dans lesquels un trait plein dénote une hypothèse, et les pointillés dénotent une conclusion.



(a) Church-Rosser (b) Confluence (c) Confluence locale

FIG. 1.1: Propriétés sur les relations binaires

1.2. Pouvoir expressif de la réécriture

Les systèmes de réécriture permettent de modéliser des comportements calculatoires. Si l'on parvient à orienter les équations de définition d'une congruence, de façon à obtenir un système de réécriture confluent et terminant, on obtient une notion de forme normale, qui est unique, et permet de décider du *problème du mot* dans cette congruence. Pour décider si deux termes t_1 et t_2 sont équivalents pour cette congruence, il suffit de comparer avec l'égalité sur les termes leurs formes normales ; si elles sont égales, alors en utilisant la propriété de Church-Rosser, on en déduit que les termes sont équivalents.

Cependant, la plupart des systèmes de réécriture ne possèdent pas ces propriétés de terminaison ou confluence. Plutôt que de transformer le système afin d'obtenir une version utilisable, on préfère imposer un certain contrôle sur l'application des diverses règles.

1.2.1. Réécriture conditionnelle

En ajoutant des conditions sur l'application des règles de réécriture, on peut étendre les systèmes de réécriture à des systèmes de réécriture conditionnelle. Plusieurs définitions des systèmes de réécriture conditionnelle ont été proposés, avec comme différence principale l'interprétation des conditions.

Les systèmes de réécriture conditionnelles *standards* sont constitués de règles de réécriture de la forme

$$l \to r \text{ si } s_1 \downarrow t_1 \wedge \ldots \wedge s_n \downarrow t_n$$

La règle $l \to r$ est appliquée comme dans la définition 9 si avec la substitution σ obtenue pour l'application de la règle, on peut pour tout i entre 1 et n joindre les termes $\sigma(s_i)$ et $\sigma(t_i)$.

Une autre définition de la réécriture conditionnelle, dite *normale*, a des règles de réécriture de la forme

$$l \to r \text{ si } s_1 \xrightarrow{*}_! t_1 \wedge \ldots \wedge s_n \xrightarrow{*}_! t_n$$

dans lesquelles $s \xrightarrow{*}_! t$ indique que t est une forme normale de s.

Note A. Il est possible de transformer un système de réécriture standard en un système de réécriture normal en ajoutant les symboles *eq* et *true* au système, ainsi que la règle

$eq(x, x) \to true$, et en remplaçant les règles par

$$l \to r \text{ si } eq(s_1, t_1) \xrightarrow{*}_! true \wedge \ldots \wedge eq(s_n, t_n) \xrightarrow{*}_! true$$

Les termes ne comportant pas les nouveaux symboles eq et $true$ auront alors des réductions similaires dans le système standard original et le système normal produit.

1.2.2. Réécriture sous stratégies

Dans le cadre de la réécriture de termes classique, lorsqu'on cherche à calculer une forme normale, la méthode qui détermine l'application des règles de réécriture est implicite ; les règles sont appliquées exhaustivement sur un terme. C'est à dire que les règles sont appliquées sur le terme jusqu'à ce que plus aucune règle ne puisse être appliquée. Si le système de réécriture R est *Church-Rosser* et terminant, l'ordre d'application n'aura aucun effet sur le résultat, mais peut affecter l'efficacité du calcul, c'est à dire le nombre de pas de réécriture nécessaires pour atteindre une forme normale.

Cependant, lorsqu'on utilise des systèmes de réécriture de termes pour effectuer des calculs, il est souvent naturel d'écrire des systèmes de réécriture qui ne terminent pas, ou ne sont pas confluents. L'ensemble des règles de réécriture intéressantes pour un problème donné est généralement non-confluent et non-terminant. Il est alors nécessaire de contrôler l'application des règles du système de réécriture, en ajoutant un système de contrôle sur l'application des règles du système.

Exemple 1. Avec la signature $\{a, f, g\}$, a d'arité 0, f et g d'arité 1, le système

$$\begin{cases} f(x) & \to & f(f(x)) \\ g(x) & \to & g(g(x)) \\ f(a) & \to & a \\ g(a) & \to & a \end{cases}$$

ne termine pas, même si il est confluent. En effet, il existe une suite infinie de pas de réécriture partant de $f(a)$ ou $g(a)$. Ils se réduisent pourtant tous deux en a.

Un *système de calcul* est une théorie de la logique de réécriture à laquelle on ajoute un ensemble de stratégies décrivant les calculs autorisés [KKV95]. Ces stratégies spécifient le parcours à suivre dans l'arbre de toutes les dérivations, et décrivent quels sont les nœuds de cet arbre de dérivation qu'il faut considérer comme des résultats. Elles permettent donc de décrire le déroulement de cette dérivation, et en même temps de réduire l'espace de recherche à explorer. En effet, une augmentation du nombre de règles de réécriture dans un système et de la taille des termes traités entraîne une explosion de l'espace des dérivations possibles. Or, tous les termes qui se trouvent dans cet arbre des dérivations sont considérés comme des résultats, ce qui montre l'importance et la nécessité d'un tel mécanisme de contrôle.

Exemple 2. Une stratégie permettant de faire terminer les calculs sur les termes clos dans le système de l'exemple 1 est de donner la priorité à l'application des règles $f(a) \to a$ et $g(a) \to a$ sur les autres règles.

Les stratégies peuvent prendre plusieurs formes. La stratégie de l'exemple 2 permet de sélectionner les règles appliquées, en donnant une priorité, mais les règles sont appliquées à toutes les positions dans le terme. D'autres stratégies permettent de spécifier les positions où appliquer les règles.

1.2.3. Réécriture modulo une théorie

Les termes sont utilisés pour représenter les objets du calcul. Certaines théories associées à ces termes, comme l'associativité, la commutativité ou les éléments neutres sont communément utilisées lors de la description du problème à modéliser à l'aide de règles de réécriture.

Ces théories équationnelles définissent des classes d'équivalence entre les termes, et afin de faciliter la définition des règles de réécriture, on s'intéresse à la définition des opérations sur les termes modulo cette théorie équationnelle.

Pour introduire cette notion, nous allons nous baser sur un exemple : le filtrage associatif-commutatif. Cela permet de modéliser le calcul dans les groupes ou sur d'autres structures algébriques dotées d'opérateurs commutatifs et associatifs. Typiquement, cela permet quand on calcule sur les entiers, de ne pas avoir à spécifier les cas des différentes permutations.

Exemple 3. Le symbole d'arité 2 « *add* » est associatif-commutatif (AC). Les axiomes d'associativité et commutativité sont les suivants :

$$\forall x, y;\ add(x, y) = add(y, x)$$
$$\forall x, y;\ add(x, add(y, z)) = add(add(x, y), z).$$

- Le problème de filtrage $add(x, y) \ll add(a, b)$ possède alors deux solutions distinctes modulo AC : $\sigma_1 = \{x \mapsto a, y \mapsto b\}$ et $\sigma_2 = \{x \mapsto b, y \mapsto a\}$.
- Le problème de filtrage $add(x, y) \ll add(add(a, b), c)$ possède quant à lui 6 solutions modulo AC :

$$\sigma_1 = \{x \mapsto a, y \mapsto add(b, c)\} \qquad \sigma_2 = \{x \mapsto add(b, c), y \mapsto a\}$$
$$\sigma_3 = \{x \mapsto b, y \mapsto add(a, c)\} \qquad \sigma_4 = \{x \mapsto add(a, c), y \mapsto b\}$$
$$\sigma_5 = \{x \mapsto c, y \mapsto add(a, b)\} \qquad \sigma_6 = \{x \mapsto add(a, b), y \mapsto c\}$$

Dans cet exemple, on voit informellement ce qu'est un axiome. Une théorie équationelle est l'ensemble des classes d'équivalence sur une structure modulo un ensemble d'axiomes.

1.3. Langages basés sur la réécriture

Les systèmes d'équations et les systèmes de réécriture permettent de spécifier des problèmes et offrent la puissance des machines de Turing [Dau89]. Ces spécifications doivent ensuite être rendues exécutables.

1.3.1. La réécriture comme paradigme de calcul

La logique de réécriture offre par exemple un cadre adéquat pour la définition d'un langage de programmation.

Étant donnée une théorie dans la logique de réécriture représentée par un système de réécriture R et un terme t clos, nous cherchons tous les termes t' vérifiant

$$t \xrightarrow{*}_R t'.$$

On doit pour cela explorer toutes les dérivations obtenues à partir de t en appliquant les règles de R. Cette recherche de solutions peut être considérée comme une procédure algorithmique, ou comme une sémantique opérationnelle d'un langage de programmation basé sur la logique de réécriture. Un programme d'un tel langage est constitué d'un ensemble de règles de réécriture et d'un terme de départ.

Dans le cas où le système R n'est pas confluent, il est possible d'obtenir suivant l'ordre d'application des règles plusieurs résultats. Ceci permet de modéliser l'évolution de systèmes non-déterministes comme c'est le cas des simulations de protocoles réseau ou la recherche de preuves.

1.3.2. Des langages basés sur la réécriture

De nombreux langages de programmation ont été conçus pour exploiter les concepts de la réécriture. OBJ [GWM+93] peut être considéré comme l'aïeul de cette lignée.

OBJ Ce projet initié par Joseph Goguen [Gog78], puis développé en collaboration avec Kokichi Futatsugi, Jean-Pierre Jouannaud et José Meseguer [GWM+93] met l'accent sur le formalisme de spécification et l'expressivité du langage. Il propose de définir des signatures avec inclusion de sortes, des systèmes de réécriture modulo toutes les combinaisons de l'associativité et la commutativité ainsi que des modules paramétrés et la notion de vue.

Maude Directement inspiré d'OBJ et introduit en 1996 [CM00], Maude enrichit le formalisme d'OBJ en intégrant la notion de réflexivité, la programmation objet ainsi qu'en permettant d'effectuer la réécriture modulo des axiomes plus variés, comme les éléments neutres.

ASF+SDF Jan Heering et Paul Klint ont mis en route le projet ASF+SDF dans les années 1980, avec pour objectif de définir un environnement de programmation générique permettant d'éditer, exécuter et déboguer des programmes écrits dans un langage spécifié par une grammaire [BHKO02b]. Le premier interpréteur vit le jour en 1989, et le premier compilateur en 1993. Il est utilisé par exemple pour effectuer automatiquement de la rénovation d'applications critiques en COBOL.

Elan Le langage ELAN a été élaboré au sein du projet Protheo à Nancy au début des années 90 [Vit94]. ELAN à été conçu comme cadre logique pour le prototypage de systèmes de calcul. Une de ses contributions fondamentales est l'introduction d'un langage de stratégies permettant à l'utilisateur de définir finement comment l'espace de recherche doit être exploré. Le système ELAN a beaucoup évolué au cours du temps. Un premier compilateur à été écrit par Marian Vittek [Vit96], puis un second compilateur du langage permettant d'utiliser des symboles associatifs-commutatifs à été développé [MK98] ; le langage de stratégies a été rendu plus expressif par Peter Borovanský [Bor98].

Stratego Eelco Visser, s'inspirant du langage ASF+SDF ainsi que du langage de stratégies d'ELAN a défini le langage de transformations de programmes Stratego à partir de 1998 [VBT98]. Dans ce cadre, il est possible de simplifier le langage de stratégies (supprimant la gestion du non-déterminisme par exemple) pour obtenir un langage de transformation efficace et simple. Ce projet repose aussi sur l'héritage d'ASF+SDF en reprenant les techniques d'analyse syntaxique et grammaires.

Les langages à base de réécriture ont permis de traiter des problèmes variés, montrant la généralité du paradigme de réécriture, ainsi que l'expressivité et la puissance de celle-ci comme outil de programmation.

Parmi ces applications, on peut citer dans divers domaines :

- *Les prouveurs de théorèmes*
  Il existe une implantation du prouveur de prédicats B en ELAN [CK97] ;
- *Vérification de protocoles*
  Vérification du protocole d'authentification de Needham-Schroeder [Cir01] ;
- *Transformation de programmes*
  La définition d'optimiseurs de programmes en Stratego [VBT98],
  Définition d'afficheurs de programmes avec ASF+SDF [BV96], restructuration de code COBOL [BSV97, BSV98] ;
- *Compilation*
  Le compilateur Stratego est lui-même écrit avec Stratego et *bootstrappé*. Le compilateur d'ASF+SDF est quand à lui décrit avec ASF+SDF [BHKO02a] ;
- *Résolution de contraintes*
  La combinaison d'algorithmes d'unification [Rin97], la résolution de problèmes de satisfaction de contraintes [Cas98] ainsi qu'un un algorithme d'unification d'ordre supérieur [Bor95].
- *Complétion de systèmes équationnels*
  Deux implantations de la procédure de complétion de Knuth-Bendix [KM95, KLS96]

2. Le langage Tom

> The most important thing in the programming language is the name. A language will not succeed without a good name. I have recently invented a very good name and now I am looking for a suitable language.
>
> *(Donald Knuth)*

2.1. Motivations et historique

L'idée principale du langage Tom est d'intégrer les concepts et techniques de la réécriture dans les langages de programmation généralistes et largement utilisés, comme Java ou C.

Le langage Tom a tout d'abord été présenté comme un simple préprocesseur pour les langages de type C ou Java, permettant d'ajouter une primitive de filtrage [MRV03].

L'objectif premier était de rendre l'écriture d'un compilateur pour un langage à base de règles plus facile, en autorisant le développeur à s'intéresser aux différents aspects du travail d'implantation d'un tel compilateur séparément, en fournissant une brique réutilisable implantant le filtrage. En effet, la compilation des primitives de filtrage intervient à tous les stades de la compilation d'un langage tel qu'ELAN, et cela interfère généralement avec la compilation des autres traits du langage, comme le parsing des signatures, les structures de contrôle de l'application des règles et les stratégies.

Le compilateur pour le langage Tom a très rapidement été *bootstrappé* afin de permettre l'utilisation de filtrage pour l'écriture de compilateur. C'est alors que la langage à évolué pour être utilisé par des utilisateurs humains.

2.2. Le cœur du langage

Nous ne décrirons pas dans ce chapitre le langage Tom de manière formelle. Les idées de base du langage sont décrites par les auteurs du tout premier compilateur Tom [MRV03], et le langage et sa librairie ainsi que les outils associés sont décrits en détail dans le manuel de référence du langage [BBK+06]. Nous nous attacherons donc plutôt ici à décrire le langage par de petits exemples, ainsi qu'à présenter rapidement les notions qui seront utiles par la suite.

2.2.1. Filtrage dans Java

Le langage Java, comme le langage C et la plupart des langages de programmation impératifs habituellement utilisés, ne comporte pas de notion de types algébriques, ou termes, mais uniquement des types de données dits « scalaires », ou des structures composées comme les objets de Java ou les structures de C.

De même, ces langages ne contiennent pas d'instruction de filtrage, permettant de tester la présence de certains motifs dans une structure de données, et d'instancier des variables en fonction du résultat de l'opération de filtrage.

Ces constructions sont pourtant un des constituants essentiels des langages logiques et des langages fonctionnels. Ainsi, les types inductifs et le filtrage sur les objets appartenant à ces types inductifs sont les premières instructions que les nouveaux programmeurs Caml apprennent.

Le langage Tom permet d'apporter ces constructions de filtrage dans les langages comme Java et C. De plus, les constructions de filtrage de Tom étant plus expressives que le filtrage du langage Caml, il est aussi possible de les utiliser dans ce dernier, afin d'utiliser le filtrage équationnel. Les constructions de filtrage de Tom sont introduites par le lexème **%match**. Cette construction est une extension de la construction classique `switch/case`, avec la différence principale que la discrimination se fait 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.

Exemple 4. Un exemple élémentaire pour présenter Tom est tout simplement la définition de l'addition sur les entiers de Peano. On représente alors les entiers par les termes constitués du *zéro*, `Zero()`, qui est une constante, et du *successeur*, `Suc(x)`, qui prend en argument un entier de Peano. L'addition est alors définie en Tom, en utilisant le langage hôte Java :

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

Dans cet exemple, étant donnés deux termes t_1 et t_2, représentant des entiers de Peano, l'évaluation de la fonction `plus` retourne la somme de t_1 et t_2. Ce calcul est implémenté par filtrage : x filtre t_1 et les motifs $Zero()$ et $Suc(y)$ filtrent éventuellement t_2. Lorsque le motif $Zero()$ filtre t_2, le résultat de l'évaluation de la fonction `plus` est x, qui est instancié par t_1 par filtrage. Lorsque le motif $Suc(y)$ filtre t_2, cela signifie que le terme t_2 a Suc comme symbole de tête ; le sous-terme y est alors ajouté à x, et le successeur de ce terme est retourné. L'expression de cette fonction `plus` est donnée d'une manière fonctionnelle, mais définit une fonction Java, qui peut alors être utilisée de manière classique.

La construction « ‘ » (appelée *backquote*) est utilisée pour exprimer la construction de termes de manière algébrique, et permet d'utiliser les variables qui sont instanciées par filtrage. Cette construction permet aussi d'utiliser la fonction `plus` comme un constructeur algébrique de sorte `Nat`, effectuant ainsi l'appel récursif.

Le code de l'exemple 4 montre aussi l'une des caractéristiques principales de Tom : les constructions **%match** sont intégrées de manière peu intrusive au sein du langage hôte. Le compilateur Tom ne parse pas les constructions du langage hôte, mais les considère simplement comme du texte englobant les constructions de filtrage, et remplace ces constructions de filtrage par des séquences d'instructions du langage hôte correspondantes. Seules les constructions de Tom sont examinées en détail pour être traduites, mais le compilateur n'utilise pas d'information provenant du langage hôte, considérant ces parties du programme comme du texte simple.

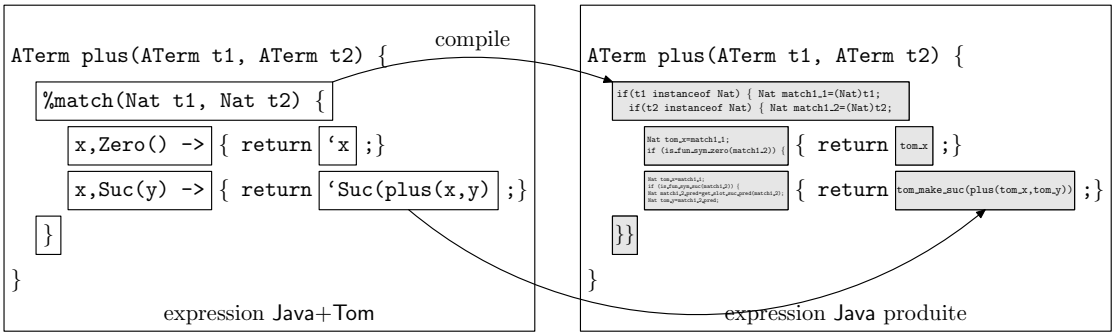


FIG. 2.1: Compilation d'un programme Java+Tom : les constructions Tom sont traduites par des instructions Java, et le code Java n'est pas modifié

Les parties du programme correspondant au *code hôte* ne sont pas examinées par le compilateur. Le programme contient aussi des constructions de filtrage qui seront remplacées par le compilateur par les séquences d'instructions du langage hôte correspondantes, les portions que le compilateur n'examine pas n'étant pas modifiées. Comme le montre la figure 2.1, seules les constructions Tom sont modifiées par le processus de compilation, le code hôte étant laissé intact.

2.2.2. *Mappings* : notion d'ancrage

On peut remarquer dans l'exemple 4 que la fonction Java `plus` prend en argument deux objets de type `ATerm`, alors que la construction de filtrage est spécifiée sur le type de données algébrique `Nat`.

Généricité et composants réutilisables

Cela illustre un autre aspect de la généricité du langage Tom : le filtrage est compilé indépendamment de l'implémentation concrète des termes. Ainsi, les constructions de

filtrage sont exprimées sur un type de données algébrique, et le compilateur les traduit dans des manipulations du type de données concret qui représente ces termes algébriques. C'est à l'utilisateur du langage de spécifier le type algébrique des termes manipulés, le type de données concret qui le représente, ainsi que comment ces objets concrets représentent les termes algébriques, par l'intermédiaire d'un ancrage.

Cette généricité est une spécificité ainsi qu'une caractéristique forte du langage Tom et des outils associés : chaque outil est conçu comme un composant réutilisable distinct. Ainsi, chaque composant, comme le compilateur Tom, mais aussi le compilateur pour le langage Gom et le langage de stratégies qui seront présentés dans les chapitres 4 et 6 peut être utilisé indépendamment des autres. Cela permet de faire évoluer les composants de manière indépendante, ainsi que de contrôler les problèmes de maintenance des outils. En effet, l'effort fait pour la conception de composants réutilisables est compensé par la facilité de maintenance et d'évolution de ces différents composants.

Ce découpage en composants distincts peut avoir un coût : certaines optimisations ne sont pas possibles. Ainsi, lorsque la structure de données est connue et contrôlée lors du développement d'un compilateur de filtrage, des optimisations lors de la compilation du filtrage sont possibles, qui ne le sont pas lorsque cette structure est un paramètre du système. De même, cela simplifierait la conception du compilateur, en supprimant le besoin d'un ancrage.

Cependant, une telle intégration impose de faire évoluer les différents composants (dans notre cas, la représentation des termes, leur analyse par filtrage et le contrôle des transformations) simultanément, chaque évolution d'une partie imposant de mettre à jour le tout. Une conception en composants distincts permet *a contrario* de circonscrire et contrôler les coûts de changement dans les différents composants.

Implémentation des structures d'arbre

Pour que l'exemple 4 soit complet, il faudrait tout d'abord définir la sorte algébrique `Nat`, comme étant composée des opérateurs : $\{$`Zero` $:\rightarrow$ `Nat`, `Suc` : `Nat` $\rightarrow$ `Nat`$\}$. Ensuite, il est nécessaire de choisir une implantation pour les objets qui représenteront ces termes algébriques : on utilisera pour cet exemple la bibliothèque ATerm [BJKO00], qui fournit une interface pour représenter et manipuler des structures de termes. Une description détaillée des particularités de cette bibliothèque est donnée dans la section 4.1.1. En particulier, cette bibliothèque permet de représenter les termes algébriques comme l'application d'un symbole de tête à une liste d'arguments, et de manipuler cette liste d'arguments.

Ainsi, le rôle du *mapping* est de définir comment les opérations sur la structure de données abstraite se traduisent dans des instructions du langage hôte. Dans notre exemple, on devrait alors expliquer comment il est possible de représenter le terme algébrique `Zero()` par `makeAppl(symzero)` avec `AFun symzero = makeAFun("Zero",0)` une constante définissant un symbole pour `Zero`. De même, le *mapping* doit définir comment obtenir le successeur d'un terme `t` donné. Cette construction peut être définie par `makeAppl(symsuc,t)`, avec `AFun symsuc = makeAFun("Suc",1)` une constante utilisée pour représenter le symbole de fonction `Suc`. Les objets alors produits par cette méthode

`makeAppl` sont de type `ATerm`, qui est utilisé pour représenter concrètement les termes algébriques.

Il reste à préciser les fonctions permettant d'examiner un terme existant, de le décomposer (extraire les sous-termes) et de tester l'égalité entre termes. La définition de ce mapping se fait par l'intermédiaire de constructions particulières **%typeterm**, **%op** et **%oplist**, qui seront présentées en section 2.3.1.

2.2.3. Filtrage associatif

Les constructions de filtrage de Tom, débutées par le lexème **%match**, sont légèrement différentes des constructions `match` des langages fonctionnels comme Caml, car alors que ceux-ci définissent une valeur comme membre droit, les membres droits des **%match** sont des instructions du langage hôte. De manière similaire à l'instruction `switch/case`, lorsqu'un filtre est trouvé, ces instructions sont exécutées, et si le flot de contrôle du programme n'est pas interrompu, les filtres suivants sont essayés. La définition de la fonction `plus` de l'exemple 4 utilise l'instruction Java `return` afin d'interrompre le flot de contrôle lorsqu'un filtre est trouvé.

Si le filtrage n'est pas unitaire — c'est-à-dire qu'il existe plusieurs filtres solutions du problème de filtrage — alors l'action associée au motif sera exécutée pour chaque filtre. Le filtrage syntaxique est unitaire, et donc ne permet pas d'expérimenter ce comportement. Le langage Tom possède une notion de *filtrage de liste*, aussi appelé *filtrage associatif avec élément neutre*, qui n'est pas unitaire, et permet d'exprimer simplement des algorithmes manipulant des listes. Les listes sont dans le cas de Tom représentées par un opérateur algébrique particulier, qui prend un nombre indéfini d'arguments. Si `conc` est un tel opérateur définissant une liste d'objets algébriques de sorte `Nat`, `conc()` désigne la liste vide d'entiers naturels, tandis que `conc(Zero())` dénote une liste à un seul élément, et `conc(Zero(),Zero(),Suc(Zero()))` une liste à trois éléments. La liste vide est considérée comme l'élément neutre des listes, et le système Tom introduit une distinction syntactique entre les variables de filtrage représentant des éléments, comme la variable `x` de l'exemple 4 et les variables représentant une sous-liste d'une liste existante, qui sont suivies de « `*` ».

Exemple 5. Le filtrage de liste permet d'itérer sur les éléments d'une liste lorsque l'action associée au motif utilisant ce filtrage de liste n'interrompt pas le flot de contrôle. Si on considère les listes d'entiers naturels comme des objets algébriques de sorte `List` implantées par le type Java `ATermList`, avec l'opérateur de liste `conc`, on peut définir l'affichage de tous les éléments d'une liste par :

```
public void affiche(ATermList l) {
  %match(List l) {
    conc(X1*,x,X2*) -> {
      System.out.println("En␣position␣:␣" + `X1*.length());
      System.out.println("Element␣:␣"+ `x);
    }
  }
```

```
}
```

L'action associée au motif défini dans cette construction **%match** est exécutée pour chaque filtre trouvé, en instanciant les variables de liste `X1*` et `X2*` par toutes les sous-listes préfixes et suffixes de la liste `l`. Notons que `X1*` correspond à un objet Java de type `ATermList`, ce qui permet d'utiliser les fonctions de cette bibliothèque. En particulier, la méthode `length()` qui retourne la longueur de la liste est utilisée pour connaître la position de l'élément `x` dans la liste.

Ainsi, cette fonction exécutée sur l'entrée `l = ‘conc(Zero(), Suc(Zero()), Zero(), Suc(Suc(Zero())))` produit en sortie :

```
En position : 0
Element : Zero()
En position : 1
Element : Suc(Zero())
En position : 2
Element : Zero()
En position : 3
Element : Suc(Suc(Zero()))
```

Le langage Tom ne spécifie pas l'ordre dans lequel les filtres sont énumérés, mais énumère les filtres un par un, sans se répéter[1]. On voit ici que la liste `X1*` a été successivement instanciée par des sous-listes à 0, 1, 2 puis 3 éléments.

Cette énumération se fait tant que le flot de contrôle de la construction **%match** n'est pas interrompu lors de l'exécution d'une action, à la manière de l'instruction `switch/case` qui peut être interrompue par une instruction `break` ou `return` terminant l'exécution de la fonction contenant cette instruction.

Exemple 6. Le filtrage de liste peut être utilisé pour implanter un tri sur les listes par filtrage de liste conditionnel. Si la même structure de données que dans l'exemple 5 est utilisée, et que l'on se donne une fonction de comparaison `greaterThan` sur les objets `ATerm`, on peut définir ce tri de manière récursive par :

```
public ATermList bubbleSort(ATermList l) {
  conc(X1*,x,y,X2*) -> {
    if(greaterThan(‘x,‘y)) {
      return ‘bubbleSort(conc(X1*,y,x,X2*));
    }
  }
  _ -> { return l; }
}
```

[1] Certains utilisateurs s'appuyant sur l'ordre d'évaluation actuellement utilisé par le compilateur, il est possible que cet ordre soit spécifié dans une version future

La règle conditionnelle est exécutée pour tous les filtres, c'est à dire pour tous les couples d'éléments x, y dans la liste argument, avec `X1*` et `X2*` les variables de listes capturant les contextes dans la liste. Le test `if(greaterThan(‘x,‘y))`, qui est une instruction Java, est exécuté pour chacun de ces couples ; lorsque la fonction trouve un couple tel que x est plus grand que y, alors ces deux éléments sont échangés et la fonction est rappelée récursivement sur le résultat.

S'il n'existe pas de tel couple, alors tous les filtres pour le premier motif sont éliminés, et la règle suivante est essayée, car il n'y a pas d'interruption du flot de contrôle du programme. Cette seconde règle, utilisant la notation `_`, filtre pour toute entrée, et la liste originale est retournée, correspondant au cas de base de la définition récursive car il n'existe pas de couple x, y avec x plus grand que y, et donc la liste est triée.

Les constructions de filtrage de Tom offrent 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 pour obtenir un filtre que les sous-termes correspondant aux instances de ces variables soient égaux. 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 7. En utilisant la définition de structure de données de l'exemple 5, on peut définir une fonction éliminant les doublons dans une liste par filtrage de liste non-linéaire, sur le modèle de l'algorithme de tri de l'exemple 6 :

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

Exemple 8. On peut aussi utiliser la non-linéarité lorsque la construction de filtrage a plusieurs sujets (comme celle qui définit la fonction `plus` de l'exemple 4), pour retrouver des sous-termes partagés par ces sujets. On peut ainsi définir la recherche d'un élément `e` dans une liste `l`, en utilisant toujours la structure de données de l'exemple 5, retournant `true` si l'élément est présent dans la liste, et `false` sinon :

```
public boolean search(ATerm e, ATermList l) {
  %match(Nat e, List l) {
    x, conc(_*,x,_*) -> {
      return true;
    }
  }
```

```
  return false;
}
```

La notation `_*` dans cet exemple indique une variable de liste non nommée. On ne peut alors pas utiliser sa valeur dans le membre droit de la règle. Les deux occurrences de `_*` désignent des variables non-nommées différentes, ce qui fait que le second motif itère sur les éléments de la liste `l`. L'action associée n'est cependant exécutée que lorsque le motif entier filtre, c'est à dire lorsque l'élément `x`, ayant pour valeur `e` par filtrage est retrouvé dans la liste `l`.

2.2.4. Notations utiles

Nous avons présenté jusque là les constructions essentielles de filtrage de Tom. Ces constructions permettent d'exprimer des algorithmes complexes de manière algébrique, et souvent élégante. Afin de permettre une plus grande concision, ainsi que pour aider à la lisibilité des programmes complexes et aider à leur évolution, le langage comporte un certain nombre d'extensions. Nous allons ici présenter les extensions du langage les plus importantes, car elles seront utiles par la suite.

Le *Backquote* « ‘ »

La construction « ‘ » permet de signaler la construction d'un terme algébrique par Tom. Les variables introduites par filtrage sont accessibles uniquement lorsqu'elles sont utilisées dans de telles construction. Nous avons déjà rencontré divers aperçus d'utilisation de cette construction dans les exemples 4, 6 et 7. Ainsi, pour le type algébrique `Nat` utilisé dans ces exemples, il est possible de construire des objets représentant les termes de ce type algébrique en utilisant cette construction. L'instruction

```
ATerm trois = ‘Suc(Suc(Suc(Zero())));
```

déclare une variable `trois` dont le type est `ATerm`, qui est le type des représentants concrets des termes algébriques, comme ayant pour valeur le représentant du terme algébrique $Suc(Suc(Suc(Zero())))$.

Cela permet à l'utilisateur de ne pas avoir à connaître les détails de l'implémentation concrète de ce type des termes. On utilise généralement un nom de type concret égal au nom de la sorte, afin de simplifier l'écriture des programmes. Les termes algébriques sont manipulés dans le programme de manière algébrique lors du filtrage et lors de la construction.

Cette construction « ‘ » permet aussi de considérer des fonctions manipulant les termes algébriques à la manière d'opérateurs algébriques. Ainsi, dans les exemples 4 et 7, les fonctions respectivement `plus` et `removeDouble` sont intégrées à la construction *backquote*, qui lors de son évaluation appellera la fonction et utilisera le résultat pour construire un objet `ATerm`.

Les alias de sous-termes dans les motifs

Un premier élément ajoutant de l'expressivité aux motifs de Tom est la possibilité d'attribuer un nom aux sous-termes obtenus par filtrage. Cela permet de simultanément vérifier qu'un certain motif filtre un sous-terme, et de donner à une variable une valeur égale à ce sous-terme si le motif filtre. La notation « `@` » permet de définir de tels alias.

Exemple 9. On peut en utilisant la structure de liste d'entiers naturels de l'exemple 5 décider de n'afficher que les nombres qui dans une liste sont supérieurs ou égaux à 2, en vérifiant cette condition par filtrage, les nombres supérieurs à deux pouvant être filtrés par le motif $Suc(Suc(\_))$.

```
public void afficheSup2(ATermList l) {
  %match(List l) {
    conc(X1*,x@Suc(Suc(_)),X2*) -> {
      System.out.println(‘x + "␣>=␣2");
    }
  }
}
```

L'action associée au motif n'est exécutée que pour les éléments de la liste pouvant être filtrés par le motif $Suc(Suc(\_))$. La valeur de ces éléments est alors donnée à la variable x définie par l'alias.

Cette notation est importante, car il est souvent possible en l'utilisant d'éviter l'emploi de constructions **%match** imbriquées.

La notation « crochets »

Les signatures algébriques définies pour les opérateurs pour Tom ont la particularité de donner un nom à chaque argument des opérateurs. Ainsi, à la manière des structures du langage C, chaque argument constituant un opérateur possède un nom. Il est alors possible d'utiliser ce nom dans un motif pour désigner un sous-terme particulier ainsi qu'omettre de traiter certains sous-termes, qui sont alors ignorés lors du filtrage, en utilisant une notation dite « crochets » dans les motifs, matérialisée par les lexèmes [et].

Exemple 10. Si on définit les entrées d'un carnet d'adresses comme les termes construits en utilisant l'opérateur *Entree* d'arité *3* et de type *Entree*, ayant des arguments nommés *nom*, *adresse* et *telephone*, tous de type « chaîne de caractères », les fonctions suivantes sont équivalentes :

```
public void afficheNoms(Entree e) {
  %match(Entree e) {
    Entree(x,_,_) -> {
      System.out.println(‘x);
    }
```

```
  }
}
public void afficheNomsBis(Entree e) {
  %match(Entree e) {
    Entree[nom=x] -> {
      System.out.println(‘x);
    }
  }
}
```

La seconde écriture a certains avantages sur la seconde. Tout d'abord, le fait que l'on s'intéresse au *nom* dans cette fonction est explicite au niveau du motif, ce qui rend la relecture plus facile. Ensuite, cette seconde écriture est plus robuste au changement : si la signature est changée, par exemple pour ajouter un champ *fax* aux entrées du carnet, la seconde fonction ne devra pas être modifiée, alors que la première devra contenir un _ de plus. Lors d'une mise à jour de la structure de donnée, seuls les motifs manipulant explicitement les champs de la structure qui ont été changés doivent être modifiés, rendant le programme plus facile à faire évoluer.

2.3. Structures de données

Nous avons vu en section 2.2 que les constructions de filtrage que Tom apporte à ces langages sont indépendantes de la structure de données qui permet de représenter les termes algébriques. Il est alors nécessaire de donner au compilateur une définition de la structure de données, précisant la structure algébrique ainsi que la manière dont cette structure est implémentée dans le langage hôte, comme nous l'avons introduit en section 2.2.2.

2.3.1. Définition des *mappings*

Ces *mappings* sont définis en utilisant les constructions **%typeterm**, **%op** et **%oplist** permettant de décrire respectivement l'implémentation des sortes, les différents opérateurs algébriques et les opérateurs de liste.

Nous allons illustrer ces constructions par la définition du *mapping* entre le type algébrique `Nat` et son implémentation par des objets crées en utilisant la bibliothèque ATerm, comme esquissé en section 2.2.2.

Tout d'abord, la construction **%typeterm** permet de déclarer une sorte algébrique, ainsi que de spécifier le type des objets concrets qui représentent les termes de cette sorte. D'autre part, cette construction impose de spécifier le test d'égalité de deux termes de la sorte lorsque l'on a deux représentants concrets.

```
%typeterm Nat {
  implement    { aterm.ATermAppl }
  equals(t1, t2) { t1 == t2}
```

```
}
```

La sorte `Nat` est déclarée par la construction **%typeterm** comme implémentée par le type `ATermAppl`, et il est spécifié qu'il est suffisant de comparer les valeurs des pointeurs de deux objets pour savoir si les termes qu'ils représentent sont égaux. Cependant, un autre *mapping* pourrait faire usage d'une fonction de comparaison définie par l'utilisateur.

Les opérateurs de cette sorte sont alors définis en utilisant la construction **%op**. On utilise pour cette description les constantes déjà présentées en section 2.2.2 `symzero` et `symsuc` correspondant aux symboles de fonction *Zero* et *Suc*, ainsi qu'une variable `factory` servant de point d'accès pour la création des différents objets.

```
ATermFactory factory = new aterm.pure.PureFactory();
AFun symzero = factory.makeAFun("Zero",0);
AFun symsuc = factory.makeAFun("Suc",1);

%op Nat Zero() {
  is_fsym(t) { t.getAFun() == symzero }
  make { factory.makeAppl(symzero) }
}

%op Nat Suc(pred:Nat) {
  is_fsym(t) { t.getAFun() == symsuc }
  get_slot(pred,t) { t.getArgument(0) }
  make(t) { factory.makeAppl(symsuc,t) }
}
```

Ces définitions d'opérateurs définissent comment on doit créer un objet représentant un terme algébrique, ici par un appel à la fonction `makeAppl`. Elles définissent aussi par la construction **is_fsym** la manière de tester si un objet donné représente bien un terme dont le symbole de tête est l'opérateur, et la manière d'extraire les différents sous-termes en fonction de leur nom par la construction **get_slot**. La première ligne de chaque construction **%op** définit la signature de l'opérateur algébrique, ainsi que les noms des éventuels sous-termes.

Les opérateurs de liste sont définis de manière similaire en utilisant la construction **%oplist**. La première ligne spécifie le domaine et le co-domaine de l'opérateur de liste, ainsi que son nom. Les opérations spécifiques aux listes doivent être définies, ce sont elles qui permettent de compiler le filtrage de liste. Le *mapping* précise ainsi comment construire une liste vide, et insérer un nouvel élément en tête d'une liste. Il détaille aussi la manière de déconstruire une liste, en extrayant l'élément de tête ainsi que le reste de la liste.

```
%typeterm List {
  implement { aterm.ATermList }
  equals(l1,l2) { l1.equals(l2) }
```

```
}

%oplist List conc( Nat* ) {
  is_fsym(t)     { t instanceof ATermList }
  get_head(l)    { l.getFirst() }
  get_tail(l)    { l.getNext() }
  is_empty(l)    { l.isEmpty() }
  make_empty() { factory.makeList() }
  make_insert(e,l) { l.insert((ATerm)e) }
}
```

La définition de la définition **%oplist** pour la liste de naturels implémentée comme une liste `ATermList` est facile, le type `ATermList` fournissant une interface proche des besoins de Tom. La bibliothèque du compilateur Tom fournit des *mapping* pour les implantations standard des différentes listes de la bibliothèque Java, qui peuvent aussi être réutilisées.

2.3.2. Contraintes sur les structures de données pour Tom

Il est possible de définir un *mapping* pour une structure de données arbitraire du langage hôte. Cependant, cette structure de données et le *mapping* associé doivent vérifier un certain nombre de propriétés afin d'implanter correctement le filtrage.

Une contrainte générale est que le type de données abstrait doit être représenté de manière fidèle par l'implémentation concrète. Cela signifie que les opérations définies sur les données concrètes doivent correspondre à leurs équivalents sur les données abstraites. Ainsi, l'opération de test d'égalité spécifiée par `equals` dans chaque définition de sorte doit correspondre au test d'égalité des termes. Nous avons choisi dans notre exemple d'implémenter ce test d'égalité par une comparaison de pointeurs, mais ce choix n'est valide qu'en raison des particularités de la structure concrète qui assure un partage maximal en mémoire. Une structure de données différente devrait fournir un test d'égalité compatible avec l'égalité sur les termes algébriques.

Le prédicat défini par `is_fsym` doit permettre de tester si un terme possède bien un certain symbole de tête. L'utilisation d'un prédicat plutôt que d'une fonction permettant de récupérer la valeur du symbole de tête d'un objet représentant un terme et d'une fonction de test d'égalité sur les symboles de tête permet d'éviter d'obliger l'utilisateur à manipuler des représentants de symboles de tête. Nos définitions des symboles *Zero* et *Suc* utilisent une représentation des symboles de tête dans le programme, par l'intermédiaire des objets constants `symzero` et `symsuc`, mais, comme nous le verrons au chapitre 4, il est possible d'utiliser des classes pour représenter les différents opérateurs, **is_fsym** étant alors implémenté par `instanceof`.

Les fonctions `get_slot`, `get_head` et `get_tail` doivent quant à elles permettre de déconstruire un terme, et d'obtenir ses sous-termes. Il faut que l'utilisation de la spécification de la construction par `make` avec les différents sous-termes obtenus par `get_slot` produise un objet égal (selon le test spécifié par `equals`) à l'objet d'origine.

Il en est de même avec les listes, qui sont déconstruites par les fonctions spécifiées

par `get_head` et `get_tail`. L'opération spécifiée par `make_empty` doit construire une liste vide, tandis que celle qui est spécifiée par `make_insert(e,l)` retourne une nouvelle liste l' correspondant à la liste l dans laquelle l'élément e a été inséré en tête. Ainsi, des expressions comme `equals(get_head(l'),e)` et `equals(get_tail(l'),l)` doivent être vraies.

Cette notion informelle de validité des *mappings* sera discutée et formalisée en section 3.3.1, car elle est l'une des bases sur lesquelles se fonde la certification de la compilation du filtrage par Tom.

2.3.3. Générateurs de structures de données

La définition d'un tel *mapping* n'est pas difficile lorsque la structure de données utilisée est conçue pour représenter des termes algébriques. Ainsi, on a vu que pour la définition de la sorte algébrique `Nat`, ainsi que pour les listes d'entiers, l'utilisation de la bibliothèque ATerm [BJKO00] permet une description simple du lien entre implémentation et structure algébrique. Cette bibliothèque, disponible pour les langages C et Java, fournit une implantation générique et efficace des arbres, qui implémente le partage maximal des sous-termes et fournit aussi gestion de la mémoire et ramasse-miettes dans la version C, facilitant ainsi la programmation avec cette bibliothèque [MZ04].

Cependant, cette bibliothèque ne donne aucune garantie de correction du typage des termes construits : il est possible de construire des termes n'appartenant pas au langage manipulé dans le programme. Les parties Tom d'un programme effectuent une vérification de type pour les types algébriques manipulés, mais cette information est perdue dans les sections Java ou C des programmes, pouvant conduire à des erreurs difficiles à corriger.

Il est cependant possible de générer à partir d'une signature algébrique donnée un ensemble de classes Java ou de fichiers source C implantant cette signature, et fournissant un *mapping* pour cette signature. L'outil ApiGen [BMV05] est un tel générateur, qui fournit une implantation typée de la signature qui lui est passée en argument. Une signature est décrite dans un fichier listant les différents opérateurs, leur sorte ainsi que leurs arguments. Cette description est compilée en une implémentation pour le langage hôte, Java ou C ainsi qu'un *mapping* permettant à l'utilisateur d'utiliser cette structure dans des programmes Tom. La description de la sorte `Nat` ainsi que de la liste d'entiers naturels pourrait alors être :

```
[
  constructor(Nat,Zero,Zero()),
  constructor(Nat,Suc,Suc(<pred(Nat)>)),
  named-list(conc,List,Nat)
]
```

L'implantation typée générée pour la description, basée sur la bibliothèque ATerm, possède les mêmes avantages, mais permet de garantir une utilisation correcte de la signature dans les sections de langage hôte des programmes. D'autre part, les types

concrets générés ont le même nom que leurs équivalents algébriques, ce qui rend les programmes plus lisibles. C'est particulièrement le cas lorsqu'un programme doit manipuler une signature très étendue, comme pour le compilateur Tom lui-même (car c'est un programme Tom). Il est alors utile de pouvoir distinguer par leur type une variable contenant une expression d'une variable contenant une instruction.

Cette méthode est pourtant peu usuelle, les langages utilisant des structures algébriques ou des types inductifs avec une notion de type étant généralement implémentés en utilisant une structure d'arbres génériques. Les types sont alors perdus lors de l'exécution afin d'une part de simplifier l'implantation, et d'autre part de permettre l'utilisation d'algorithmes génériques. Dans le cas des structures générées par ApiGen, il est toujours possible d'utiliser la structure non-typée sous-jacente aux données pour utiliser des algorithmes génériques, comme des parcours d'arbres.

2.4. Des problèmes à résoudre

La langage ici décrit est particulièrement utile pour spécifier des transformations de données par filtrage et réécriture, tout en conservant la souplesse d'utilisation d'un langage généraliste comme Java ou C. En particulier, il reste possible d'utiliser toutes les ressources du langage hôte pour implémenter les aspects qui ne traitent pas de la transformation elle même, comme les entrées sorties, les problèmes de persistance ou la modularité du programme.

La possibilité d'utiliser des structures de données arbitraires permet par exemple d'implémenter des transformations de documents XML de manière algébrique. L'utilisation de Tom peut ainsi se faire au sein d'un projet sans modifier les structures de données utilisées, ni impacter les autres modules du projet.

Cependant, le langage et ses utilisations évoluant, certains problèmes apparaissent, qui soit limitent l'utilité du langage, soit rendent l'implémentation de certains programmes difficile, soit limitent la confiance que l'on peut avoir vis-à-vis de ces programmes.

2.4.1. Correction du code généré

Une première question qui se pose lorsque l'on doit utiliser un langage tel que Tom est : « est-ce que le programme généré fait bien ce que j'attends ? ». En effet, le compilateur Tom n'est pas comme le sont les compilateurs ML, C ou Java utilisé quotidiennement par des milliers d'entreprises très sérieuses, alors peut-on avoir confiance dans ce compilateur, et est-ce que je ne ferai pas mieux d'écrire manuellement tout ce code de filtrage ?

La réponse d'un utilisateur de Tom serait bien sûr « non », car il est bien plus efficace d'utiliser les constructions de filtrage de Tom, cela permet de passer moins de temps à implémenter et limite les risques d'erreurs. De plus, l'utilisation de ces constructions amène naturellement à exprimer les algorithmes de manière plus fonctionnelle et plus sûre.

Notre premier souci sera alors de donner une garantie forte du fait que le compilateur Tom compile effectivement les constructions de filtrage de manière correcte. Ceci nous conduira aussi à formaliser ce qu'est un *mapping* correct pour un type algébrique.

2.4.2. Les invariants des structures de données

Nous avons vu qu'il est possible d'utiliser des structures de données arbitraires. Cependant, la plupart des applications qui n'imposent pas d'utiliser une structure de données pré-existante vont utiliser une structure générée à partir d'une description de la signature algébrique.

Des outils, comme le générateur ApiGen existent pour générer une telle implémentation. Cependant, de nombreuses applications non seulement utilisent des types algébriques, mais aussi une notion de classes d'équivalence sur ces données. Ainsi, un compilateur considérera des classes d'équivalence modulo propagation et évaluation des constantes, un solveur de contraintes modulo arithmétique booléenne, et un module de calcul formel modulo la distributivité de diverses opérations. Ceci introduit une notion d'*invariants de structure de données*, lesquels doivent être préservés.

Le langage Tom permet de facilement écrire des fonctions établissant ces invariants, mais il est toujours à la charge du programmeur de bien maintenir les invariants et d'appeler ces fonctions.

2.4.3. Traversées génériques des arbres

Le langage permet d'exprimer le filtrage, mais la plupart des algorithmes exprimés par des règles de réécriture ou du filtrage nécessitent de pouvoir parcourir les termes en profondeur. Le langage ne fournit aucun support pour de telles traversées, et c'est au programmeur de spécifier ces traversées d'arbres sur la structure de données concrète.

Dans le cas de l'implémentation du compilateur Tom (avant l'introduction des stratégies présentées au chapitre 6), un ensemble de fonctions de traversée génériques, inspirées des fonctions de traversée de ASF+SDF [BKV03] sont fournies, qui utilisent la représentation non-typée sous forme de `ATerm` des structures générées par ApiGen. Ces fonctions étant non-typées, une certaine discipline est nécessaire pour garantir que le programme manipule toujours des données valides. Un langage de stratégies plus élaboré permettrait de spécifier ces transformations de manière algébrique, pour écrire des transformations plus sûres.

Ce sont ces différents points qui nous serviront de point de départ et de motivation pour développer les contributions présentées dans les chapitres suivants.

3. Certification du filtrage

> PROOF, n. Evidence having a shade more of plausibility than of unlikelihood. The testimony of two credible witnesses as opposed to that of only one.
>
> *(Ambrose Bierce)*

3.1. Certifier la compilation

La question de la correction des compilateurs, c'est à dire de s'assurer que le compilateur préserve les propriétés de son entrée est présente depuis l'écriture des premiers compilateurs.

Les propriétés que le compilateur doit préserver impérativement peuvent être très fortes, comme préserver la sémantique du programme d'entrée, ou des propriétés de plus haut niveau, moins fortes, sur l'algorithme qui constitue son entrée, comme sa terminaison ou le respect de certains invariants dans la mémoire.

3.1.1. Compilateur certifié

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].

Néanmoins, il est encore hors de portée de prouver qu'un programme à été compilé de manière entièrement correcte, et encore plus qu'un compilateur compilera tous les programmes valides de son langage d'entrée de manière correcte. Dans la pratique, les programmeurs, et par conséquent les applications qu'ils écrivent, se reposent entièrement sur le compilateur : avant que l'on exécute effectivement le programme, on n'a aucune indication que le compilateur a compilé le programme correctement. Et bien sûr, une procédure de tests intensive de l'application compilée n'offre pas une telle garantie, mais seulement celle que le programme se comporte comme on l'attendait pour les cas d'usage du jeu de test. Ainsi, le programmeur généralement porte une confiance aveugle au compilateur qu'il utilise. Heureusement, les compilateurs C ou Fortran les plus couramment utilisés ne génèrent du code incorrect que très sporadiquement, et sont parmi les systèmes logiciels les plus fiables. Cette fiabilité est due au grand nombre de développeurs travaillant à rendre (ou maintenir) ces compilateurs corrects autant qu'au très grand nombre d'utilisateurs de ces systèmes, qui contribuent, de fait, à ce débuggage.

Cependant, lorsque l'on doit concevoir ou développer un compilateur pour un nouveau langage de haut niveau, la situation est moins confortable. D'une part, le nombre d'utilisateurs du langage ainsi que les applications l'utilisant est petit (et souvent proche de 1 au début de la vie du langage), et d'autre part, l'introduction de nouvelles constructions de haut niveau rend le compilateur difficile à écrire. Comme les conséquences des problèmes de correction d'un compilateur peuvent être désastreuses — rendant tous les programmes potentiellement faux — cette situation contribue à rendre les utilisateurs méfiants quand ils sont confrontés à de nouveaux langages et implémentations de compilateurs.

Cela rend la vérification du compilateur non seulement intéressante, mais aussi très importante afin d'augmenter le niveau de confiance que les utilisateurs peuvent avoir dans le compilateur, niveau de confiance qui ne peut que difficilement être obtenu par le nombre d'utilisateurs et de testeurs.

L'approche consistant à essayer de prouver par des méthodes formelles que le compilateur lui-même est correct, c'est-à-dire compile correctement tout programme d'entrée, ou un sous-ensemble des programmes d'entrée (par exemple, en ne considérant que des programmes syntaxiquement valides, ou valides et bien typés), bien que séduisante, car elle offre un compilateur dont on sait avant de l'utiliser qu'il va fonctionner comme prévu, n'est pas vraiment applicable dans le cas général. Toutefois, des travaux récents ont montré qu'il est possible de certifier un compilateur (ou des parties de celui-ci) pour un langage proche de C en utilisant un assistant de preuve [Ler06, BDL06], ainsi que d'extraire automatiquement une implantation certifiée du compilateur à partir de ces preuves.

En particulier, dans les cas d'un langage comme Tom, qui est amené à évoluer au cours du temps, cette méthode introduirait une certaine inertie, augmentant la résistance au changement, car chaque changement dans le langage lui-même impose de revoir toutes les preuves, et d'actualiser non seulement le compilateur lui-même, ce qui peut représenter beaucoup de travail, mais aussi les preuves de correction de ce compilateur. D'autre part, le compilateur lui-même étant un programme, il est fréquent et même souhaitable de modifier son implantation, soit pour corriger un problème particulier, soit pour faire évoluer le programme, le rendre plus lisible, ou plus modulaire, plus efficace. La nécessité de faire évoluer les preuves en même temps que le programme peut permettre d'éviter l'introduction de nouvelles erreurs, et ainsi d'obtenir un système d'une qualité irréprochable, mais elle risque aussi de bloquer les développements et les évolutions, le coût de chaque évolution devenant trop important pour les développeurs. Le compilateur peut aussi être automatiquement extrait de la preuve en utilisant un assistant de preuve comme Coq [BDL06], mais même ainsi, les preuves doivent encore évoluer.

Cette solution n'est donc pas actuellement envisageable pour un langage et un compilateur évoluant rapidement, comme l'est Tom.

3.1.2. Validation

Nous nous intéresserons donc à une autre approche 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 traiter deux types de problèmes :

- d'une part, la présence non intentionnelle de bogues dans le compilateur ;
- d'autre part, elle peut permettre de détecter un comportement non voulu par le programmeur, mais introduit malicieusement par le compilateur.

Dans cette approche, pour un programme dans le langage de haut niveau d'entrée du compilateur, on considère alors ce compilateur comme une boîte noire sur laquelle on n'a aucun contrôle, et on examine le code produit par cette boîte noire, essayant de prouver que le code produit est correct vis-à-vis du programme d'entrée. Cette idée n'est pas nouvelle, et est typique des travaux précurseurs de [BY92], et plus récemment de la méthode *« translation validation »* [PSS98, NL98]. Une approche sensiblement comparable et appelée *« credible compilation »* à été présentée dans [RM99], laquelle permet de gérer l'utilisation de pointeurs dans le programme source.

On peut remarquer 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.

3.2. Particularités de l'environnement : îlot formel

Nous nous intéressons dans ce chapitre à montrer la correction du code généré par le compilateur Tom, et plus particulièrement à la compilation du filtrage par ce compilateur. Comme nous l'avons présenté dans le chapitre 2, le langage Tom est intimement lié et imbriqué dans son langage *hôte*, que celui-ci soit Java, C ou Caml. Ainsi, le filtrage que le compilateur Tom compile s'effectuera lors de l'exécution du programme sur les valeurs et des objets issus de ce langage hôte.

Il sera alors nécessaire pour pouvoir vérifier le code généré par le compilateur de se donner les moyens de raisonner avec des structures et objets venant de ces langages hôtes, ainsi que de modéliser le mécanisme des *ancrages*, qui définissent comment Tom va manipuler les objets du langage hôte.

Un tel cadre a été développé spécifiquement dans le but de vérifier la compilation du filtrage de Tom, et a ensuite pu être généralisé pour d'autres langages « à îlots », comme SqlJ (intégration de SQL dans Java) [BKM06].

Nous allons tout d'abord présenter brièvement ce cadre généralisé, ainsi que ses caractéristiques, avant de présenter une de ses instances, et de l'utiliser pour vérifier la compilation de Tom.

3.2.1. Un cadre plus large

L'élaboration du langage Tom à conduit à le décrire comme un langage à *îlots*. Ces îlots sont les parties de code Tom écrits à l'intérieur d'un programme du langage qui

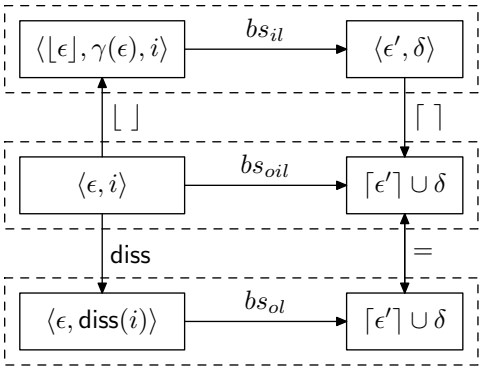


FIG. 3.1: Composition des sémantiques *îlot* et *océan*

« héberge » ces îlots : le langage hôte.

Lorsque l'on décrit la sémantique du langage des îlots, celle-ci dépend de la manière dont le langage des îlots perçoit les données du langage hôte. Ainsi, on ne peut pas simplement décrire les sémantiques de ces deux langages indépendamment, mais il faut établir un lien entre les deux. C'est ce que décrit la notion d'ancrage formel que l'on définit dans la section 3.3.1. Cette notion a été étendue à la notion d'*îlot formel*, qui généralise le travail réalisé pour la certification de la compilation du filtrage par Tom. La formalisation des *îlots formels* décrite dans [BKM06] permet ainsi de décrire la combinaison d'un langage *îlot* et d'un langage *océan*, au niveau de leur grammaire, mais aussi au niveau des objets qu'ils manipulent tous les deux et de la sémantique à donner aux programmes écrits dans le langage combiné.

En particulier, cette formalisation généralise la notion d'ancrage formel présentée dans la définition 15 à une notion d'ancrage paramétré par des prédicats (alors que l'on ne considère ici que les prédicats définissant une structure de termes). D'autre part, cette définition permet aussi de décrire la sémantique de la composition des langages *îlot* et *océan* ainsi que la définition d'une fonction de *dissolution*, ou compilation qui transforme le programme composé en un programme exclusivement écrit dans le langage *océan*.

Le diagramme de la figure 3.1 montre comment la sémantique de la composition des langages *îlot* et *océan*, notée bs_{oil} est obtenue en étendant la sémantique *océan*, bs_{ol} avec la sémantique des îlots bs_{il}, faisant correspondre les objets manipulés par les deux langages au moyen de fonctions $\lceil\ \rceil$ et $\lfloor\ \rfloor$. La composition est alors « dissolue » dans le langage *océan*, et la sémantique du programme après dissolution pour bs_{ol} est montrée cohérente avec celle du programme composé avec la sémantique composée bs_{oil}.

Cette technique pourrait permettre de valider le processus de compilation du filtrage du compilateur Tom de manière globale, comme décrit dans la section 3.1.1. Cependant, cela n'est pas compatible avec les impératifs d'évolutivité que l'on s'est fixés.

Le fait qu'une définition de la notion d'*îlots formels* soit proche du cadre mis en place pour la validation de la compilation du filtrage de Tom montre que cette approche pourrait être adaptée à d'autres langages impliquant des interactions fortes entre les objets manipulés par deux langages imbriqués.

3.3. Certifier le filtrage

On considérera ici une instance du cadre des îlots formels, qui est adaptée au problème de la validité du filtrage dans Tom. Pour traiter ce problème, on considérera 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 dans le langage de sortie du compilateur `PIL`, et qui sont supposés résoudre des problèmes de filtrage, ainsi que les données qui représentent des termes clos. Ce langage, décrit en section 3.3.3, est un langage impératif simple correspondant au sous ensemble commun des langages hôtes utile pour décrire les procédures de filtrage.

3.3.1. Ancrage formel

3.3.2. Fonction de représentation

Définition 14 (Représentation)**.** *Étant donnés un tuple $\langle \mathcal{F}, \mathcal{X}, \mathcal{T}(\mathcal{F}), \mathbb{B}, \mathbb{N} \rangle$, composé d'une signature $\mathcal{F}$, d'un ensemble de variables $\mathcal{X}$, des booléens $\mathbb{B}$ et des entiers $\mathbb{N}$, ainsi que des ensembles $\Omega_{\mathcal{F}}$, $\Omega_{\mathcal{X}}$, $\Omega_{\mathcal{T}}$, $\Omega_{\mathbb{B}}$, et $\Omega_{\mathbb{N}}$, on considère une famille de fonctions de* représentation $\ulcorner\urcorner$ *qui lie :*

- *les symboles de fonction $f \in \mathcal{F}$ aux éléments de $\Omega_{\mathcal{F}}$, notés $\ulcorner f \urcorner$,*
- *les variables $v \in \mathcal{X}$ aux éléments de $\Omega_{\mathcal{X}}$, notées $\ulcorner v \urcorner$,*
- *les termes clos $t \in \mathcal{T}(\mathcal{F})$ aux éléments de $\Omega_{\mathcal{T}}$, notés $\ulcorner t \urcorner$,*
- *les booléens $b \in \mathbb{B} = \{\top, \bot\}$ aux éléments de $\Omega_{\mathbb{B}}$, notés $\ulcorner b \urcorner$,*
- *les nombres naturels $n \in \mathbb{N}$ aux éléments de $\Omega_{\mathbb{N}}$, notés $\ulcorner n \urcorner$.*

Cette fonction de représentation fait le lien entre les entités algébriques ($\mathcal{F}$, $\mathcal{X}$, $\mathcal{T}(\mathcal{F})$, $\mathbb{B}$ et $\mathbb{N}$) et les objets que le langage intermédiaire `PIL` manipule, qui sont éléments de $\Omega_{\mathcal{F}}$, $\Omega_{\mathcal{X}}$, $\Omega_{\mathcal{T}}$, $\Omega_{\mathbb{B}}$ et $\Omega_{\mathbb{N}}$.

On note $\ulcorner \mathcal{T}(\mathcal{F}) \urcorner$ l'ensemble contenant les représentants des termes, défini comme $\ulcorner \mathcal{T}(\mathcal{F}) \urcorner = \{\ulcorner t \urcorner | t \in \mathcal{T}(\mathcal{F})\}$, et ainsi, on a $\ulcorner \mathcal{T}(\mathcal{F}) \urcorner \subseteq \Omega_{\mathcal{T}}$, car tous les termes clos sont représentables et manipulables par `PIL`, mais les objets manipulables par le langage `PIL` ne sont pas nécessairement tous des représentants de termes clos.

Exemple 11. Soit $\mathcal{F} = \{e, s\}$, avec $ar(e) = 0$ et $ar(s) = 1$, et la fonction $\ulcorner\urcorner$ telle que $\ulcorner e \urcorner = 0 \in \Omega_{\mathcal{F}}$, $\ulcorner s \urcorner = 1 \in \Omega_{\mathcal{F}}$. $\ulcorner\urcorner$ lie les symboles e et s respectivement aux « entiers machine » 0 et 1 (c'est à dire à la notion d'entiers dans le langage intermédiaire), où on suppose une mémoire infinie. De la même manière, la fonction $\ulcorner\urcorner$ peut être étendue pour lier la constante $e \in \mathcal{T}(\mathcal{F})$ à 0 ($\ulcorner e \urcorner = 0 \in \Omega_{\mathcal{T}}$), et tout terme de la forme $s(x)$ au résultat de l'addition de 1 à la représentation de x ($\ulcorner s(x) \urcorner = 1 + \ulcorner x \urcorner \in \Omega_{\mathcal{T}}$).

Cette fonction de représentation est un moyen de projeter les entiers de Peano sur les « entiers machine ».

Dans la définition 14, la notion de fonction de représentation a été introduite pour établir une correspondance entre les objets algébriques et leurs représentants dans le langage intermédiaire. Cependant, nous n'avons pour le moment placé aucune contrainte sur les représentants des objets. En particulier, la fonction $\ulcorner\urcorner$ ne préserve pas nécessairement les propriétés structurelles des objets algébriques. Par exemple, on peut imaginer de représenter tous les termes par une unique constante. La notion de « différence » entre deux termes ne serait alors pas préservée. On doit alors introduire des contraintes sur la définition des fonctions de représentation afin de pouvoir les utiliser en pratique.

Définition 15 (Ancrage formel)**.** *Étant donné un tuple* $\langle \mathcal{F}, \mathcal{X}, \mathcal{T}(\mathcal{F}), \mathbb{B}, \mathbb{N} \rangle$*, une fonction de représentation* $\ulcorner\urcorner$ *et les injections :*

- $\texttt{eq} : \Omega_{\mathcal{T}} \times \Omega_{\mathcal{T}} \to \Omega_{\mathbb{B}}$*,*
- $\texttt{is\_fsym} : \Omega_{\mathcal{T}} \times \Omega_{\mathcal{F}} \to \Omega_{\mathbb{B}}$ *et*
- $\texttt{subterm}_f : \Omega_{\mathcal{T}} \times \Omega_{\mathbb{N}} \to \Omega_{\mathcal{T}}$ *$(f \in \mathcal{F})$*

Un ancrage formel *est une application* $\lceil\ \rceil : \mathcal{T}(\mathcal{F}) \to \ulcorner \mathcal{T}(\mathcal{F}) \urcorner$ *telle que les propriétés structurelles de* $\mathcal{T}(\mathcal{F})$ *soient préservées, dans* $\ulcorner \mathcal{T}(\mathcal{F}) \urcorner$ *par la sémantique de* eq*,* is_fsym *et* $\texttt{subterm}_f$*.*

Ceci signifie que l'on veut que les fonctions dans le langage PIL *qui permettent d'explorer la structure d'un représentant de terme,* eq *permettant de tester l'égalité,* is_fsym *d'examiner le symbole de tête, et* $\texttt{subterm}_f$ *d'extraire les sous-termes, correspondent bien à ces opérations au niveau algébrique.*

$\forall t, t_1, t_2 \in \mathcal{T}(\mathcal{F}), \forall f \in \mathcal{F}, \forall i \in [1..ar(f)]$ *on a :*

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

On peut remarquer que cette application $\lceil\ \rceil$ doit être étendue aux booléens, de sorte que : $\lceil \top \rceil \mapsto \texttt{true}$ et $\lceil \bot \rceil \mapsto \texttt{false}$.

Dans la suite, on considérera toujours qu'une fonction de représentation est un ancrage formel. Ainsi la notation $\ulcorner\urcorner$ notera les fonctions de représentations qui sont aussi des ancrage formels.

Exemple 12. Dans les langages C ou Java, la notion de terme peut être implantée par une structure, ou *record* `(sym :integer, sub :array of term)` dans laquelle le premier champ (`sym`) correspond au symbole de tête, et le second champ (`sub`) aux sous-termes. Il est facile de vérifier que les définitions suivantes de `eq`, `is_fsym`, et $\texttt{subterm}_f$ (où $=$ note l'égalité atomique) définissent un *ancrage formel* pour $\mathcal{T}(\mathcal{F})$:

$$\begin{aligned} \texttt{eq}(\texttt{t}_1, \texttt{t}_2) &\triangleq \texttt{t}_1\texttt{.sym} = \texttt{t}_2\texttt{.sym} \wedge \forall i \in [1..ar(\texttt{t}_1\texttt{.sym})], \\ &\quad\ \texttt{eq}(\texttt{t}_1\texttt{.sub}[i], \texttt{t}_2\texttt{.sub}[i]) \\ \texttt{is\_fsym}(\texttt{t}, f) &\triangleq t\texttt{.sym} = \texttt{f} \\ \texttt{subterm}_\texttt{f}(\texttt{t}, i) &\triangleq \texttt{t.sub}[i] \text{ si } \texttt{t.sym} = \texttt{f} \text{ et } i \in [1..ar(f)] \end{aligned}$$

Exemple 13. Un autre exemple, inspiré d'un exemple 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 (dont le constructeur est pt) peuvent être représentés comme un record `(x :int, y :int)` de coordonnées cartésiennes, en supposant que tous les records représentent des termes (c'est à dire que `is_fsym` répond toujours vrai). On peut alors définir les accès aux sous-termes $\texttt{subterm}_{pt}(\texttt{t}, r)$ et $\texttt{subterm}_{pt}(\texttt{t}, \Theta)$ représentant la norme et l'angle polaire du point.

$$\begin{aligned}
\texttt{eq}(\texttt{t}_1, \texttt{t}_2) &\triangleq \texttt{t}_1.\texttt{x} = \texttt{t}_2.\texttt{x} \wedge \texttt{t}_1.\texttt{y} = \texttt{t}_2.\texttt{y} \\
\texttt{is_fsym}(\texttt{t}, pt) &\triangleq \texttt{true} \\
\texttt{subterm}_{pt}(\texttt{t}, n) &\triangleq \texttt{sqrt}(\texttt{t.x}^2 + \texttt{t.y}^2) \\
\texttt{subterm}_{pt}(\texttt{t}, \theta) &\triangleq \texttt{arctan}(\texttt{t.y}/\texttt{t.x})
\end{aligned}$$

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.

La bonne définition d'un ancrage formel est un point crucial pour permettre la vérification formelle de la bonne compilation d'un problème de filtrage. Comme écrire de telles définitions est technique, on utilise en pratique des outils externes qui génèrent pour nous les ancrages et implantation pour une signature donnée. Gom, qui sera présenté dans le chapitre 4 est un exemple d'outil de ce type.

3.3.3. Le langage intermédiaire `PIL`

On décrit ici la syntaxe du langage intermédiaire `PIL`. Ensuite, après avoir introduit la notion d'environnement d'exécution, nous donnerons une sémantique à grands pas (notée $\mapsto_{bs}$) à ce langage `PIL`. Ce langage intermédiaire est un sous ensemble de C ∩ Java ∩ CAML qui est juste assez expressif pour décrire les procédures de filtrage qui nous intéressent.

Ce langage est très proche du fragment des langages hôtes 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 majeures. Cependant, celle-ci nécessite de supposer une sémantique formelle pour chacun des langages hôtes, pour le fragment du langage considéré.

Syntaxe

Étant donnés $\mathcal{F}$, $\mathcal{X}$, $\mathcal{T}(\mathcal{F})$, $\mathbb{B}$, $\mathbb{N}$, `eq`, `is_fsym`, `subterm`, ainsi qu'un ancrage formel $\ulcorner\urcorner$ comme défini en 15, la syntaxe du langage intermédiaire `PIL` est définie comme présenté

dans la figure 3.2.

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

FIG. 3.2: Syntaxe du langage intermédiaire PIL

L'ensemble des termes de $\langle term \rangle$ est construit à partir des représentants de $\mathcal{T}(\mathcal{F})$ et de la construction $\texttt{subterm}_f$ qui permet de récupérer le i^{me} 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 à effectivement un certain symbole donné en argument comme symbole de tête. L'ensemble des instruction $\langle instr \rangle$ contient seulement 4 instructions :

- `let`, qui représente l'assignation d'une valeur à une variable. On considérera interdit le fait d'assigner la même variable plusieurs fois dans une même portée.
- `if` correspond à la construction « *if-then-else* » : la condition de test est évaluée, et suivant que sa valeur est `true` ou `false`, la première ou la seconde instruction sera exécutée.
- `accept` et
- `refuse` qui sont deux instructions spéciales dont le but est d'approximer le corps des cas de filtrage. En effet, on ne s'intéresse ici qu'à l'instruction de filtrage, et ceci ne nécessite que deux instructions pour placer l'exécution dans un état (`accept` ou `refuse`) exprimant si le motif filtre le sujet ou non.

On utilise une construction de type *if then else* plutôt qu'une instruction *switch* qui est généralement utilisée pour compiler le filtrage car on veut que l'algorithme de filtrage généré par le compilateur soit indépendant de l'ancrage et de la manière dont les termes sont effectivement représentés. En effet, la compilation du filtrage se basant sur l'utilisation d'une instruction *switch*, comme dans [LFM01] utilise implicitement le fait

que chaque terme est représenté en mémoire comme l'application d'un symbole de tête à une liste de sous-termes, et que ce symbole est représenté par un entier machine. Alors, on peut utiliser l'instruction *switch* pour discriminer les cas de différents symboles de tête.

Dans le cadre de Tom, les termes peuvent être représentés sous une forme définissable par l'utilisateur, et ceci rend cette approche inutilisable : il n'est pas possible d'utiliser une constante pour représenter chaque symbole, mais un prédicat peut permettre d'examiner le symbole de tête. Par exemple, si l'on utilise des entiers machines pour représenter un ensemble de termes en les énumérant, il n'est pas possible d'obtenir directement le symbole de tête d'un terme donné, mais une fonction pourra le vérifier.

Note A. Un programme écrit en utilisant le langage intermédiaire `PIL` peut contenir des variables libres, c'est à dire des variables qui ne sont pas liées par une construction `let`. Celles-ci représentent les entrées du programme : dans notre cas, les termes que l'algorithme de filtrage doit tester. On appelle ces variables *variables d'entrée*. Dans la suite, on considérera qu'un programme est évalué dans un environnement pour lequel toutes ces variables libres sont instanciées par une valeur, c'est à dire un représentant de terme.

Exemple 14. Étant donnée une signature $\mathcal{F} = \{a, f\}$ et un ensemble de variables $\mathcal{X} = \{s, x\}$, une compilation possible pour le problème de filtrage $f(x) \ll s$ est le programme `PIL` :

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

Ce programme est évalué dans un environnement qui assigne un représentant de terme à la variable libre $\ulcorner s \urcorner$. Ce programme vérifie tout d'abord que le symbole de tête de s correspond à la représentation de f. Quand c'est le cas, le premier sous-terme de s est assigné à la variable x et le programme se place dans l'état `accept`. Sinon, il se place dans l'état `refuse`.

Sur cet exemple, il est facile de s'auto-persuader que le programme ne se place dans l'état `accept` que si et seulement si le motif filtre effectivement le sujet. On veut maintenant obtenir une preuve formelle et vérifiable de cette propriété.

Nous avons jusque là distingué les objets mathématiques (objets de $\mathbb{B}, \mathbb{N}$ et $\mathcal{X}$) de leurs représentants. Cependant, puisque la plupart des langages de programmation supportent les notions de booléens, entiers naturels et variables, lorsqu'il n'y a pas d'ambiguïté, on notera :

- $\ulcorner \top \urcorner = \texttt{true}$, $\ulcorner \bot \urcorner = \texttt{false}$,
- $\ulcorner 0 \urcorner = 0$, $\ulcorner 1 \urcorner = 1, \ldots, \ulcorner n \urcorner = n$ for $n \in \mathbb{N}$ et
- $\ulcorner x \urcorner = x$ avec $x \in \mathcal{X}$.

Parmi tous les programmes que l'on peut écrire dans le langage PIL, on ne considère que le sous ensemble des programmes dont l'évaluation (dans un environnement complet, comme noté dans la note A), termine dans l'un des états `accept` ou `refuse`, et ce quel que soit l'entrée. Ces programmes sont appelés programmes *bien formés*.

Définition 16 (Bien formé)**.** *Un programme $\pi \in$ PIL est dit* bien formé *lorsqu'il satisfait les propriétés suivantes :*

- *Chaque expression* $\texttt{subterm}_f(\texttt{t}, n)$ *est telle que* `t` *appartient à* $\langle term \rangle$ *et est tel que* $\texttt{is\_fsym}(\texttt{t}, \ulcorner f \urcorner) \equiv \texttt{true}$ *et* $n \in [1..ar(f)]$.
  En pratique, on vérifie que chaque expression de la forme $\texttt{subterm}_f(\texttt{t}, n)$ *appartient à la partie « `else` » d'une instruction* $\texttt{if}(\texttt{is\_fsym}(\texttt{t}, \ulcorner f \urcorner), \ldots)$.
- *Toute variable apparaissant dans une sous-expression a été précédemment initialisée par une construction* `let`*, ou l'a été dans l'environnement d'exécution.*

Afin de vérifier qu'un programme donné est bien formé dans un contexte particulier (modélisant l'évaluation du programme), on introduit un système de type très simple. Un programme pouvant être typé dans ce système dans un environnement fixé sera un programme bien formé dans cet environnement.

Le contexte du système de type est formé par les variables qui ont été introduites dans l'environnement d'évaluation, notées Γ, et d'une liste de couples $(\langle term \rangle, \textbf{symbol})$, notée Δ, représentant le fait que dans l'environnement d'évaluation, le symbole de tête d'un certain terme est connu.

Propriété 1. *Un programme* PIL 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 le système de type présenté dans la figure 3.3.* Γ *contient les variables initialisées par l'environnement, et* Δ *contient les termes dont le symbole de tête est connu, ainsi que ce symbole.*

Démonstration. Soit π in programme PIL, Γ, Δ des contextes tels qu'il existe une dérivation $\Gamma, \Delta \vdash \pi : wf$ dans le système de type de la figure 3.3.

Pour chaque variable v dans le programme π, il existe des contextes Γ, Δ' tels que $\Gamma', \Delta' \vdash v : wf$, et donc $v \in \Gamma'$. Puisque v peut seulement être introduit dans le contexte Γ soit par une initialisation précédente, ou en appliquant la règle de typage pour `let`, la variable v a bien été initialisée.

De plus, pour chaque construction $\texttt{subterm}_f(\texttt{t}, i)$ dans le programme π, parce que la dérivation dans le système de type existe, il existe des contextes Γ', Δ' tels que $\Gamma', \Delta' \vdash \texttt{subterm}_f(\texttt{t}, i) : wf$ avec $(\texttt{t}, \ulcorner f \urcorner) \in \Delta'$. Le couple $(\texttt{t}, \ulcorner f \urcorner)$ pouvant seulement être introduit dans le contexte Δ' par l'application de la règle de typage pour $\texttt{if}(\texttt{is\_fsym}(\texttt{t}, \ulcorner f \urcorner), i_1, i_2)$ ou par un test effectué précédemment, le représentant `t` a été vérifié, et possède bien le symbole f comme symbole de tête.

Inversement, soit π un programme de PIL bien formé dans un certain environnement d'évaluation. Si on initialise les contextes Γ et Δ respectivement avec les variables déjà instanciées dans l'environnement et avec les termes connus pour avoir un symbole de tête particulier, on peut alors construire une dérivation de $\Gamma, \Delta \vdash \pi : wf$. En effet, les règles de typage pour les variables et les constructions `subterm` vont pouvoir être appliquées,

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

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

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

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

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

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

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

FIG. 3.3: Système de type pour vérifier la validité

chaque variable étant soit initialisée dans l'environnement initial, ou introduite par une construction `let` avant d'être utilisée, et les validités des symboles de tête et arités des termes étant vérifiés avant l'utilisation d'une construction `subterm`, soit par un test dans le programme, soit par la connaissance de l'environnement d'évaluation. □

En pratique, lorsqu'on vérifie qu'un programme donné est bien formé, on initialise les environnements avec l'ensemble des variables d'entrée du programme pour Γ (correspondant au sujet contre lequel le programme doit effectuer le filtrage), et une liste de couples Δ vide, ce qui montre que le programme ne fait aucune supposition sur les termes d'entrée.

On peut noter que vérifier la bonne formation d'un programme `PIL` est linéairement décidable, cette propriété puvant être vérifiée par le système de type de la figure 3.3. Le typage d'un programme `PIL` par ce système ne nécessite que l'application d'un jugement de typage par construction du langage constituant ce programme. La vérification est alors linéaire dans la taille du programme.

Exemple 15. Le programme donné dans l'exemple 14 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.

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 $\texttt{if}(\texttt{is\_fsym}(\ulcorner s \urcorner, \ulcorner g \urcorner), \ldots)$.

Environnements

Étant donné un problème de filtrage, le fait de savoir si un terme particulier filtre ou ne filtre pas est bien évidemment 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, ainsi que la définition du filtrage présentée en 6 le requiert.

On introduit pour cela une notion d'*environnement*, modélisant l'état de la mémoire de la machine durant l'évaluation d'un programme. Pour représenter une substitution, on modélise un environnement par une pile d'assignations de termes concrets (des représentants de termes) à des noms de variables. De plus, on définit aussi une fonction Φ qui permet de passer d'un environnement concret à une substitution algébrique.

Définition 17 (Environnement)**.** *Un* environnement *atomique ϵ est une assignation de $\ulcorner \mathcal{X} \urcorner$ dans $\ulcorner \mathcal{T}(\mathcal{F}) \urcorner$, notée $[x \leftarrow \ulcorner t \urcorner]$. La composition d'environnements est associative à gauche et s'écrira $[x_1 \leftarrow \ulcorner t_1 \urcorner][x_2 \leftarrow \ulcorner t_2 \urcorner] \cdots [x_k \leftarrow \ulcorner t_k \urcorner]$. L'application d'un environnement est telle que :*

$$\epsilon[x \leftarrow \ulcorner t \urcorner](y) = \begin{cases} \ulcorner t \urcorner & \textit{si } y \equiv x \\ \epsilon(y) & \textit{sinon} \end{cases}$$

On étend la notion d'environnement à un morphisme ϵ' de `PIL` *dans* `PIL` *et on se donnera la notation $\mathcal{E}nv$ pour désigner l'ensemble de tous les environnements. L'extension de la notion d'environnement à un morphisme de* `PIL` *dans* `PIL` *est définie inductivement par :*

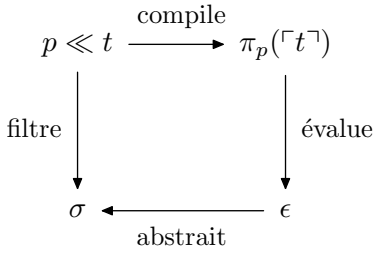
- $\epsilon'(\ulcorner x \urcorner) = \epsilon(\ulcorner x \urcorner)$ *pour toute variable* $x \in \mathcal{X}$
- $\epsilon'(\texttt{subterm}_f(t, \ulcorner n \urcorner)) = \texttt{subterm}_f(\epsilon'(t), \ulcorner n \urcorner)$ *pour tout* $f \in \mathcal{F}$, $n \in \mathbb{N}$ *et* $t \in \langle term \rangle$
- $\epsilon'(\texttt{eq}(t_1, t_2)) = \texttt{eq}(\epsilon'(t_1), t_2)$ *pour tout* $t_1, t_2 \in \langle term \rangle$
- $\epsilon'(\texttt{is\_fsym}(t, \ulcorner f \urcorner)) = \texttt{is\_fsym}(\epsilon'(t), \ulcorner f \urcorner)$ *pour tout* $f \in \mathcal{F}$, $n \in \mathbb{N}$ *et* $t \in \langle term \rangle$
- $\epsilon'(\texttt{let}(\ulcorner x \urcorner, t, i)) = \texttt{let}(\ulcorner x \urcorner, \epsilon'(t), \epsilon'(i))$ *pour toute variable* $x \in \mathcal{X}$, $t \in \langle term \rangle$ *et* $i \in \langle instr \rangle$
- $\epsilon'(\texttt{if}(c, i_1, i_2)) = \texttt{if}(\epsilon'(e), \epsilon'(i_1), \epsilon'(i_2))$ *pour tout* $e \in \langle bexpr \rangle$ *et* $i_1, i_2 \in \langle instr \rangle$
- $\epsilon'(f(t_1, \ldots, t_n)) = f(\epsilon'(t_1), \ldots, \epsilon'(t_1))$ *pour tout symbole de fonction* $f \in \mathcal{F}$ *d'arité* n
- $\epsilon'(t) = t$ *dans tous les autres cas.*

On définit alors la fonction Φ, permettant d'obtenir une substitution algébrique à partir d'un environnement d'exécution concret pour `PIL`.

Définition 18. *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 :*

$$\sigma = \{x_i \mapsto t_i \mid \epsilon(\ulcorner x_i \urcorner) = \ulcorner t_i \urcorner \text{ avec } x_i \in \mathcal{X} \text{ et } t_i \in \mathcal{T}(\mathcal{F})\}$$

Alors, pour prouver la correction du code compilé π_p correspondant à un certain motif p, on veut s'assurer que pour le modèle d'exécution du langage dans lequel le programme π est décrit (noté « évalue »), et pour tout terme t, le diagramme suivant commute :



Afin de pouvoir vérifier cette propriété formellement, on devra rendre le mécanisme d'évaluation du langage `PIL` explicite.

Sémantique à grands pas du langage `PIL`

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 constitués d'un environnement et d'une instruction du langage `PIL`, notés $\langle \epsilon, i \rangle$. La relation de réduction pour la sémantique à grands pas est présentée dans la figure 3.4

Cette sémantique est assez commune, traitant d'un langage très simple. Cependant, on peut noter que les conditions (pour le « *if-then-else* ») sont évaluées modulo un ancrage formel $\ulcorner \urcorner$ et les équivalences données dans la définition 15.

Ainsi, dans le programme de l'exemple 14, si on évalue le programme dans un environnement où s a pour valeur $\ulcorner f(a) \urcorner$, la condition $[s \leftarrow \ulcorner f(a) \urcorner](\texttt{is\_fsym}(s, \ulcorner f \urcorner)) \equiv \texttt{true}$ est équivalente à la condition $\ulcorner \mathcal{S}ymb(f(a)) = f \urcorner \equiv \texttt{true}$, qui dans ce cas est vraie, car le symbole de tête de $f(a)$ est bien f.

3.3.4. Processus de validation

Nous avons maintenant 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.

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

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

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

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

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

FIG. 3.4: Sémantique à grands pas pour PIL

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`.

Ensuite, on établira un théorème de correction et des propriétés qui montrent comment cette approche peut être utilisée pour vérifier formellement qu'un problème de filtrage est effectivement compilé correctement. Ce résultat sera ensuite étendu pour supporter des fonctionnalités de la construction **%match** de Tom ou de CAML comme les motifs multiples.

Correction de la compilation du filtrage

La sémantique à grands pas introduite précédemment nous permet de définir maintenant la notion de compilation *correcte* π_p d'un motif, ou problème de filtrage $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, c'est à dire une compilation à la fois *valide* et *complète*.

Définition 19 (Compilation valide)**.** *Étant donné un ancrage formel* $\ulcorner \urcorner$*, un programme bien formé* π_p *est une compilation* valide *de* p *lorsque l'on a les deux propriétés :*

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

Définition 20 (Compilation complète)**.** *Étant donné un ancrage formel* $\ulcorner\urcorner$*, un programme bien formé* π_p *est une compilation* complète *de* p *lorsque l'on a les deux propriétés :*

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

Définition 21 (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*.*

De manière informelle, la définition 21 dit que pour qu'un programme π_p soit une compilation *valide* d'un motif p, il faut que l'exécution du programme π_p mène à l'état `accept` pour tous les représentants des termes qui sont filtrés par le motif p, et réciproquement. L'exécution doit en outre conduire à l'état `refuse` si et seulement si p ne filtre pas le terme représenté par son sujet. De plus, les définitions 19 et 20 assurent que l'environnement ϵ' calculé par l'exécution du programme π_p correspond à une substitution σ telle que $\sigma(p) = t$, c'est-à-dire qui exhibe une solution du problème de filtrage.

Pour certifier qu'un certain programme π_p correspond bien à une compilation correcte d'un motif p, le théorème 1 montre qu'il est suffisant de calculer toutes les dérivations possibles de π_p pour savoir si une compilation est correcte ou non.

Théorème 1. *Étant donné un ancrage formel* $\ulcorner\urcorner$*, un motif* $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *et un programme bien formé* $\pi_p \in \texttt{PIL}$*, on a :*

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

Démonstration. La preuve de ce théorème découle de l'application des propriétés 2 et 3 suivantes. □

Propriété 2. *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.*

Démonstration. Par induction sur la structure des instructions.

On considère comme hypothèse d'induction la propriété suivante : pour tout environnement $\epsilon \in \mathcal{E}nv$, la dérivation d'une instruction bien formée dans le contexte Γ, Δ $i \in \langle instr\rangle$ conduit à l'un des états `accept` ou `refuse`, de manière unique.

Soit $i \in \langle instr\rangle$:

- lorsque $i = \texttt{accept}$ ou $i = \texttt{refuse}$, une seule règle d'inférence de la sémantique à grands pas présentée en figure 3.4 peut être appliquée : l'axiome (*accept*), et respectivement ($refuse$). Ainsi, la dérivation mène de manière unique soit à l'état `accept`, soit à l'état `refuse`.
- lorsque $i = \texttt{let}(x, u, i_1)$, une seule règle d'inférence peut être appliquée : la règle (let).

$$\frac{\langle \epsilon[x \leftarrow \ulcorner t \urcorner], i_1\rangle \mapsto_{bs} \langle \epsilon', i_2\rangle}{\langle \epsilon, \texttt{let}(x, u, i_1)\rangle \mapsto_{bs} \langle \epsilon', i_2\rangle} \text{ si } \epsilon(u) \equiv \ulcorner t \urcorner \quad (let)$$

  Pour compléter la preuve, on doit montrer qu'il existe un terme $t \in \mathcal{T}(\mathcal{F})$ tel que l'évaluation de l'expression u dans l'environnement ϵ soit égale au représentant concret de t : $\epsilon(u) \equiv \ulcorner t \urcorner$. On sait que i est une instruction bien formée dans le contexte Γ, Δ, donc toute variable apparaissant dans u a été initialisée précédemment : $\epsilon(u)$ est clos. Comme $u \in \langle term\rangle$, u est soit un représentant de terme, soit une variable, soit une construction $\texttt{subterm}_f$, avec f un symbole de $\mathcal{F}$.
  - Si u est déja un représentant de terme, alors il existe $t \in \mathcal{T}(\mathcal{F})$ tel que $\epsilon(u) \equiv \ulcorner t \urcorner$.
  - Si u est une variable, u a été instancée par un représentant de terme dans l'environnement d'évaluation car i est bien formé.
  - Le fait que i soit bien formé assure que chaque construction $\texttt{subterm}_f$ avec $f \in \mathcal{F}$ est encapsulée par un $\texttt{if}(\texttt{is\_fsym}(\ulcorner \ldots \urcorner, \ulcorner f \urcorner), \ldots)$ et que chaque variable a été initialisée. Alors, toute expression $\texttt{subterm}_f$ est $\equiv$-équivalente (voir définition 15) à un représentant de terme : $\exists t \in \mathcal{T}(\mathcal{F})$ tel que $\epsilon(u) \equiv \ulcorner t \urcorner$.

  De par l'hypothèse d'induction, on sait que la dérivation de i_1 conduit de manière unique soit à `accept` soit à `refuse`, et donc la dérivation de i est elle aussi unique et mène soit à l'état `accept`, soit à l'état `refuse`.
- lorsque $i = \texttt{if}(e, i_1, i_2)$, en utilisant des arguments similaires au cas précédent, on montre que chaque $\langle term\rangle$ apparaissant dans e est $\equiv$-equivalent à un représentant de terme. Comme l'expression e est un représentant de booléen, une construction `is_fsym` ou une construction `eq` (dont les sous termes sont des représentants de termes, ou $\equiv$-équivalents à des représentants de termes), e est par définition $\equiv$-équivalent à un représentant de booléen. Ainsi, on a soit $e \equiv \texttt{true}$, soit $e \equiv \texttt{false}$. Lorsque $e \equiv \texttt{true}$ (respectivement $e \equiv \texttt{false}$), la seule règle de la sémantique à grands pas applicable est ($iftrue$) (respectivement ($iffalse$)) et on a :

$$\frac{\langle \epsilon, i_1\rangle \mapsto_{bs} \langle \epsilon', i_3\rangle}{\langle \epsilon, \texttt{if}(e, i_1, i_2)\rangle \mapsto_{bs} \langle \epsilon', i_3\rangle} \text{ si } \epsilon(e) \equiv \texttt{true} \quad (iftrue)$$

  En utilisant l'hypothèse d'induction, la réduction de i_1 (respectivement i_2) dans l'environnement ϵ conduit soit à `accept`, soit à `refuse` de manière unique, et la réduction de i fait de même.

Ainsi, étant donné $\epsilon \in \mathcal{E}nv$, la réduction d'une instruction bien formée i dans un environnement Γ, Δ conduit soit à l'état `accept`, soit à l'état `refuse`, et ce de manière unique. □

Propriété 3. *Étant donnés un ancrage formel* $\ulcorner\urcorner$ *et un programme* $\pi_p \in$ `PIL` *bien formé, la propriété suivante (liant les définitions 19 et 20) est vraie :*

$$\forall \epsilon \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}), (sound_{OK}) \Rightarrow (complete_{KO}) \text{ et } (complete_{OK}) \Rightarrow (sound_{KO})$$

Démonstration. Supposons $(sound_{OK})$ et $p \not\ll t$. Puisque la dérivation de $\langle \epsilon, \pi_p(\ulcorner t \urcorner)\rangle$ est unique (d'après la propriété 2), on ne peut pas avoir de dérivation $\langle \epsilon, \pi_p(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}\rangle$ sans contredire $p \not\ll t$. La propriété 2 montre aussi qu'une dérivation mène nécessairement soit à `accept`, soit à `refuse`. Ainsi, on a nécessairement $\langle \epsilon, \pi_p(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{refuse}\rangle$ et donc $(sound_{OK}) \Rightarrow (complete_{KO})$.

Supposons maintenant $(complete_{OK})$ et $\exists \epsilon' \in \mathcal{E}nv$, $\langle \epsilon, \pi_p(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{refuse}\rangle$. On doit alors montrer que $p \not\ll t$. Si $p \ll t$, alors en utilisant $(complete_{OK})$ on obtient $\exists \epsilon' \in \mathcal{E}nv$, $\langle \epsilon, \pi_p(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}\rangle$. Ceci contredit l'unicité de la dérivation de $\langle \epsilon, \pi_p\rangle$, dont on a $p \not\ll t$. Ainsi, $(complete_{OK}) \Rightarrow (sound_{KO})$. □

Interprétation de la sémantique à grands pas

Le théorème 1 est le résultat qui permet de prouver qu'un programme π_p est une compilation correcte d'un motif p. Cependant, l'équivalence entre $\langle \epsilon, \pi_p(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}\rangle$ 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 une environnement ϵ tel que $\Phi(\epsilon)(p) = t$ ».

Ce résultat est utile en pratique, car la sémantique à grands pas $\mapsto_{bs}$ n'apparaît plus explicitement dans la propriété à prouver. Ceci rend la preuve plus petite et éventuellement plus facile à traiter pour un prouveur automatique.

Définition 22. *Une dérivation dans la sémantique à grands pas qui mène à un état* `accept` *est dite* réussie. *Soit* `s` *une variable d'entrée et* π_p *un programme bien formé de* `PIL`.

À chaque dérivation $\mathcal{D}$ *réussie dans la sémantique de* `PIL`, *on associe une conjonction* $\mathcal{C}_\mathcal{D}$ *de toutes les contraintes levées par la dérivation.* $\mathcal{C}\pi_p(\texttt{s})$ *est définie comme la disjonction de toutes les contraintes* $\mathcal{C}_\mathcal{D}$ *pour toutes les dérivations réussies dans la sémantique de* `PIL`.

En pratique, on peut utiliser un outil dédié pour extraire les contraintes d'un programme. Commençant à partir d'un environnement ϵ contenant seulement la variable

d'entrée, il est suffisant de calculer toutes les dérivations dans la sémantique à grands pas menant à `accept` : $\langle \epsilon, \pi_p(\ulcorner t \urcorner) \rangle \mapsto_{bs} \langle \epsilon', \texttt{accept} \rangle$. Ces contraintes correspondent aux conditions levées par l'application d'une règles de la sémantique donnée dans la figure 3.4.

Note B. On peut remarquer ici que le nombre de contraintes générées pour un programme donné est linéaire en sa taille. En pratique, pour un motif simple, le programme est lui même linéaire en la taille du motif, ce qui rend cette approche raisonnable.

Étant donné un terme t, on note $\mathcal{C}\pi_p(t)$ le fait que t satisfasse la contrainte $\mathcal{C}\pi_p$. Un exemple d'une telle contrainte est donné dans la figure 3.6.

Propriété 4. *Étant donnés un ancrage formel* $\ulcorner \urcorner$ *et un programme* $\pi_p \in \texttt{PIL}$ *bien formé, on a :*

$$\begin{aligned} &\forall \epsilon \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}), \\ &\quad \exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_p(\ulcorner t \urcorner) \rangle \mapsto_{bs} \langle \epsilon', \texttt{accept} \rangle \Leftrightarrow \mathcal{C}\pi_p(t) \end{aligned}$$

Démonstration. Il est clair que si la dérivation $\langle \epsilon, \pi_p(\ulcorner t \urcorner) \rangle \mapsto_{bs} \langle \epsilon', \texttt{accept} \rangle$ est possible, alors t satisfait la contrainte $\mathcal{C}\pi_p$. D'autre part, si t satisfait la contrainte $\mathcal{C}\pi_p$, alors une dérivation menant à l'état `accept` peut être construite (et cette dérivation est unique). □

Théorème 2. *Étant donnés un ancrage formel* $\ulcorner \urcorner$*, un motif* $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *et un programme bien formé* $\pi_p \in \texttt{PIL}$*, on a :*

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

Ce théorème peut être utilisé pour prouver la compilation d'un motif correcte. Comme illustré par la figure 3.5, 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 $t_1, \ldots, t_k$ correspond aux sous-termes d'un sujet t. Lorsqu'elle est satisfaire, 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.

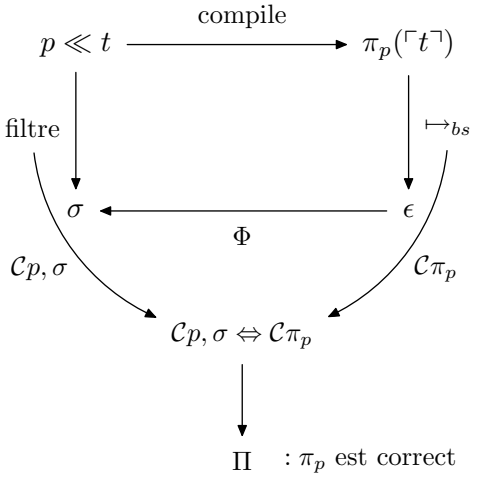


FIG. 3.5: Schéma général de la certification

Un exemple détaillé

On considère comme exemple le motif $g(x, b)$, avec $x \in \mathcal{X}$, $g, b \in \mathcal{F}$. Supposons que notre compilateur de filtrage produise le programme suivant, dans lequel `s` est une variable d'entrée :

$\pi_{g(x,b)}(\mathtt{s}) \triangleq$
  $\mathtt{if}(\mathtt{is\_fsym}(\mathtt{s}, \ulcorner g \urcorner),$
    $\mathtt{let}(x_1, \mathtt{subterm}_g(\mathtt{s}, 1),$
      $\mathtt{let}(x_2, \mathtt{subterm}_g(\mathtt{s}, 2),$
        $\mathtt{let}(x,\ x_1,$
          $\mathtt{if}(\mathtt{is\_fsym}(x_2, \ulcorner b \urcorner), \mathtt{accept}, \mathtt{refuse}))))),$
    $\mathtt{refuse}$
)

Étant donné un terme t et un environnement $\epsilon_0 = [\mathtt{s} \leftarrow \ulcorner t \urcorner]$, supposons que l'on ait une dérivation de $\langle \epsilon_0, \pi_{g(x,b)}(\mathtt{s}) \rangle \mapsto_{bs} \langle \epsilon', \mathtt{accept} \rangle$. La figure 3.6 montre la seule dérivation qui peut être calculée en appliquant les règles d'inférence définies dans la figure 3.4.

Pour rendre cette dérivation possible, l'ensemble suivant de contraintes doit être satisfait :

$$\mathcal{C}\pi_p(\mathtt{s}) = \begin{cases} \epsilon_0(\mathtt{is\_fsym}(\mathtt{s}, \ulcorner g \urcorner)) & \equiv & \mathtt{true} & (1) \\ \epsilon_0(\mathtt{subterm}_g(\mathtt{s}, 1)) & \equiv & \ulcorner t_{|1} \urcorner & (2) \\ \epsilon_1(\mathtt{subterm}_g(\mathtt{s}, 2)) & \equiv & \ulcorner t_{|2} \urcorner & (3) \\ \epsilon_2(x_1) & \equiv & \ulcorner t_{|1} \urcorner & (4) \\ \epsilon_3(\mathtt{is\_fsym}(x_2, \ulcorner b \urcorner)) & \equiv & \mathtt{true} & (5) \end{cases}$$

(1) et (5) peuvent être simplifiés en utilisant les équations de l'ancrage formel et (2), (3) et (4) sont des tautologies. Ainsi, pour prouver la correction de la compilation de p en

$$\begin{array}{l|l}
\texttt{Let}_3 \triangleq \texttt{let}(x, x_1, \texttt{if}(\texttt{is_fsym}(x_2, \ulcorner b \urcorner), \texttt{accept}, \texttt{refuse})) & \epsilon_0 = [\texttt{s} \leftarrow \ulcorner t \urcorner] \\
\texttt{Let}_2 \triangleq \texttt{let}(x_2, \texttt{subterm}_g(\texttt{s}, 2), \texttt{Let}_3) & \epsilon_1 = \epsilon_0[x_1 \leftarrow \ulcorner t_{|1} \urcorner] \\
\texttt{Let}_1 \triangleq \texttt{let}(x_1, \texttt{subterm}_g(\texttt{s}, 1), \texttt{Let}_2) & \epsilon_2 = \epsilon_1[x_2 \leftarrow \ulcorner t_{|2} \urcorner] \\
 & \epsilon_3 = \epsilon_2[x \leftarrow \ulcorner t_{|1} \urcorner]
\end{array}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle \epsilon_3, \texttt{accept} \rangle \mapsto_{bs} \langle \epsilon_3, \texttt{accept} \rangle}}{\langle \epsilon_3, \texttt{if}(\texttt{is\_fsym}(x_2, \ulcorner b \urcorner), \texttt{accept}, \texttt{refuse}) \rangle \mapsto_{bs} \langle \epsilon_3, \texttt{accept} \rangle}\ \boxed{5} = \epsilon_3(\texttt{is\_fsym}(x_2, \ulcorner b \urcorner)) \equiv \texttt{true}}{\langle \epsilon_2, \texttt{let}(x, x_1, \texttt{if}(\texttt{is\_fsym}(x_2, \ulcorner b \urcorner), \texttt{accept}, \texttt{refuse})) \rangle \mapsto_{bs} \langle \epsilon_3, \texttt{accept} \rangle}\ \boxed{4} = \epsilon_2(x_1) \equiv \ulcorner t_{|1} \urcorner}{\langle \epsilon_1, \texttt{let}(x_2, \texttt{subterm}_g(\texttt{s}, 2), \texttt{Let}_3) \rangle \mapsto_{bs} \langle \epsilon_3, \texttt{accept} \rangle}\ \boxed{3} = \epsilon_1(\texttt{subterm}_g(\texttt{s}, 2)) \equiv \ulcorner t_{|2} \urcorner}{\langle \epsilon_0, \texttt{let}(x_1, \texttt{subterm}_g(\texttt{s}, 1), \texttt{Let}_2) \rangle \mapsto_{bs} \langle \epsilon_3, \texttt{accept} \rangle}\ \boxed{2} = \epsilon_0(\texttt{subterm}_g(\texttt{s}, 1)) \equiv \ulcorner t_{|1} \urcorner}{\langle \epsilon_0, \texttt{if}(\texttt{is\_fsym}(\texttt{s}, \ulcorner g \urcorner), \texttt{Let}_1, \texttt{refuse}) \rangle \mapsto_{bs} \langle \epsilon_3, \texttt{accept} \rangle}\ \boxed{1} = \epsilon_0(\texttt{is\_fsym}(\texttt{s}, \ulcorner g \urcorner)) \equiv \texttt{true}$$

FIG. 3.6: Exemple de dérivation menant à `accept`. On a $\mathcal{C}\pi_p(\texttt{s}) = \{\boxed{1} \wedge \boxed{2} \wedge \boxed{3} \wedge \boxed{4} \wedge \boxed{5}\}$

π_p, on doit prouver les équivalences :

$$\begin{gathered}\forall t \in \mathcal{T}(\mathcal{F}),\\ \sigma(g(x,b)) = t \wedge \sigma = \{x \mapsto t_{|1}\}\\ \Longleftrightarrow\\ \mathcal{S}ymb(t) = g \wedge \mathcal{S}ymb(t_{|2}) = b\end{gathered}$$

Ceci est prouvé en appliquant tout d'abord la substitution dans la première partie des obligations de preuve et ensuite en utilisant les définitions des termes, symboles et sous-termes.

Dans cet exemple, avec g un symbole de fonction d'arité 2 et b un symbole constant (ou symbole de fonction d'arité 0), la définition du mapping donne les axiomes suivants :

$$\begin{gathered}\forall t \in \mathcal{T}(\mathcal{F}),\ \mathcal{S}ymb(t) = g \Leftrightarrow \exists x, y \in \mathcal{T}(\mathcal{F}),\ t = g(x,y)\\ \forall t \in \mathcal{T}(\mathcal{F}),\ \mathcal{S}ymb(t) = b \Leftrightarrow (t = b)\\ \forall x, y \in \mathcal{T}(\mathcal{F}),\ g(x,y)_{|1} = x\\ \forall x, y \in \mathcal{T}(\mathcal{F}),\ g(x,y)_{|2} = y\end{gathered}$$

Les deux premiers axiomes définissent la signification de la fonction $\mathcal{S}ymb$, tandis que les deux suivants définissent la fonction sous-terme pour les termes dont le symbole de tête est g. Comme on utilise un prouveur de théorème dans la logique du premier ordre, il est nécessaire de donner une telle définition pour chaque symbole et pour chaque sous-terme associé à un symbole.

Pour prouver l'implication de gauche à droite, on applique simplement la substitution dans la partie gauche, et ensuite on applique le premier axiome pour obtenir $\mathcal{S}ymb(t) = g$ et le quatrième axiome pour obtenir $\mathcal{S}ymb(t_{|2}) = b$.

Pour prouver l'implication restante ($\Leftarrow$), on applique le premier axiome à la première contrainte pour obtenir $\exists x, y$ tels que $t = g(x,y)$. On applique alors les troisième et quatrième axiomes pour instancier x et y respectivement par $t_{|1}$ et $t_{|2}$. Le second axiome appliqué à la seconde contrainte donne $t_{|2} = b$. On peut alors obtenir $g(t_{|1}, b)$ ainsi qu'extraire la substitution.

La forme de telles substitutions est assez simple, mais un grand nombre d'entre elles peut être généré, et donc ce genre de preuve est plutôt à laisser faire par un prouveur de théorèmes automatique. On utilisera pour cela le prouveur Zenon [Dol], qui est un prouveur de théorème automatique basé sur la méthode des tableaux. Une des particularités intéressantes de Zenon est de générer une preuve formelle dans le langage d'entrée de l'assistant de preuve Coq lorsqu'un théorème peut être prouvé. Dans notre cas, c'est donc un témoin de la correction de la compilation du filtrage qui est alors généré, et associé au code généré.

3.4. Une extension pour Tom et Caml : *multi-match*

On a montré jusqu'ici la méthode utilisée pour prouver correcte la compilation de motifs algébriques simples. On peut remarquer cependant que ceci inclut les motifs non-linéaires, car on peut utiliser l'instruction `eq` pour vérifier si deux termes sont égaux.

Cependant, ceci ne suffit pas pour être capable de vérifier la correction de la compilation du filtrage du langage Tom, ou d'autres langages supportant le filtrage, comme CAML. En effet, ceux-ci supportent des constructions de filtrage (respectivement **%match** et `match with`), qui agrègent plusieurs motifs dans une seule construction. Cette fonctionnalité est très importante pour l'utilisation effective du filtrage comme outil de programmation : en effet, le filtrage non seulement permet d'explorer une structure de donnée, mais aussi de « *brancher* » le programme en fonction de la forme du terme à filtrer, et donc à programmer par cas en discriminant sur la forme des arguments. On va donc s'intéresser tout particulièrement à vérifier la bonne compilation des constructions de ce type.

On s'intéresse donc à des constructions de la forme `match` s `with` $(p_1 \rightarrow a_1), \ldots, (p_n \rightarrow a_n)$, dont la sémantique est :

- si p_1 filtre le sujet s, le programme va dans l'état `accept`, et pour garder la trace du motif ayant filtré, l'état `accept` sera labellé par p_1, ce qui sera noté $\texttt{accept}_{p_1}$.
- sinon, le sous-problème `match` s `with` $(p_2 \rightarrow a_2), \ldots, (p_n \rightarrow a_n)$ est considéré.
- Lorsqu'aucun motif p_i ne filtre le sujet s, le programme se place dans l'état `refuse`.

Cette construction de filtrage nouvelle peut être facilement compilée en utilisant le langage intermédiaire `PIL` : il suffit de copier dans la branche `refuse` du premier motif, la compilation du sous problème de filtrage sans le premier motif, et récursivement. Cependant, afin d'éviter la duplication de code, et pour faciliter l'expression de tels algorithmes dans le langage `PIL`, il est pratique de considérer une construction *séquence* : $\langle instr \rangle$; $\langle instr \rangle$.

De plus, une telle instruction de séquence étant présente dans l'intersection des langages que Tom peut utiliser comme langage de sortie, cette instruction est utilisée par le générateur de code du compilateur Tom pour ces constructions. Notre objectif étant de valider le code généré par le compilateur, il ne serait pas raisonnable de ne pas considérer cette construction dans notre langage intermédiaire `PIL`.

Une fois notre langage intermédiaire `PIL` dans sa forme la plus complète (c'est à dire étendu avec l'instruction « ; »), on donnera en section 3.5.1 un algorithme formellement défini permettant d'extraire les contraintes de tout programme `PIL` bien formé.

3.4.1. Sémantique de la séquence

On doit donner pour la construction *séquence* que l'on ajoute au langage `PIL` une sémantique qui permettra d'évaluer les programmes écrits dans `PIL` étendu avec cette construction « ; ».

Tout d'abord, on peut étendre le système de type de la figure 3.3 en ajoutant une règle permettant de vérifier qu'un programme constitué d'une instruction séquence est bien formé :

$$\frac{\Gamma, \Delta \vdash i_1 : wf \quad \Gamma, \Delta \vdash i_2 : wf}{\Gamma, \Delta \vdash i_1 \,;\, i_2 : wf}$$

La sémantique d'une instruction « ; » est alors définie par :

$$\frac{\langle \epsilon, i_1\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}_p\rangle}{\langle \epsilon, i_1 \,;\, i_2\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}_p\rangle} \quad (seq_a)$$

$$\frac{\langle \epsilon, i_1\rangle \mapsto_{bs} \langle \epsilon', \texttt{refuse}\rangle \quad \langle \epsilon', i_2\rangle \mapsto_{bs} \langle \epsilon'', i\rangle}{\langle \epsilon, i_1 \,;\, i_2\rangle \mapsto_{bs} \langle \epsilon'', i\rangle} \quad (seq_b)$$

Il est facile de montrer que l'ajout de la règle pour la séquence ne remet pas en cause la propriété d'unicité de la dérivation d'une instruction bien formée.

Note C. La sémantique que nous avons donné à la séquence est adaptée à la preuve de constructions multi-motifs qui correspondent à celles du langage CAML, ou Tom dans lequel les actions associées à chaque filtre interrompent le flot de contrôle du programme.

Cependant, la construction multi-motif de Tom a une sémantique qui diffère de celle du langage CAML : lorsqu'un filtre pour le sujet à été trouvé, l'action associée est exécutée, puis les autres motifs de la construction sont essayés, et leur action associée éventuellement exécutée. On doit alors modifier la règle (seq_a) comme suit :

$$\frac{\langle \epsilon, i_1\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}_p\rangle \quad \langle \epsilon', i_2\rangle \mapsto_{bs} \langle \epsilon'', i\rangle}{\langle \epsilon, i_1 \,;\, i_2\rangle \mapsto_{bs} \langle \epsilon'', i\rangle} \quad (seq_a)$$

La notion de compilation correcte d'une construction « *match* » est une extension de la définition de la compilation correcte d'un filtre. Les différences viennent de la présence de plusieurs motifs. Ainsi, lorsqu'un filtre est sélectionné pour activer une règle ($\texttt{accept}_p$ dans notre terminologie), on doit s'assurer non seulement que le terme d'entrée est bien filtré par p, mais aussi que tous les précédents motifs ne filtrent pas le sujet. Dans la suite, on ne fera pas de supposition sur la forme du code à valider. Ceci nous assure de pouvoir considérer toute sorte d'optimisation du code de filtrage, comme la factorisation des tests communs, sans avoir à adapter l'algorithme.

3.4.2. Correction de la compilation

Soit $\mathcal{P}_m$ l'ensemble des motifs de la construction « *match* », et $<$ une relation d'ordre total sur les motifs de $\mathcal{P}_m$. Dans le cas des langages de la famille de CAML par exemple, on définit $<$ par l'ordre d'apparition des motifs dans le texte de la construction de filtrage : $p_i < p_j$ si p_i apparaît avant p_j dans la construction « *match* ».

Définition 23 (Compilation multi-motif valide)**.** *Étant donné un ancrage formel $\ulcorner\urcorner$, un programme bien formé π_m est une compilation* valide *d'un multi-motif m lorsque l'on a simultanément :*

$$\begin{array}{l}
\forall \epsilon \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}) : \\
\quad \forall p \in \mathcal{P}_m, \exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_m(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}_p\rangle \\
\qquad \Rightarrow \Phi(\epsilon')(p) = t \wedge (\forall p' \in \mathcal{P}_m \textit{ tel que } p' < p, \Phi(\epsilon')(p') \neq t) \quad (Msound_{OK}) \\
\quad \exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_m(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{refuse}\rangle \Rightarrow \forall p \in \mathcal{P}_m, p \not\ll t \quad (Msound_{KO})
\end{array}$$

Définition 24 (Compilation multi-motif complète)**.** *Étant donné un ancrage formel* $\ulcorner\urcorner$*, un programme bien formé* π_m *est une compilation* complète *d'un multi-motif* m *lorsque l'on a les deux propriétés suivantes :*

$$\begin{array}{l}\forall \epsilon \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}) : \\ \quad \forall p \in \mathcal{P}_m, p \ll t \wedge (\forall p' \in \mathcal{P}_m \text{ tel que } p' < p, p' \not\ll t) \Rightarrow \\ \qquad \exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_m(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}_p\rangle \wedge \Phi(\epsilon')(p) = t \\ \qquad\qquad\qquad \wedge (\forall p' \in \mathcal{P}_m \text{ tel que } p' < p, \Phi(\epsilon')(p') \neq t) \quad (Mcomplete_{OK}) \\ \quad \forall p \in \mathcal{P}_m, p \not\ll t \Rightarrow \exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_m(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{refuse}\rangle \quad (Mcomplete_{KO})\end{array}$$

De manière similaire à la définition 21, la compilation d'une construction multi-motif m en un programme π_m est dite *correcte* lorsqu'elle est à la fois valide et complète.

Note D. Lorsque l'on s'intéresse à des constructions multi-motifs avec la sémantique de Tom, on doit non seulement considérer une règle de sémantique pour la séquence modifiée comme illustré en note C, mais aussi modifier les définitions de la validité et de la complétude pour prendre en compte cette sémantique. Avec la sémantique de Tom, il n'est pas nécessaire que les motifs précédant celui qui filtre n'aient pas filtré eux-même. Il faut donc modifier les règles $(Msound_{OK})$ et $(Msound_{OK})$ pour supprimer cette condition comme suit :

$$\begin{array}{l}\forall \epsilon \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}) : \\ \quad \forall p \in \mathcal{P}_m, \exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_m(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}_p\rangle \Rightarrow \Phi(\epsilon')(p) = t \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (Msound_{OK}) \\ \forall \epsilon \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}) : \\ \quad \forall p \in \mathcal{P}_m, p \ll t \Rightarrow \exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_m(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}_p\rangle \wedge \Phi(\epsilon')(p) = t \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (Mcomplete_{OK})\end{array}$$

Propriété 5. *La dérivation d'une instruction bien formée* $i \in \langle instr\rangle$ *du langage* `PIL` *étendu à la séquence dans un environnement* Γ, Δ *mène de manière unique à l'un des états* `accept` *ou* `refuse`*.*

Démonstration. On procède par induction sur la structure des instructions. La preuve est très similaire à la preuve de la propriété 2.

On utilise l'extension présentée plus haut du système de type de la figure 3.3 avec une règle pour vérifier qu'une séquence est bien formée en imposant que les deux composants de la séquence sont bien formés pour vérifier qu'un programme dans le langage `PIL` étendu est bien formé.

On doit alors ajouter à la preuve présentée pour la propriété 2 un cas traitant la situation ou l'instruction i est une séquence.

Soit $i = i_1 \,;\, i_2$ une séquence. i est bien formé, donc i_1 et i_2 sont eux-mêmes bien formés. Par hypothèse d'induction, la réduction de i_1 est unique, et mène soit à l'état `accept`, soit à l'état `refuse`.

– Dans le premier cas, la règle de la sémantique étendue présentée dans la sous-section 3.4.1 (seq_a) est applicable. La réduction de $i_1 ; i_2$ est égale à la réduction de i_1, et est donc unique.
– Dans le second cas, la règle (seq_b) est applicable. Puisque la réduction de i_2 est elle aussi unique, la réduction de $i = i_1 ; i_2$ est unique.

□

Théorème 3. *Étant donnés un ancrage formel* $\ulcorner\urcorner$*, m une construction multi-motif et* $\pi_m \in \texttt{PIL}$ *un programme bien formé, on a :*

$$\pi_m \textit{ est une compilation correcte de } m$$
$$\iff$$
$$\forall \epsilon \in \mathcal{E}nv, \forall t \in \mathcal{T}(\mathcal{F}), \forall p \in \mathcal{P}_m :$$
$$\exists \epsilon' \in \mathcal{E}nv, \langle \epsilon, \pi_m(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}_p\rangle$$
$$\Leftrightarrow \Phi(\epsilon')(p) = t \wedge (\forall p' \in \mathcal{P}_m \textit{ tel que } p' < p, \Phi(\epsilon')(p') \neq t)$$

Démonstration. On veut montrer, de la même manière que pour la propriété 3, que $(Msound_{OK}) \Rightarrow (Mcomplete_{KO})$ et $(Mcomplete_{OK}) \Rightarrow (Msound_{KO})$.

Dans le premier cas, supposons $(Msound_{OK})$ et $\forall p \in \mathcal{P}_m, p \not\ll t$. Puisque la réduction de $\langle \epsilon, \pi_m(\ulcorner t \urcorner)\rangle$ est unique, il est impossible d'avoir une réduction de $\langle \epsilon, \pi_m(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{accept}_p\rangle$. Cette réduction existe, et donc on a $\langle \epsilon, \pi_m(\ulcorner t \urcorner)\rangle \mapsto_{bs} \langle \epsilon', \texttt{refuse}\rangle$.

Le second cas peut être prouvé de manière similaire. □

Afin de prouver que la compilation π_m d'une construction multi-motif m est correcte, on doit considérer chaque instruction $\texttt{accept}_p$ dans le programme séparément. Pour chaque motif p dans la construction « *match* », on construit toutes les dérivations dans $\mapsto_{bs}$ qui mènent vers $\texttt{accept}_p$, et déduit de celles si une contrainte, formée d'une disjonction de conjonctions de contraintes simples $\mathcal{C}\pi_m, p$. On peut alors pour chaque contrainte prouver l'obligation de preuve correspondante, comme exprimé dans le théorème 3.

3.5. Génération des contraintes et des obligations de preuve

Il est maintenant nécessaire d'expliciter l'algorithme générant les contraintes associées à un programme bien formé du langage PIL. On décrit ici cet algorithme, et discute de sa complexité.

3.5.1. Algorithme de collecte de contraintes

L'extraction commence à partir d'un environnement ϵ instanciant toutes les variables libres du programme à vérifier, comme montré dans la sous-section 3.3.4.

Définition 25 (Extraction des contraintes)**.** *L'algorithme d'extraction de contraintes, noté $\mathcal{C}$, est définit comme suit :*

$$\begin{array}{lcl}
\mathcal{C}(\texttt{let}(x,u,i),but) & = & \mathcal{C}(i,but) \wedge x = u \\
\mathcal{C}(\texttt{if}(e,i_1,i_2),but) & = & (\mathcal{C}(i_1,but) \wedge e \equiv \texttt{true}) \\
 & & \vee \mathcal{C}(i_2,but) \wedge e \equiv \texttt{false}) \\
\mathcal{C}(i_1\ ;\ i_2,but) & = & \mathcal{C}(i_1,but) \vee (\mathcal{C}(i_1,\texttt{refuse}) \wedge \mathcal{C}(i_2,but)) \\
\mathcal{C}(i,but) & = & \top \textit{ si } i = but, \\
 & & \bot \textit{ sinon}
\end{array}$$

Cet algorithme calcule une disjonction de conjonction de contraintes élémentaires. La disjonction représente les différents *chemins* que le flot de contrôle peut suivre dans le programme, tandis que les conjonctions représentent les ensembles de contraintes qui sont levées sur un chemin.

Il est intéressant de noter que dans le case de motifs simples, lorsque l'on utilise pas l'instruction « ; » et qu'il n'y a qu'une occurrence de `accept` dans le programme, alors un seul chemin de flot de contrôle est possible pour atteindre l'état `accept`, et ainsi, toutes les disjonctions peuvent être simplifiées pas une analyse booléenne simple de la contrainte générée.

Cependant, cette fonction est encore trop abstraite pour pouvoir être utilisée ou implantée telle qu'elle : on a besoin pour construire les obligations de preuve du problème de correction de la compilation du filtrage des substitutions construites par le programme lorsqu'il atteint l'état `accept`. Pour rendre plus facile l'implantation, on s'autorisera à passer en argument à la fonction d'extraction la substitution construite par l'évaluation, et à l'appliquer aux contraintes dès que possible.

Définition 26 (Extraction des contraintes et substitutions)**.** *On obtient alors l'algorithme d'extraction des contraintes et des substitutions :*

$$\begin{array}{lcl}
\mathcal{C}(\epsilon,\texttt{let}(x,u,i),but) & = & \mathcal{C}(\epsilon[x \leftarrow \epsilon(u)],i,but) \\
\mathcal{C}(\epsilon,\texttt{if}(e,i_1,i_2),but) & = & (\mathcal{C}(\epsilon,i_1,but) \wedge \epsilon(e) \equiv \texttt{true}) \\
 & & \vee (\mathcal{C}(\epsilon,i_2,but) \wedge \epsilon(e) \equiv \texttt{false}) \\
\mathcal{C}(\epsilon,i_1\ ;\ i_2,but) & = & \mathcal{C}(\epsilon,i_1,but) \vee (\mathcal{C}(\epsilon,i_1,\texttt{refuse}) \wedge \mathcal{C}(\epsilon,i_2,but)) \\
\mathcal{C}(\epsilon,i,but) & = & \top \textit{ si } i = but, \bot \textit{ sinon}
\end{array}$$

Dans l'ensemble de contraintes résultant de l'exécution de cet algorithme, on propage les instanciations de variables et applique les équations de l'ancrage formel pour simplifier les contraintes.

En pratique, cette simplification est faite durant l'extraction proprement dite des contraintes, pour permettre la détection le plus tôt possible d'ensembles insatisfiables de contraintes (et dont de chemins impossibles dans le flot de contrôle) afin de les supprimer.

Exemple 16. On considère comme exemple le programme résultant de la compilation du problème de filtrage $g(x,b)$ avec $x \in \mathcal{X}$ décrit dans la sous-section 3.3.4, qui consiste

en un programme $\pi_{g(x,b)}$ dans lequel $\mathtt{s}$ est la variable d'entrée. On notera $\pi_{g(x.b)_1}$, $\pi_{g(x.b)_2}$, $\pi_{g(x.b)_3}$ et $\pi_{g(x.b)_4}$ les sous parties de $\pi_{g(x,b)}$.

$$
\begin{array}{l}
\pi_{g(x,b)}(\mathtt{s}) \triangleq \\
\quad \mathtt{if}(\mathtt{is\_fsym}(\mathtt{s}, \ulcorner g \urcorner), \\
\qquad \pi_{g(x.b)_1} \left| \begin{array}{l} \mathtt{let}(x_1, \mathtt{subterm}_g(\mathtt{s}, 1), \\ \quad \mathtt{let}(x_2, \mathtt{subterm}_g(\mathtt{s}, 2), \\ \qquad \pi_{g(x.b)_3} \left| \begin{array}{l} \mathtt{let}(x, x_1, \\ \pi_{g(x.b)_4} \end{array} \right| \mathtt{if}(\mathtt{is\_fsym}(x_2, \ulcorner b \urcorner), \mathtt{accept}, \mathtt{refuse})))), \end{array} \right| \pi_{g(x.b)_2} \\
\quad \mathtt{refuse} \\
\quad)
\end{array}
$$

On initialise l'algorithme d'extraction des contraintes de la définition 26 avec l'environnement $\epsilon = [\mathtt{s} \leftarrow t]$ et `accept` comme but.

Une fois cette contrainte calculée comme le montre la figure 3.7, on peut l'utiliser pour montrer la validité de la compilation du filtrage. La substitution nous donne les valeurs des variables, tandis que les contraintes associées nous donnent l'« allure » de ces valeurs.

3.5.2. Simplification des contraintes

L'utilisation de l'algorithme de la définition 26 pour collecter les contraintes et les substitutions générées par un programme `PIL` produit de très grosses formules, contenant beaucoup de constantes $\top$ et $\bot$, et donc il est possible de simplifier cette formule comme une formule booléenne.

On applique le système de règles suivant, avec une stratégie *leftmost-innermost*[1] :

$$
\begin{array}{rcl}
\bot \wedge x & \rightarrow & \bot \\
x \wedge \bot & \rightarrow & \bot \\
\top \vee x & \rightarrow & \top \\
x \vee \top & \rightarrow & \top \\
\bot \vee x & \rightarrow & x \\
x \vee \bot & \rightarrow & x \\
\neg \top & \rightarrow & \bot \\
\neg \bot & \rightarrow & \top
\end{array}
$$

Ce système de réécriture simplifie les contraintes booléennes pour obtenir des contraintes simplifiées, qui ont maintenant pour notre exemple $\pi_{g(x,b)}$ la forme :

$$
\begin{array}{rc}
 & \mathtt{is\_fsym}(\mathtt{s}, \ulcorner g \urcorner) \equiv \mathtt{true} \\
\wedge & x_1 = \mathtt{subterm}_g(\mathtt{s}, 1) \\
\wedge & x_2 = \mathtt{subterm}_g(\mathtt{s}, 2) \\
\wedge & x = x_1 \\
\wedge & \mathtt{is\_fsym}(\mathtt{subterm}_g(\mathtt{s}, 2), \ulcorner b \urcorner) \equiv \mathtt{true}
\end{array}
$$

[1] On verra dans le chapitre 4 comment on aurait pu utiliser Gom pour faire en sorte que l'application de cette simplification soit implicite et intégrée à la structure de données

$$
\begin{array}{lll}
& \mathcal{C}([\mathtt{s} \leftarrow t], \pi_{g(x,b)}, \mathtt{accept}) & \\
= & \quad \mathcal{C}([\mathtt{s} \leftarrow t], \pi_{g(x,b)_1}, \mathtt{accept}) \wedge \mathtt{is\_fsym}(t, \ulcorner g \urcorner) \equiv \mathtt{true} & \\
\vee & \quad \mathcal{C}([\mathtt{s} \leftarrow t], \pi_{g(x,b)_1}, \mathtt{accept}) \wedge \mathtt{is\_fsym}(t, \ulcorner g \urcorner) \equiv \mathtt{false} & \\
= & \quad \mathcal{C}([\mathtt{s} \leftarrow t][x_1 \leftarrow \mathtt{subterm}_g(t,1)], \pi_{g(x,b)_2}, \mathtt{accept}) \wedge \mathtt{is\_fsym}(t, \ulcorner g \urcorner) \equiv \mathtt{true} & \\
\vee & \quad \bot \wedge \mathtt{is\_fsym}(t, \ulcorner g \urcorner) \equiv \mathtt{false} & \\
= & \quad \mathcal{C}([\mathtt{s} \leftarrow t][x_1 \leftarrow \mathtt{subterm}_g(t,1)][x_2 \leftarrow \mathtt{subterm}_g(t,2)]\pi_{g(x,b)_3}, \mathtt{accept}) & \\
& \quad \wedge \mathtt{is\_fsym}(t, \ulcorner g \urcorner) \equiv \mathtt{true} & \\
\vee & \quad \bot \wedge \mathtt{is\_fsym}(t, \ulcorner g \urcorner) \equiv \mathtt{false} & \\
= & \quad \mathcal{C}([\mathtt{s} \leftarrow t][x_1 \leftarrow \mathtt{subterm}_g(t,1)][x_2 \leftarrow \mathtt{subterm}_g(t,2)][x \leftarrow \mathtt{subterm}_g(t,1)], & \\
& \quad \pi_{g(x,b)_4}, \mathtt{accept}) & \\
& \quad \wedge \mathtt{is\_fsym}(t, \ulcorner g \urcorner) \equiv \mathtt{true} & \\
\vee & \quad \bot \wedge \mathtt{is\_fsym}(t, \ulcorner g \urcorner) \equiv \mathtt{false} & \\
= & \quad \mathcal{C}([\mathtt{s} \leftarrow t][x_1 \leftarrow \mathtt{subterm}_g(t,1)][x_2 \leftarrow \mathtt{subterm}_g(t,2)][x \leftarrow \mathtt{subterm}_g(t,1)], & \\
& \quad \mathtt{accept}, \mathtt{accept}) & \\
& \quad \wedge \mathtt{is\_fsym}(\mathtt{subterm}_g(t,2), \ulcorner b \urcorner) \equiv \mathtt{true} & \\
\vee & \quad \mathcal{C}([\mathtt{s} \leftarrow t][x_1 \leftarrow \mathtt{subterm}_g(t,1)][x_2 \leftarrow \mathtt{subterm}_g(t,2)][x \leftarrow \mathtt{subterm}_g(t,1)], & \\
& \quad \mathtt{refuse}, \mathtt{accept}) & \\
& \quad \wedge \mathtt{is\_fsym}(\mathtt{subterm}_g(t,2), \ulcorner b \urcorner) \equiv \mathtt{false} & \\
& \quad \wedge \mathtt{is\_fsym}(t, \ulcorner g \urcorner) \equiv \mathtt{true} & \\
\vee & \quad \bot \wedge \mathtt{is\_fsym}(t, \ulcorner g \urcorner) \equiv \mathtt{false} & \\
= & \quad \mathcal{C}([\mathtt{s} \leftarrow t][x_1 \leftarrow \mathtt{subterm}_g(t,1)][x_2 \leftarrow \mathtt{subterm}_g(t,2)][x \leftarrow \mathtt{subterm}_g(t,1)], & \\
& \quad \mathtt{accept}, \mathtt{accept}) & \\
& \quad \wedge \mathtt{is\_fsym}(\mathtt{subterm}_g(t,2), \ulcorner b \urcorner) \equiv \mathtt{true} & \\
\vee & \quad \bot \wedge \mathtt{is\_fsym}(\mathtt{subterm}_g(t,2), \ulcorner b \urcorner) \equiv \mathtt{false} & \\
& \quad \wedge \mathtt{is\_fsym}(t, \ulcorner g \urcorner) \equiv \mathtt{true} & \\
\vee & \quad \bot \wedge \mathtt{is\_fsym}(t, \ulcorner g \urcorner) \equiv \mathtt{false} &
\end{array}
$$

FIG. 3.7: Extraction des contraintes pour $\pi_{g(x,b)}$

Ces contraintes sont ensuite simplifiées en utilisant les définitions de l'ancrage formel, données en définition 15 :

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

Les égalités de l'ancrage formel peuvent être orientées pour être utilisées comme un système de réécriture. Le but ici est de transformer une contrainte donnant des informations à propos des objets manipulés par le programme en une contrainte donnant des informations sur les termes algébriques utilisés au niveau du code source (ceux auquel

pense l'utilisateur lorsqu'il utilise Tom, par exemple).

$$\begin{aligned}
\texttt{eq}(\ulcorner t_1 \urcorner, \ulcorner t_2 \urcorner) &\rightarrow \ulcorner t_1 = t_2 \urcorner \\
\texttt{is_fsym}(\ulcorner t \urcorner, \ulcorner f \urcorner) &\rightarrow \ulcorner \mathcal{S}ymb(t) = f \urcorner \\
\texttt{subterm}_f(\ulcorner t \urcorner, \ulcorner i \urcorner) &\rightarrow \ulcorner t_{|i} \urcorner \text{ if } \mathcal{S}ymb(t) = f
\end{aligned}$$

L'application de ce système de réécriture aux contraintes générées pour notre programme $\pi_{g(x,b)}$ produit alors la contrainte :

$$\begin{aligned}
&\mathcal{S}ymb(\mathtt{s}) = g \\
\wedge\quad & x_1 = \mathtt{s}_{|1} \\
\wedge\quad & x_2 = \mathtt{s}_{|2} \\
\wedge\quad & x = x_1 \\
\wedge\quad & \mathcal{S}ymb(\mathtt{s}_{|2}) = b
\end{aligned}$$

On peut alors utiliser les définitions des notions de sous-terme et de symbole de tête associées à l'algèbre de termes, comme définis dans la sous-section 3.3.4 pour prouver l'équivalence avec le problème de filtrage.

3.6. Autres extensions : notation crochet, alias

3.6.1. Notations crochet et souligné

Le langage Tom permet l'utilisation de la notation _ dans un filtre en lieu et place d'une variable. Ceci dénote un sous terme qui est présent (et cela peut être vérifié au moment de la compilation, en utilisant l'arité du symbole étant filtré), mais dont on ne compte pas utiliser la valeur. La notation dite « crochets » utilise quand à elle les étiquettes associées aux différents sous-termes d'un symbole pour permettre au programmeur de spécifier les seuls sous-termes (ou champs) dont la valeur est signifiante. Ces deux notations sont équivalentes, mais la seconde est généralement préférée, car elle autorise plus de flexibilité lors de l'évolution du programme, ainsi qu'il est montré dans la section 2.2.4.

Une manière naïve de compiler ces constructions est de remplacer ces « *trous* » dans les motifs par des variables utilisant un nom de variable frais. Cela permettrait à notre méthode de validation de ne pas avoir à être adaptée, mais possède un inconvénient : il peut être arbitrairement coûteux de calculer la valeur d'un sous-terme (car l'ancrage peut utiliser des fonctions dont le calcul peut être difficile). Ainsi, il est préférable de tout simplement ne pas calculer la valeur de ces sous-termes : le programme compilé correspondant au filtre ne fera aucune mention de ces sous-termes négligés.

Cependant, le théorème à prouver pour obtenir la correction du programme résultant de la compilation d'un tel filtre devra tenir compte de ces *trous*. Les motifs utilisant la notation « crochet » doivent être convertis en motifs utilisant _, en utilisant l'arité et les positions des différents champs utilisés dans le filtre. Ensuite, les « *trous* » du filtre devront être remplacés par des variables fraîches quantifiées existentiellement.

Exemple 17. Le filtre $g(\_, b)$ peut être compilé en un programme $\pi_{g(\_,b)}$. Afin de prouver la validité de cette compilation, on doit montrer :

$$\forall t \in \mathcal{T}(\mathcal{F}), \mathcal{C}\pi_{g(\_,b)}(t) \Leftrightarrow \exists \epsilon' \in \mathcal{E}nv, \exists z \in \mathcal{T}(\mathcal{F}), \Phi(\epsilon')(g(z, b)) = t$$

Cette formulation est utile, car elle permet aux preuves automatiques de ne pas être plus difficiles à produire : la définition de $\mathcal{S}ymb(t) = g$ produit lorsqu'elle est utilisée un témoin de l'existence de chaque sous-terme, et donc montrer l'existence de z dans le cas de notre exemple n'est pas chose difficile, car il n'y a pas d'autre contrainte portant sur z.

3.6.2. Alias

Les motifs de Tom peuvent contenir des alias pour certains sous-termes, notés `nom@`. Ceux-ci sont utiles lorsque l'on écrit des motifs compliqués, ou encore pour autoriser la déclaration d'une variable dont la forme du filtre assure que sa valeur aura une certaine forme.

Le filtre dans le théorème à prouver devra alors être abstrait au niveau du sous-terme sur lequel porte l'alias : il consistera dont en une conjonction de problèmes de filtrage sans alias.

Exemple 18. Le filtre $f(name@Name(n))$ utilise un alias pour instancier la variable *name* avec une valeur qui est nécessairement de la forme $Name(n)$, sans avoir a recourir à une imbrication de motifs. La propriété à prouver alors pour montrer que $\pi_{f(name@Name(n))}$ est une compilation correcte de ce filtre est :

$$\begin{gathered}\forall t \in \mathcal{T}(\mathcal{F}), \\ \mathcal{C}\pi_{f(name@Name(n))}(t) \\ \Leftrightarrow \\ \exists \epsilon' \in \mathcal{E}nv, \exists z \in \mathcal{T}(\mathcal{F}), \Phi(\epsilon')(f(name)) = t \wedge name = Name(n)\end{gathered}$$

3.6.3. Filtrage de liste

Telle que présentée dans ce chapitre, la méthode de validation ne permet pas de certifier les motifs utilisant le filtrage associatif de Tom. En effet, le langage `PIL` utilisé n'est pas assez expressif pour exprimer les itérations sur les différentes solutions des motifs associatifs. Pour cela, le compilateur Tom utilise dans son langage intermédiaire une instruction `while`, qui exécute une certaine instruction tant qu'une condition booléenne est vraie. La méthode d'extraction des contraintes présentée en section 3.5 étendue simplement à une boucle `while` conduirait à un ensemble infini de contraintes.

D'autre part, le filtrage de liste tel qu'il est utilisé dans Tom manipule des listes, ou opérateurs variadiques, ce qui rend l'établissement d'une correspondance entre filtrage de liste de Tom et filtrage associatif algébrique plus difficile. Nous décrirons les détails des liens entre les objets du filtrage associatif algébrique et les opérateurs variadiques manipulés lors du filtrage « *de liste* » de Tom dans le chapitre 5.

3.7. Implémentation et intégration

La méthode décrite jusqu'ici, ainsi que ses extensions permettant de supporter les particularités du filtrage dans le langage Tom à été implantée et intégrée au compilateur du langage Tom.

Les algorithmes présentés font un usage intensif du filtrage, afin d'examiner dans une première partie l'arbre de syntaxe d'un programme Tom, pour extraire et analyser les constructions de filtrage `%match`, ainsi que pour examiner le code `PIL` généré par le compilateur. De plus, l'algorithme d'extraction des contraintes et de substitutions s'exprime naturellement comme un système de réécriture. Il est alors naturel d'utiliser le langage Tom lui même pour implanter le vérificateur de la validité de la compilation du filtrage.

Le compilateur Tom étant lui-même écrit en utilisant Tom, et *bootstrappé*, nous avons intégré la procédure de validation au compilateur Tom, où elle s'applique sur le langage intermédiaire du compilateur, qui est une extension du langage `PIL`, contenant de l'information importante pour le travail du compilateur, comme les origines des différentes constructions, permettant de produire des messages d'erreur utiles pour le programmeur, mais que l'on néglige lors de la validation.

Étant donnée une construction `%match`, le compilateur traduit les motifs en un ensemble d'instructions `PIL` utilisant l'ancrage formel défini pour les sortes algébriques manipulées. On suppose en pratique cet ancrage correct, c'est-à-dire tel que les propriétés structurelles des termes sont préservées. On verra dans le chapitre 4 comment un générateur automatique d'ancrages formels ainsi que d'implémentation pour les structures de termes peut être utilisé.

Le composant de validation de la compilation à été intégré au compilateur Tom, et génère, pour chaque construction de filtrage m, ainsi que sa contrepartie compilée π_m les ensembles de contraintes $\mathcal{C}\pi_m$ pour le programme et les contraintes de filtrage. Ces contraintes sont ensuite envoyées à un prouveur extérieur, afin de prouver leur équivalence. Nous avons utilisé le prouveur Zenon car non content d'être automatique et efficace, il génère lorsqu'il arrive à prouver le théorème un terme de preuve dans le calcul des constructions qui constitue un témoin vérifiable de la preuve. Ce témoin peut ensuite être vérifié par l'assistant de preuve Coq, donnant une garantie forte de la preuve trouvée par Zenon. Cette étape est essentielle, car elle permet à l'utilisateur du programme généré de vérifier lui-même la preuve : non seulement l'outil de validation lui certifie que le compilateur à fait son travail de manière correcte, mais il lui fournit un témoin vérifiable de ce fait.

L'outil de validation est intégré à l'architecture du compilateur Tom, comme on peut le voir dans la figure 3.8, mais on peut remarquer qu'il ne requiert aucun support particulier de la part du compilateur : les contraintes de filtrage sont extraites de l'arbre de syntaxe abstrait produit par le parseur, et les contraintes du programme sont extraites à partir du programme `PIL` produit par le compilateur ou d'autre composants, comme l'optimiseur par exemple. On voit alors le compilateur comme une boite opaque, ce qui autorise l'application de tout type d'optimisation tant que du code `PIL` est généré. En particulier, un des intérêts de cette approche est de permettre la validation de code correspondant à

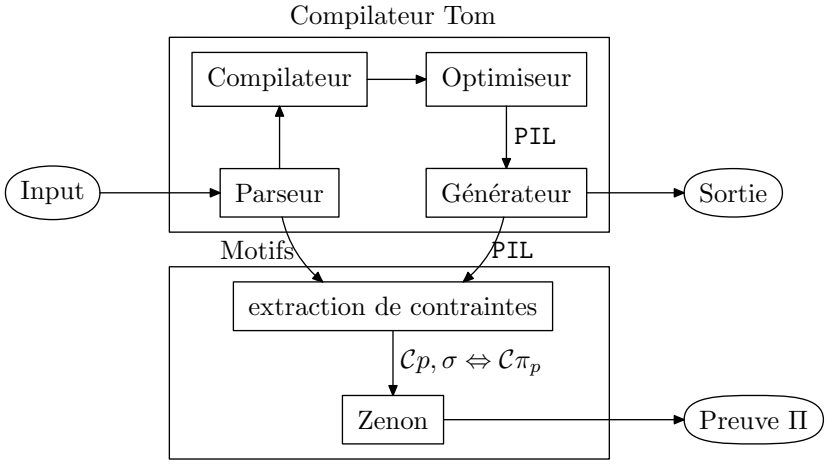


FIG. 3.8: Architecture globale de Tom avec le vérificateur

un algorithme optimisé « *many-to-one* » factorisant les tests des différents motifs d'une construction multi-motif.

Cependant, cette intégration ne prend en compte que le code intermédiaire généré par le compilateur, et ignore le parseur et le générateur de code. Dans le cadre de Tom, le générateur de code effectue une simple traduction *une-à-une* des instructions du langage `PIL` vers le langage hôte. Cette transformation est assez directe pour que l'on puisse lui accorder notre confiance, mais il serait possible de prouver cette transformation en utilisant une formalisation stricte de la sémantique du langage hôte.

Cette implémentation à été testée sur de nombreux exemples. Aussi, elle est intégrée aux jeux de tests du compilateur Tom, qui permettent de détecter tout régression dans le fonctionnement du compilateur. Les tests utilisant le vérifieur permettent non seulement de s'assurer que le code qui est effectivement testé par le test unitaire fonctionne sur l'entrée du jeu de test, mais aussi que ce code est correct, et fonctionnera sur toute entrée : cela constitue donc un test de bien meilleure qualité.

La procédure de validation a aussi été appliquée au code du compilateur Tom lui-même, puisque le compilateur Tom est lui-même un programme Tom. Environ 200 motifs ont été extraits (les motifs utilisant l'associativité n'étant pas traités) après exécution du parseur. L'étape d'extraction des contraintes produit à partir des programmes générés environ 840 contraintes ($\mathcal{C}\pi_p$), après simplification booléenne. De nombreuses contraintes sont alors des tautologies de la forme : $\epsilon(\mathtt{subterm}_g(\mathtt{s}, 1)) \equiv \mathtt{s}_{|1}$. Après simplification en utilisant les équations de l'ancrage formel, on obtient environ 275 contraintes élémentaires (en utilisant plus de 2500 pas de $\equiv$-équivalence). Cette extraction, ainsi que la traduction des contraintes et obligations de preuve dans le formalisme d'entrée de ZENON accroissent le temps de compilation du projet « compilateur Tom » de moins de 20%.

4. Structure de données sûre

> Il y a deux sortes d'arbres : les hêtres et les non-hêtres.
>
> *(Raymond Queneau)*

> Vna cosa es alabar la disciplina, y otra el darse con ella, y, en efeto, *del dicho al hecho ay gran trecho*.
>
> *(Miguel de Cervantes)*

Gom est un langage permettant de décrire de manière concise la structure d'arbres de syntaxe abstraits, et de générer pour ceux-ci une implémentation dans le langage Java (mais cela pourrait être étendu à tout langage orienté objet). Cette implémentation présente du typage fort, garantissant que les objets créés sont valides, ainsi que du partage maximal, rendant ces structures très efficaces.

De plus, Gom fournit un moyen de spécifier des invariants que la structure de données générée devra respecter. Le respect de ces invariants est assuré par construction des objets, fournissant l'assurance qu'aucun objet ne contreviendra à ceux-ci. Ces invariants prennent la forme de fonctions calculant des formes canoniques sur les termes représentés : seuls les représentants canoniques des classes d'équivalence sont des objets valides.

4.1. Des arbres de syntaxe abstraits en Java

Les arbres de syntaxe abstraits (que l'on appellera AST dans la suite) sont la structure de donnée la plus commune lorsqu'il s'agit de représenter des structures en vue d'implanter des transformations de programmes, interpréteurs ou compilateurs, des transformations de documents ou d'implanter des algorithmes complexes, comme une inférence de type, qui est décidable en général ou une recherche de preuve, souvent indécidable.

D'autre part, les AST sont aussi le type de structure de donnée sur lequel reposent les notions de filtrage et de réécriture. L'objet même des langages basés sur ces notions est d'offrir les moyens d'écrire de manière concise, précise et sûre des transformations et manipulations d'AST. De tels langages fournissent généralement des constructions permettant de manipuler des AST qui sont une notion interne au langage, comme les types inductifs de Caml ou les termes de langages comme Elan, ASF+SDF ou Maude. Le langage Tom se démarque en mettant l'accent sur l'indépendance vis-à-vis des structures de données. Ainsi, en fournissant un *mapping* pour un ensemble de constructions permet-

tant de représenter une structure d'arbre, on peut utiliser les fonctionnalités du langage sur cette structure *a priori* indépendante du langage Tom. Cette notion de *mapping* est présentée dans la section 2.2.2. Un tel mapping est appelé *ancrage formel* lorsqu'il permet effectivement de représenter une structure de termes, comme défini en section 3.3.1. On confondra souvent par la suite les notions de structure de termes et AST. En effet, l'AST n'est que la représentation concrète d'une structure de termes.

Cependant, les langages hôtes de Tom n'intègrent pas de notion d'AST par défaut, mais fournissent des constructions de plus bas niveau (les structures de C ou les classes de Java) permettant de définir des structures de données, et en particulier des structures arborescentes. Ces primitives offrent une grande souplesse au programmeur, afin d'implanter des structures de données potentiellement compliquées. Toutefois, lorsqu'il s'agit de manipuler des structures arborescentes comme les AST, elle ne sont pas suffisantes. Pour le programmeur Java, il est courant d'utiliser le *patron de conception* [GHJV93] « Composite » pour représenter des structures arborescentes. Il est pourtant facile de mal utiliser ce patron, et la possibilité d'erreur est accrue lorsque l'AST à manipuler correspond à un langage d'arbre complexe, comprenant de nombreux constructeurs.

4.1.1. Une bibliothèque générique pour représenter des arbres : ATerm

Une solution au problème de la représentation des AST est de définir une structure générique d'arbre, avec pour seuls constructeurs les *feuilles* et les *nœuds*, qui eux-mêmes contiennent une liste éventuellement vide de nœuds.

L'implantation de ces arbres génériques est alors partagée par toutes les applications utilisant une telle structure. Il est alors raisonnable d'envisager des optimisations complexes pour rendre ces structures efficaces, ainsi que de fournir des fonctionnalités comme la sérialisabilité (pouvoir stocker des arbres dans des fichiers et les relire, ou les transmettre sur le réseau).

C'est l'approche que suit la bibliothèque des ATerm [BJKO00], en fournissant une implantation générique des termes algébriques robuste et très efficace. Les arbres sont alors représentés de manière générique, en considérant chaque nœud comme l'application d'un symbole de tête à une liste d'arguments. Les opérations sur les arbres consistent à modifier le symbole de tête ou la liste d'arguments. Un arbre, ou terme, comme $f(g(a), b)$ est représenté par un objet qui peut s'écrire :

```
ATermAppl(
  AFun("f"),
  [ATermAppl(AFun("g",[ATermAppl(AFun("a"),[])])),
   ATermAppl(AFun("b"),[])]
)
```

avec `ATermAppl` dénotant l'application d'un symbole de fonction à une liste de terme, `AFun` permettant de construire un symbole de fonction à partir de son nom, et `[]` représentant les listes de termes.

Cette bibliothèque implémente le partage maximal des sous-termes qui optimise l'utilisation mémoire. Cette implantation réalisée au niveau de la bibliothèque générique peut alors profiter à toutes les structures de termes manipulées par l'application.

Partage maximal

Le partage maximal de sous-termes permet de s'assurer qu'une seule instance de chaque sous-terme (ou sous-arbre) existe dans la mémoire. Cette technique, souvent mise en œuvre par *hash consing* est bien connue dans les domaines de la programmation fonctionnelle et de la réécriture de termes, car les exécutions dans de tels systèmes conduisent souvent à une redondance importante dans les objets créés. Le terme *hash consing* provient de l'héritage du langage Lisp, dans lequel la seule fonction d'allocation est *cons* et le partage est obtenu généralement en utilisant une table de hachage globale [All78]. Le *hash consing* assure alors que si le même nœud est construit deux fois, une seule instance de celui-ci existe en mémoire, et lors du second essai de création, un pointeur sur le nœud déjà existant est retourné. Si le calcul introduit une certaine quantité de redondance, alors le gain est significatif, tant en espace mémoire qu'en temps de calcul, comme cela à été montré à de nombreuses reprises [Gou94]. Ainsi, cette technique permet entre autres d'avoir un test d'égalité dans les arbres qui s'effectue en temps constant, par simple comparaison de pointeurs plutôt qu'un test d'égalité récursif. Non seulement le partage maximal permet d'optimiser l'usage de la mémoire, mais cela permet d'accélérer les opérations fondamentales sur la structure de données de plusieurs ordres de magnitude.

Le *hash consing* lui-même bénéficie du partage maximal des sous-termes, car lors de la création d'un nouveau terme, ses sous-termes sont déjà partagés.

Une des contributions de la bibliothèque ATerm pour Java est de fournir une bibliothèque générique (`shared-objects`) implémentant le partage maximal pour tout type d'objets se conformant à une certaine interface, qui impose de pouvoir calculer une somme de hachage, ainsi que de pouvoir vérifier l'équivalence entre deux objets partageable et de pouvoir les cloner. Ainsi, le code généré par ApiGen permet le partage maximal, par l'intermédiaire de la bibliothèque ATerm, qui elle-même repose sur `shared-objects`.

Cette bibliothèque fournit un point d'entrée pour gérer et administrer des objets existants, les objets partagés. Il est alors naturel d'utiliser le patron de conception *Factory* pour encapsuler une telle administration. Cette fabrique maintient une table de hachage permettant de vérifier rapidement si un objet partageable donné est déjà connu. Le client de la fabrique donne à cette fabrique un *prototype* de l'objet partagé qu'il souhaite voir partagé, et obtient un objet équivalent, qui est partagé.

Inconvénients

L'approche utilisant une implantation générique des arbres impose naturellement de n'avoir qu'un seul type pour tous les arbres. Les données manipulées par le programme sont alors toutes du même genre, ce qui peut conduire à des confusions dans un pro-

gramme manipulant des données de genres différents, car le typage du langage ne permettra pas de distinguer les mauvais usages consistant à utiliser un type de données plutôt que l'autre. Cela rend le programme plus difficile à lire, relire et corriger, des parties disjointes du programme semblant manipuler des données d'un même genre, alors qu'elles sont différentes.

Il est alors possible de créer des objets qui n'appartiennent pas aux langages de termes que l'on considère dans l'application, ou d'utiliser des données d'une certaine sorte comme arguments d'une fonction qui attend des arguments d'un autre genre. D'autre part, cette absence de typage ne permet pas de valider les données manipulées vis-à-vis des arités des différents constructeurs, ni vis-à-vis des types des arguments de ces constructeurs, rendant la détection d'erreurs de ce type indécelable à la compilation du programme.

C'est pourtant l'approche générique qui est généralement utilisée lors de la compilation de langages fonctionnels ou logiques, comme ML ou Elan. En effet, dans ces langages, le compilateur peut vérifier le bon typage des différentes constructions, garantissant l'absence d'erreurs de ce genre lors de l'exécution. La signature de termes utilisée est alors typée, mais pas l'implantation, qui est générique (et donc n'est écrite qu'une seule fois).

4.1.2. Un générateur d'implantation typée : ApiGen

Dans le cadre de Tom, le développeur travaille directement avec l'implantation des termes, comme nous l'avons vu en section 2.3, ce qui rend cette approche difficile : utiliser une bibliothèque de termes générique implique ne pas avoir de typage au niveau des termes.

Il devient alors important de considérer une implémentation fortement typée des AST, donnant des types distincts aux différentes sortes d'arbres, permettant de distinguer les différents aspects d'un programme : l'AST représentant une instruction aura un type spécifique, autre que celui d'un AST représentant une expression. Cela permet au compilateur de détecter statiquement des erreurs de programmation dans l'outil de transformation, et ceci le plus tôt possible.

On peut rapprocher cela de l'utilisation des types inductifs en programmation fonctionnelle : leur typage fort, ainsi que celui des fonctions permet de détecter lors de la compilation toutes les utilisations invalides des différents types de données, et donc de réduire le risque d'erreurs à l'exécution du programme. Ce typage doit être effectué dans l'environnement Tom au niveau du langage hôte directement.

ApiGen [JO04, BMV05] est un outil qui génère automatiquement une implémentation d'arbres de syntaxe abstraits dans les langages C et Java. Il prend comme entrée une définition concise du type de données abstrait (une description proche de la description des types inductifs de ML) et génère du code C ou Java pour ces arbres de syntaxe abstraits qui est fortement typé et utilise le partage maximal pour l'efficacité en mémoire et pour les tests d'égalité. L'implémentation générée pour une description donnée contient toutes les fonctionnalités souhaitées, est optimisée et possède une interface (API) typée, claire et compréhensible. En cela déjà elle se distingue d'une implantation écrite à la

main, ayant été modifiée de part et d'autre pour introduire des optimisations (ou des bogues), et dont l'interface elle-même n'est plus cohérente à cause d'évolutions qui n'ont pas été transposées à d'autres parties de l'implémentation.

4.2. Discussion

Le travail présenté dans ce chapitre s'appuie sur les acquis d'ApiGen, et propose une extension donnant plus d'efficacité ainsi qu'une plus grande expressivité et souplesse.

ApiGen était utilisé (avant d'être remplacé par Gom) pour générer les implémentations des AST utilisées par le compilateur Tom. À ce titre, j'ai eu l'occasion de participer à la maintenance et à l'évolution du compilateur ApiGen, ainsi qu'à celle du code généré par celui-ci. Ce sont les problèmes rencontrés lors de l'utilisation d'ApiGen dans le compilateur Tom ou dans des exemples d'utilisation, ainsi que certaines des limitations d'ApiGen qui ont conduit à la conception du langage Gom et à l'écriture de son compilateur.

L'outil Gom a été développé pour répondre à des besoins rencontrés dans le cadre de la manipulation d'arbres dans le langage Tom. Il aurait été possible, plutôt que de développer un langage distinct, d'intégrer la représentation d'arbres au sein du compilateur Tom lui-même. Cette approche permettrait une simplification du langage (plus de besoin d'ancrage formel), ainsi que certaines optimisations, mais au prix de la souplesse d'utilisation ainsi que de l'évolutivité des différents outils. D'autre part, les caractéristiques du langage font qu'il a un intérêt hors du cadre du langage Tom, et peut être utile et utilisable indépendamment.

4.2.1. Caractéristiques d'ApiGen pour Java

Une interface typée

ApiGen fournit une interface typée pour la manipulation d'une signature algébrique comportant des sortes d'opérateurs différentes. À chaque sorte correspond un type Java, ce qui permet au compilateur de vérifier la bonne formation des différents termes manipulés par le programme à la compilation durant la vérification de typage.

Partage maximal

L'implantation générée met en œuvre le *hash consing*, présenté en section 4.1.1, afin d'offrir du partage maximal en mémoire à l'utilisateur de la structure de données.

Compatibilité avec ATerm

ApiGen à été conçu pour être entièrement compatible avec la bibliothèque des ATerm, tout en fournissant une interface fortement typée. C'est-à-dire qu'un arbre représenté par une structure typée générée par ApiGen est lui-même un `ATerm` et les types que permet de manipuler ApiGen sont tous des sous-types de `ATerm`. Ce choix est dû en partie à la volonté d'avoir une architecture de la version d'ApiGen pour Java qui soit similaire à

celle qui existait pour le langage C, et qui utilisait la bibliothèque ATerm pour C, ainsi que pour bénéficier de l'implantation du partage maximal de la bibliothèque ATerm.

Une raison plus essentielle de ce choix est le fait qu'il était nécessaire de pouvoir utiliser une représentation générique et non typée des arbres pour exprimer des algorithmes génériques et réutilisables, comme les différentes traversées d'arbres.

4.2.2. Améliorations possibles

Efficacité

Les objets générés par ApiGen utilisent un représentation interne `ATerm` pour représenter l'arbre de manière générique. De cette architecture en couches découle le fait que le code généré par ApiGen, bien que spécifique à la signature spécifiée pour ce qui est du typage de l'interface, ne peut pas être optimisé en fonction de cette signature, car il délègue l'implémentation à une bibliothèque générique.

Ceci impose une gestion lourde des types *builtin*, comme les entiers et les chaînes de caractères, ainsi que de multiples appels de fonction et traductions entre le formalisme exposé par l'interface et la représentation générique sous-jacente. En faisant reposer l'implémentation directement sur la bibliothèque `shared-objects`, il est possible de spécialiser le code généré pour une signature, et d'obtenir de substantiels gains de performances. Cela permet de plus de réduire la taille en mémoire des objets manipulés, donnant une implantation plus efficace en temps et en mémoire.

C'est l'introduction de la bibliothèque de stratégies décrite dans le chapitre 6 qui nous permet de supprimer le besoin d'une représentation générique non typée pour implanter des algorithmes génériques.

Symboles de liste

Une limitation d'ApiGen est qu'il n'est pas possible de définir plusieurs opérateurs de liste ayant même domaine. De plus, le domaine d'un opérateur de liste ne ne peut pas non plus contenir d'opérateur algébrique. Ainsi, il n'est pas possible de définir les opérateurs `et` et `ou` comme des opérateurs représentés par des listes de booléens.

De plus, ApiGen interdit la définition de listes ayant même domaine et co-domaine. Cela interdit alors de représenter la liste d'instructions dans l'AST d'un langage de programmation comme une instruction elle-même, en imposant l'utilisation d'opérateur intermédiaires convertissant ces listes dans des instructions.

Ces constructions sont communes dans les spécifications algébriques, et il est alors important de pouvoir passer outre ces limitations.

Invariants de structure de donnée

Aucun des langages ou bibliothèques présentés jusque là ne fournit de moyen de spécifier des invariants que la structure de données devrait respecter. En effet, ils se contentent de fournir un support pour un type de données abstrait, et se focalisent sur l'indépen-

dance des données vis-à-vis de la machine et du langage, ainsi que sur l'efficacité de l'implantation générée.

Cependant, lorsque l'on utilise de telles structures de données, il est courant d'avoir besoin de spécifier une relation d'équivalence particulière entre les objets, et de travailler en faisant abstraction de cette équivalence.

Exemple 19. Un exemple d'une telle relation d'équivalence, pour laquelle on veut uniquement considérer un certain représentant canonique est la représentation des nombres rationnels. Un nombre rationnel est composé 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.

Besoin pour la compilation. Le domaine de la compilation constitue un deuxième exemple du besoin de tels invariants. Lorsque l'on considère les AST produits pour un programme donné que l'on doit compiler, il est habituel de procéder à la propagation des constantes. Toutes les expressions de l'AST qui ne contiennent que des constantes connues à la compilation et des opérations sur ces constantes sont évaluées, et la portion d'AST qui les représente est remplacée par la valeur alors calculée.

On peut envisager d'écrire une fonction qui traverse l'AST, et procède à cette propagation de constantes. Cependant, les différentes phases de compilation peuvent produire de nouvelles expressions constantes. Il est alors nécessaire de rappeler la fonction propageant les constantes. Cela impose au programmeur de toujours penser à la présence possible de sous-termes étant des expressions constantes non calculées, et celui-ci doit explicitement appeler cette fonction pour s'assurer de leur absence.

Faire de la propagation des constantes un invariant de la structure de données permet alors de s'assurer que la propagation est toujours effectuée, et donc libère le programmeur de la tâche d'expliciter tous les usages de cette fonction, ainsi que du risque d'oublier l'un de ceux-ci.

D'autres invariants peuvent être considérés. Par exemple, on peut décider de faire de la cohérence de type un invariant au niveau des AST. Alors, il sera impossible de créer un AST comportant par exemple deux utilisations de la même variable ou de la même fonction avec des types différents, permettant de détecter les erreurs lors de la transformation du programme au niveau de la construction fautive, et non dans une phase ultérieure.

Besoin pour la réécriture. Dans le contexte de la réécriture, il est souvent utile 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. Alors, les règles de réécriture ne pourront être appliquées qu'après que cette normalisation a été effectuée. Il est ainsi utile de lier l'implantation de ce système normalisant avec l'implémentation 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 de termes qui ne sont pas des formes normales pour ce système, facilitant ainsi l'implantation du moteur de réécriture ainsi que sa lisibilité.

On verra aussi dans le chapitre 5 comment cette normalisation permet de garantir des propriétés sur le filtrage dans le cadre du compilateur Tom, et d'implanter des structures et du filtrage associatifs.

4.2.3. Travaux en rapport

Le besoin de fournir des implémentations efficaces pour des types de données abstraits ou des AST dans les langages généralistes n'est pas neuf. Plusieurs systèmes ont été développés dans le but de fournir ce genre de fonctionnalités à partir d'une description concise.

ASN.1 [DF01] est un langage complexe pour décrire des structures arborescentes (par exemple, il y a plus d'une trentaine de types de chaînes différents) qui est indépendant de la machine et du langage utilisés. Ce système, comme les langages SGML et XML, résout le problème de la communication entre composants, mais sont complexes et souvent cryptiques pour le néophyte. D'autre part, le coût d'interprétation du support de ces langages à l'exécution est souvent élevé, à cause de cette complexité.

ASDL [WAK97] est un outil dont le but est de faciliter l'écriture de compilateurs. Il a été conçu dans le but de fournir des outils pour manipuler une représentation unique des AST pour des outils écrits dans différents langages de programmation. Il existe des implémentations pour C, C++, Java, Standard ML et Haskell. L'objectif de supporter autant de langages rend difficile d'introduire des fonctionnalités avancées dans la structure générée. En particulier, les structures alors générées ne supportent pas le partage maximal (car sinon, les versions Haskell et Standard ML ne pourraient pas utiliser le filtrage inclus dans ces langages).

JJForester [KV00] qui est un prédécesseur d'ApiGen pour Java[1], et dont les particularités principales sont le fait d'être connecté à un générateur de parseurs, ainsi que d'être le premier générateur à supporter JJTraveler [Vis01]. Cependant, il ne fournit pas de partage maximal. La description d'une syntaxe pour un langage donné est traduite en une définition de type de données abstrait pour les AST que le parseur produira. Le support de JJTraveler est aussi important, car il permet de décrire des traversées de termes de manière concise et élégante. C'est une fonctionnalité qui a été reprise dans ApiGen pour Java et Gom. Nous verrons au chapitre 6 comment nous avons étendu ce système pour obtenir une bibliothèque réutilisable et extensible de stratégies.

Pizza [OW97] ajoute entre autres choses des types de données abstraits à Java. C'est un langage à part entière, et non un langage dédié à la description de données. D'autre

[1] ApiGen pour C existait cependant déjà.

part, les types de données abstraits de Pizza ne supportent pas le partage maximal, ni la définition d'invariants.

Langage	Types	Partage maximal	Variadiques	Stratégies	Invariants
ATerm		✓		✓	
ApiGen	✓	✓		✓	
JJForester	✓			✓	
ASDL	✓				
XML			✓		
ASN.1					
Pizza	✓				
Gom	✓	✓	✓	✓	✓

4.3. Le langage Gom

On décrit ici le langage Gom et sa syntaxe. On s'intéressera dans cette section aux fonctionnalités de base du langage, c'est à dire la représentation efficace en Java de structures de données décrites par une signature algébrique.

On montrera aussi comment Gom interagit avec le langage Tom pour fournir un cadre efficace et cohérent pour la programmation à base de réécriture dans Java.

4.3.1. Définir des signatures

Le langage Gom est avant tout un langage permettant de définir de manière concise des signatures algébriques. Une signature algébrique décrit comment une structure arborescente doit être construite. Une telle description contient des *sortes* et des *opérateurs*. Les *opérateurs* définissent les différentes formes de nœuds d'une certaine *sorte* par leur nom et les noms et sortes de leurs sous-termes. ApiGen, XML Schema, les types inductifs de ML et ASDL sont de tels formalismes.

À cette notion basique de signature, on ajoute une notion de *module*, qui est un ensemble de sortes. Ceci permet de définir de nouvelles signatures en composant des signatures existantes. Cette composition est particulièrement utile lorsque l'on doit travailler avec des signatures de grande taille, comme peut l'être la signature des AST manipulés par un compilateur. Cela permet aussi de séparer les parties de la signature qui décrivent des aspects disjoints du langage, comme les expressions arithmétiques et les constructions de contrôle, augmente les possibilités de réutilisation et permet de créer des librairies de signatures.

La figure 4.1 présente une syntaxe simplifiée du langage de description de signatures Gom. **module** représente un lexème dans le langage, tandis que $\langle Module \rangle$ représente un non terminal du langage. La notation « * » exprime la répétition d'une construction (qui

$$
\begin{array}{lcl}
\langle GomGrammar\rangle & ::= & \langle Module\rangle \\
\langle Module\rangle & ::= & \textbf{module}\ \langle ModuleName\rangle\ [\langle Imports\rangle]\ \langle Grammar\rangle \\
\langle Imports\rangle & ::= & \textbf{imports}\ (\langle ModuleName\rangle)^* \\
\langle Grammar\rangle & ::= & \textbf{abstract syntax}\ (\langle TypeDefinition\rangle)^* \\
\langle TypeDefinition\rangle & ::= & \langle SortName\rangle\ \textbf{=}\ [|]\ \langle OperatorDef\rangle\ (|\ \langle OperatorDef\rangle)^* \\
\langle OperatorDef\rangle & ::= & \langle OperatorName\rangle\ \textbf{(}[\langle SlotDef\rangle\ (\textbf{,}\langle SlotDef\rangle)^*]\ \textbf{)} \\
 & | & \langle OperatorName\rangle\ \textbf{(}\ \langle SortName\rangle\textbf{*}\ \textbf{)} \\
\langle SlotDef\rangle & ::= & \langle SlotName\rangle\ \textbf{:}\ \langle SortName\rangle \\
\langle ModuleName\rangle & ::= & \langle Identifier\rangle \\
\langle SortName\rangle & ::= & \langle Identifier\rangle \\
\langle OperatorName\rangle & ::= & \langle Identifier\rangle \\
\langle SlotName\rangle & ::= & \langle Identifier\rangle
\end{array}
$$

FIG. 4.1: Syntaxe simplifiée de Gom

peut être absente), tandis que les constructions entre [] sont optionnelles. Dans cette syntaxe, on peut remarquer qu'un module peut importer des modules existants pour réutiliser les sortes et opérateurs qui y sont définis. Il n'est pas nécessaire de déclarer les sortes définies dans un module de manière particulière, mais une extension du langage permettant de cacher certaines définitions lors de l'importation du module peut être considérée. Une telle extension peut permettre de gérer plus simplement les problèmes qui se posent lorsqu'un module importe via deux modules différents des définitions de sortes de même nom, mais distinctes. Ceux-ci doivent sinon être désambiguïsés explicitement dans le module effectuant l'importation.

Une première version de Gom supportait une syntaxe très inspirée de la syntaxe de SDF [BHKO02b]. Cependant, celle-ci imposait beaucoup de redondance d'information, principalement parce que SDF est conçu pour décrire des arbres de parsing et des parseurs à l'aide de règles de production, qui peuvent être ambiguës, ainsi que pour permettre l'utilisation de la syntaxe concrète lors de l'élaboration des règles. La syntaxe de Gom a alors été simplifiée. La syntaxe actuelle est fortement inspirée de la définition des types inductifs dans les langages de la famille ML, qui est plus adaptée à la description des arbres de syntaxe abstraits.

Le non-terminal $\langle TypeDefinition\rangle$ décrit les définitions de sortes comme étant un ensemble d'opérateurs, chacun ayant un certain nombre d'arguments : les sous-termes. Ces définitions d'opérateurs sont précédées du lexème |, le premier | pouvant être omis. La définition des arguments d'un opérateur peut se faire de deux manières. On peut tout d'abord spécifier les arguments comme une liste (potentiellement vide) de champs $\langle SlotDef\rangle$, comprenant un nom ainsi qu'une sorte pour ce champ. Ils définissent les différents sous-termes de l'opérateur ainsi que les sortes auxquelles ces sous-termes appartiennent. Le nom associé à chaque sous-terme permet d'adresser ce sous-terme à la

manière du champ d'une structure, ainsi que l'utilisation de la notation *crochet* définie en 2.2.4 lorsque l'on utilise Tom pour manipuler de telles structures. D'autre part, on peut utiliser le symbole * (qui est différent du symbole * de répétition) pour définir un opérateur variadique, qui est l'une des particularités de Gom. Cette notation * est la même que dans [CJ03, Section 2.1.6], et peut être vue comme la définition d'une famille d'opérateurs de même nom avec des arités positives ou nulles.

Exemple 20. Considérons comme exemple la définition d'une signature Gom permettant de représenter des expressions booléennes.

```
module Boolean
abstract syntax
Bool = True()
     | False()
     | Not(b:Bool)
     | And(lhs:Bool,rhs:Bool)
     | Or(lhs:Bool,rhs:Bool)
BoolList = list(Bool*)
```

Cette description définit `True` et `False` comme des constantes de sorte `Bool` et `Not` comme un opérateur unaire ayant comme argument un sous-terme de sorte `Bool` que l'on appellera `b`, `And` et `Or` étant des opérateurs binaires.

L'opérateur `list` appartient à la sorte `BoolList`, et est défini comme un opérateur variadique, dont les arguments sont de sorte `Bool`. On peut donc avoir des opérateurs `list` vides, n'ayant aucun sous-terme, ainsi que des opérateurs `list` ayant n sous-termes, tous de sorte `Bool`.

Les termes suivants sont alors valides et de type `Bool` : `True()`, `False()`, `Not(False())` et `And(Not(False()),True())`.

Les termes `list()`, `list(True())` et `list(True(),False(),Not(True()))` sont valides et de type `BoolList`, tandis que le terme `list(True(),list())` n'est pas valide, `list()` étant de sorte `BoolList` alors que les sous-termes d'un opérateur `list` doivent être de sorte `Bool`.

4.3.2. Le code généré

À partir d'une définition de signature, le compilateur Gom (lui-même écrit en utilisant Tom et Gom, et bootstrappé) génère une implantation en Java du type de données décrit par cette signature. Une description détaillée des particularités du code alors généré est disponible dans le chapitre 6 du manuel de référence de Tom [BBK+06], en particulier les aspects liés au support de la bibliothèque de stratégies. Nous nous attacherons dans cette section à décrire l'interface de programmation concernant la structure de termes, ainsi que les aspects du code généré qui permettent de garantir les propriétés de cette structure, en occultant les aspects techniques concernant les entrées-sorties, le support de la bibliothèque de stratégies.

Interface de programmation d'applications

Afin d'illustrer l'interface de programmation d'applications (API) générée, on s'appuiera sur un exemple de signature un peu plus complet que celui de l'exemple 20.

L'interface ainsi générée est très proche de celle qui est générée pour une description similaire par l'outil ApiGen. En effet, le compilateur Tom, avant d'utiliser des structures générées par Gom comme AST, utilisait une implantation générée par ApiGen. Il était donc important de limiter autant que possible le coût de la migration. D'autre part, l'interface générée par ApiGen est simple, et a fait ses preuves au sein du développement de Tom ainsi que dans ses autres applications.

La différence principale au niveau de l'interface de programmation est l'absence d'utilisation d'une fabrique explicite pour la construction des objets, pour utiliser des fonctions de création statiques. Ceci permet de simplifier l'interface de programmation, tout en autorisant un contrôle plus fin des règles de visibilité lors de la génération des fonctions de normalisation.

Exemple 21. On considère ici un module faisant intervenir les sortes *builtin* `int` et `String`, qui représentent les entiers et les chaînes de caractères.

```
module Expressions
imports String int

abstract syntax
Bool = True()
     | False()
     | Eq(lhs:Expr, rhs:Expr)
Expr = Id(stringValue:String)
     | Nat(intValue:int)
     | Add(lhs:Expr, rhs:Expr)
     | Mul(lhs:Expr, rhs:Expr)
     | Comp(Expr*)
```

Les sortes *builtin* `int` et `String` sont directement représentées par leurs équivalents en Java. Ensuite, pour le module `Expressions`, pour chaque sorte de ce module, le patron de conception « Composite » est utilisé. Chaque sorte est représentée par une classe abstraite, et chaque constructeur de cette sorte hérite de cette classe abstraite. Cette classe abstraite définit des prédicats `boolean isOperator()` pour chaque opérateur de la sorte, qui renvoient `false` par défaut. Elle définit aussi des accesseurs et modifieurs pour chaque champ des opérateurs de la sorte (lesquels produisent une exception par défaut). Ceci permet lors de l'utilisation des objets générés pour cette sorte de pouvoir manipuler des références vers des objets du type Java de la sorte, sans avoir à les convertir vers le type Java de l'opérateur lui-même. Les implémentations de chaque opérateur sont alors placées dans un `package` du nom de la sorte à laquelle ils appartiennent, afin de forcer leur utilisation directe à être explicite (car il faut explicitement importer les classes du package). La figure 4.3 présente des parties de l'implémentation de la structure de

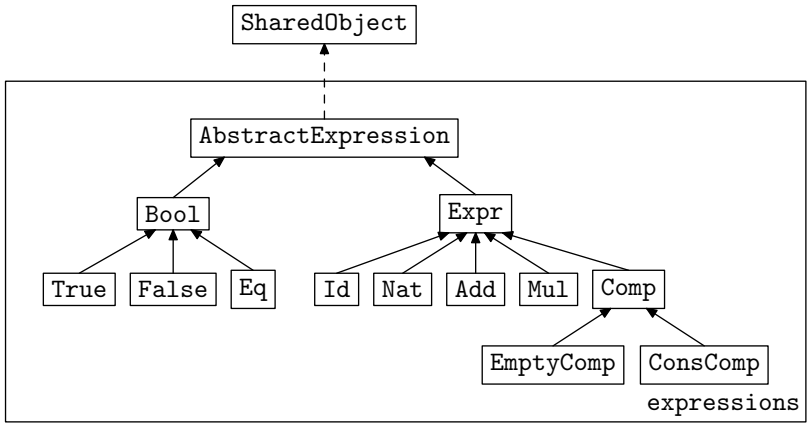


FIG. 4.2: La hiérarchie de types pour les sortes générées

données décrite dans l'exemple 21, tandis que la figure 4.2 présente la relation d'héritage entre les différentes classes. On peut remarquer que les classes implantant les opérateurs fournissent une méthode statique `make`, qui est utilisée pour construire un objet de cette classe. On verra dans la section 4.3.2 comment on peut utiliser cette méthode pour garantir le partage maximal, et plus tard les invariants de la structure.

Le statut de la classe `Comp` est un peu particulier, car c'est une classe représentant un opérateur variadique. Dans cet exemple, on pourrait utiliser cet opérateur `Comp` pour représenter la construction « `,` » du langage C permettant de composer les expressions. `Comp` étant variadique, on ne peut pas écrire de fonction `make` ayant le nombre correct d'arguments en Java, et cette classe est déclarée abstraite. On utilise pour représenter et construire les instances de `Comp` deux sous-classes de celui-ci, `ConsComp` et `EmptyComp`, le premier étant un opérateur binaire, le second une constante. Les sous-termes de `ConsComp` sont respectivement de la sorte du domaine et du codomaine de `Comp`. Cela permet de représenter un `Comp` d'arité n par un peigne droit constitué de n `ConsComp` terminé par un `EmptyComp`.

Cette représentation n'est pas sans causer quelques problèmes, à la fois pratiques et théoriques, que nous nous efforcerons de résoudre en section 5.

Partage, immutabilité

Intéressons nous tout d'abord à la manière d'assurer le partage maximal des objets que l'on définit en utilisant Gom.

Nous avons la possibilité de réutiliser la bibliothèque `shared-objects` qui a été développée comme base pour l'implantation en Java de la bibliothèque des ATerm et ensuite APIGEN. Afin de pouvoir utiliser cette bibliothèque, les objets à partager doivent comme décrit en section 4.1.1 implanter l'interface `SharedObject`, ce qu'ils font par l'intermédiaire de la classe `AbstractExpression`.

Pour s'assurer ensuite que tous les objets créés sont bien partagés, il faut garantir que tous les objets que l'utilisateur est amené à manipuler ont été obtenus par l'appel

expressions/Expr.java

```
package expressions;
public abstract class Expr extends AbstractExpression {
  public boolean isId() { return false; }
  public boolean isNat() { return false; }
  public boolean isAdd() { return false; }
  public boolean isMul() { return false; }

  public int getintValue()    { throw new GetException(...); }
  public String getstringValue(){ throw new GetException(...); }
  public Expr getlhs()        { throw new GetException(...); }
  public Expr getrhs()        { throw new GetException(...); }

  public Expr setlhs(Expr lhs) { throw new SetException(...); }
  public Expr setrhs(Expr rhs) { throw new SetException(...); }
  public int setintValue(int val) {
    throw new SetException(...);
  }
  public String setstringValue(String val) {
    throw new SetException(...);
  }
...
}
```

expressions/expr/Add.java

```
package expressions.expr;
public class Add extends Expr {
  private Expr _lhs;
  private Expr _rhs;

  public boolean isAdd() { return true; }

  public Expr getlhs() { return _lhs; }
  public Expr getrhs() { return _rhs; }
  public Expr setlhs(Expr arg) { return make(arg,_rhs); }
  public Expr setrhs(Expr arg) { return make(_lhs,arg); }

  public static Expr make(Expr lhs, Expr rhs) { ... }
...
}
```

Fig. 4.3: Les classes générées pour une sorte et des opérateurs

```
public class Mul extends Expr {
  private static Mul proto = new Mul();
  private int hashCode;
  private Mul() {}

  private Expr _lhs;
  private Expr _rhs;

  public static Mul make(Expr lhs, Expr rhs) {
    proto.initHashCode(lhs, rhs);
    return (Mul) shared.SingletonSharedObjectFactory.getInstance()
                                                    .build(proto);
  }

  private void initHashCode(Expr lhs, Expr rhs) {
    this._lhs = lhs;
    this._rhs = rhs;
    this.hashCode = hashFunction();
  }
...
```

FIG. 4.4: Exemple d'implémentation pour la méthode `make` de la classe `Mul`

à la méthode `SharedObject build(SharedObject prototype)` de la fabrique d'objets partagés. Il faut pour cela interdire à l'utilisateur d'instancier de nouveaux objets des types représentant les opérateurs par appel à `new`. Ainsi, les classes des opérateurs devront toutes déclarer leur constructeur privé, afin qu'il ne puisse être utilisé que depuis la classe elle-même, et non depuis le code de l'utilisateur. C'est pour cette raison que l'on fournira une méthode `make` statique dont le but est de permettre de récupérer un objet représentant l'opérateur à partir de valeurs pour ses sous-termes. Cette méthode `make` doit alors appeler la méthode `build` de la fabrique.

On utilise pour cela un prototype privé, qui est réutilisé pour chaque création. Ceci permet d'éviter toute allocation mémoire lorsque l'objet que l'on veut récupérer est déjà présent en mémoire. Comme illustré par la figure 4.4, le prototype est modifié pour utiliser les sous-termes arguments de `make`, et sa somme de hachage est mise à jour avant l'appel à la fabrique d'objets partagés. On utilise un *singleton* pour accéder à cette fabrique, ce qui assure un point d'entrée unique pour le partage des termes.

La fonction de hachage utilisée pour hacher les différents opérateurs est générée par le compilateur Gom, en fonction du profil de chaque opérateur. Un traitement spécial est fait pour les champs de type *builtin*, afin de s'assurer de leur partage maximal [2], ainsi qu'assurer une fonction de hachage efficace. À la différence des structures de la

[2]Par exemple, prendre garde d'utiliser la méthode `intern()` pour les champs de type `String`

Taille	ATerm	ApiGen	Gom
50	0,038	0,145	0,037
100	0,098	0,208	0,066
150	0,254	0,377	0,139
200	0,584	0,761	0,294
250	1,25	1,412	0,573
300	2,264	2,501	1,069
350	3,713	4,176	1,757
400	5,666	5,954	2,842
450	8,33	8,845	4,088
500	11,747	12,46	5,853

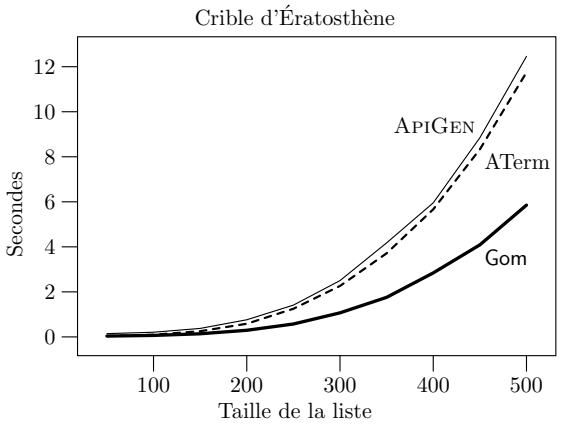


Fig. 4.5: Résultats expérimentaux comparant l'implémentation sans partage, générée par ApiGen et par Gom. Temps en secondes.

bibliothèque ATerm, ou de celles qui sont générées par ApiGen et utilisent une représentation interne ATerm, ces structures générées spécifiquement selon le profil de chaque opérateur suppriment le besoin de manipuler une liste de sous-termes (c'est-à-dire que les opérateur n'ont plus tous une structure en `ATermAppl([])`). Finalement, le compilateur génère pour chaque signature une bibliothèque équivalente a ATerm optimisée pour cette signature.

Mesures de performance

On compare ici l'implantation générée par Gom pour une certaine structure de données avec tout d'abord une implantation utilisant la bibliothèque des ATerm pour représenter les listes chaînées, puis avec l'implantation générée par l'outil ApiGen. Ces implantations fournissant des interfaces de programmation très proches, les mêmes programmes (à de petites variations près) de test sont utilisés.

Notre premier test consiste en l'implémentation du crible d'Ératosthène travaillant sur des listes d'entiers. L'algorithme du crible est réutilisé avec trois implantations différentes des listes d'entiers. La première est une implantation de la liste d'entiers par une liste de ATerm non typée. Les deux autres sont les implantations générées par ApiGen et Gom pour un opérateur `conc` variatique dont le domaine est l'ensemble des entiers. On peut voir sur la figure 4.5 que l'implantation de la liste par Gom est la plus rapide sur cet exemple. Il n'y a qu'un nombre réduit de possibilités de partage, et les trois implantations ont des quantités similaires d'allocation à effectuer. La structure générée par Gom se montre plus efficace que celle qui est générée par ApiGen, ainsi que celle qui utilise la bibliothèque ATerm. Ceci est dû à une meilleure gestion des types *builtin* : les entiers représentés par des objets de la bibliothèque ATerm doivent être encapsulés dans des objets, ce qui est coûteux. Les structures générées par ApiGen utilisent cette bibliothèque pour leur représentation interne, et donc souffrent du même problème.

Émetteurs	Messages	ApiGen	Gom	Gom avec comparaison	Gain (%)
1	1	31	25	24	22
1	2	367	267	258	29
1	3	521	321	292	43
1	4	668	358	317	52
1	5	785	387	334	57
1	6	797	386	336	58
1	7	798	385	336	58
1	8	796	386	336	58
1	9	796	384	336	58
1	10	795	387	337	58
2	1	1 049	670	622	41
2	2	82 592	17 925	13 268	84

Fig. 4.6: Implémentation de la vérification du protocole de Needham-Schroeder utilisant ApiGen ou Gom. Les temps sont donnés en millisecondes.

D'autre part, l'utilisation par ApiGen d'une représentation interne non typée générique introduit des indirections coûteuses, mais celles-ci sont minimes comparées au coût de l'encapsulation des entiers.

Une exemple plus réaliste est la recherche de vulnérabilités dans le protocole d'authentification à clé publique de Needham-Schroeder. Ce protocole est bien connu dans le domaine de la sécurité des protocoles, et fait partie des exemples classiques lorsque l'on veut expliquer l'insécurité d'un protocole de communication cryptographique [NS78]. Ce protocole est décrit par seulement quelques règles, mais n'a été prouvé vulnérable qu'en 1995 par Gavin Lowe [Low95]. Afin de montrer en quoi Tom est un langage adapté à la description de stratégies de vérification pour les protocoles, nous avons étudié comment encoder la recherche de vulnérabilités dans ce protocole en utilisant Tom et ApiGen. ApiGen fournissait alors une implantation efficace pour la structure de données représentant l'état du réseau et des agents dans ce réseau, tandis que Tom permettait d'exprimer les différentes évolutions de ces états, ainsi que la recherche de vulnérabilité [CMR04]. Cette implémentation à été montrée tout à fait compétitive sur le plan de l'efficacité par rapport à d'autres approches basées sur des langages de réécriture ou *model checker* comme Murphi. La figure 4.6 récapitule les temps d'exécution de cette recherche de vulnérabilité sur une version corrigée du protocole (et donc tout l'espace de recherche est exploré), en utilisant la structure de données générée par ApiGen ou générée par Gom. Le calcul effectué et l'ensemble de termes explorés par chaque implémentation sont exactement les mêmes. Ces temps sont calculés pour une taille de réseau (le nombre de messages circulant simultanément) variant, ainsi que pour un nombre d'émetteurs et de récepteurs variable (on a ici toujours autant d'émetteurs que de récepteurs), avec un seul

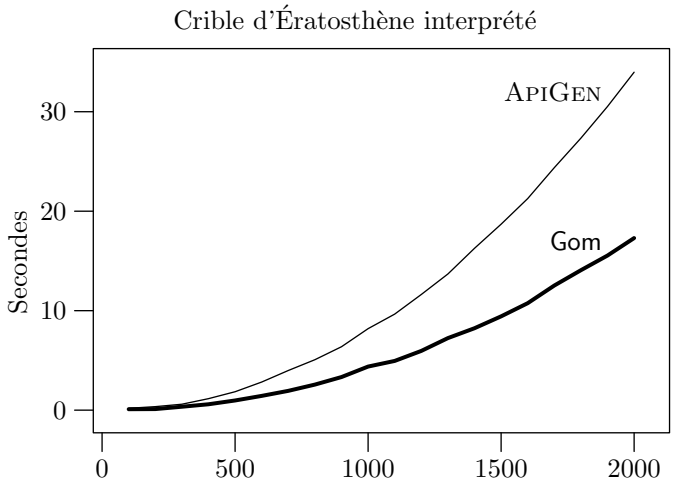
intrus sur le réseau.

L'implémentation originale utilisait des fonctions d'entrée/sortie afin d'implanter un ordre total sur les termes, en comparant les représentations sous forme de chaîne de caractères de ces objets. Cette méthode était utilisée pour pallier au manque de fonction de comparaison dans le code généré par ApiGen. La première version Gom fait de même, et utilise la conversion vers les chaînes de caractères pour la comparaison. La seconde version utilise la fonction de comparaison générée par Gom. Celle-ci est bien plus efficace, car la comparaison se fait alors sans allocation mémoire, et utilise la fonction de hachage pour éviter les appels récursifs autant que possible.

Le gain en efficacité de la structure de donnée générée par le compilateur Gom vis-à-vis de l'implémentation de la même structure générée par ApiGen est légèrement supérieur à deux pour cet exemple en général. Les dernières lignes du tableau de la figure 4.6 montrent un gain très supérieur à deux lorsque l'espace de recherche est très grand (car dans ce cas, il y a 2 émetteurs et 2 récepteurs sur le réseau). Ceci est du au fait que les structures générées par Gom utilisent moins d'espace mémoire, et dont placent moins de poids sur le ramasse miette de la machine virtuelle pour les exemples utilisant beaucoup de mémoire.

```
n ←2
tant que n ≠ MAX:
  i ←2
  p ←1
  tant que i ≠ n:
    si n mod i ≠ 0:
      p ←0
    i ←i + 1
  si p ≠ 0:
    affiche n
  n ←n + 1
```

(a) Programme Pico effectuant les tests



(b) Temps d'exécution de l'interprète

Fig. 4.7: Interpréteur Pico exécutant un programme de tests de primalité

On peut aussi comparer les efficacités des structures générées par Gom et ApiGen sur l'interprétation d'une langage impératif simple, que l'on appellera Pico. Ce langage permet d'exprimer des expressions booléennes ainsi que des calculs sur les nombres entiers, multiplication, addition et division euclidienne, et contient des instructions de test, une boucle « tant que » et l'assignation de variables. On utilise des structures Gom et ApiGen respectivement pour implanter les AST de ce langage, et définit un interpréteur pour les programmes définis dans ce langage.

On évalue alors le temps d'exécution d'un programme calculant effectuant un test de primalité simple toutes les valeurs inférieures à un maximum, à l'aide de deux boucles « tant que ».

Une dernière application montrant l'efficacité du code produit par Gom est celui des évaluations des fonctions `evalsym17`, `evalexp17` et `evaltree17`, qui sont des variations sur le thème de la normalisation d'expressions $2^n \bmod 17$ pour des valeurs de n entre 15 et 20, avec une représentation des entiers utilisant les entiers de Peano. Ces exemples ont été tout d'abord présentés pour évaluer la gestion de la mémoire du compilateur ASF+SDF [BKO99]. Ils sont caratérisés par l'évaluation d'un grand nombre de transformations, sur un grand nombre d'arbres. Ces programmes ont des comportement différents :

- `evalsym17` est surtout utilisateur de temps de calcul, et n'effectue pas beaucoup d'allocations ;
- `evalexp17` utilise beaucoup d'allocation mémoire ;
- `evaltree17` utilise aussi beaucoup d'allocation mémoire, mais présente moins d'opportunités de partage (moins de redondance).

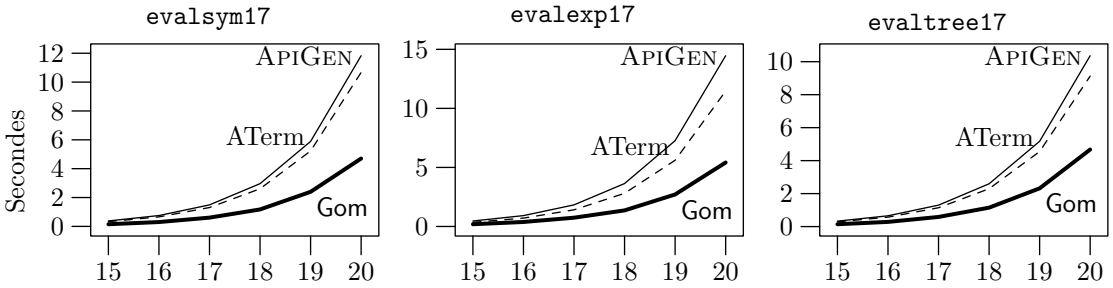


Fig. 4.8: Temps d'exécution des programmes `evalsym17`, `evalexp17` et `evaltree17` utilisant différentes structures de données

Les temps d'exécution présentés en figure 4.8 montrent dans chacun des cas un gain important.

Sur ces exemples, on constate un facteur d'accélération d'environ deux pour les structures générées par Gom par rapport aux structures similaires générées par ApiGen ou pour des objets de la bibliothèque ATerm.

Analyse de complexité Les tests d'efficacité empiriques, sur des applications d'importance variable, allant du crible d'Ératosthène à la vérification par *model checking* du protocole de Needham-Schroeder montrent un facteur d'accélération au moins égal à 2.

Ceci est dû en grande partie à la spécialisation des fonctions d'allocation dans les structures générées par Gom par rapport à celles qui sont générées par ApiGen. En particulier, celui-ci, afin de conserver une compatibilité avec la librairie ATerm, doit dans la représentation d'un opérateur n-aire stocker un tableau à n entrées, lorsque les structures générées par Gom possèdent seulement n champs. À chaque construction, il est alors nécessaire d'allouer un nouveau tableau, avant de calculer la somme de hachage du terme pour vérifier s'il n'est pas déjà présent en mémoire. Cette allocation n'est pas

nécessaire dans le cas des structures générées par Gom. Ainsi, le coût de construction en termes d'allocation mémoire est de n copies d'adresses pour les structures générées par Gom, pour n copies d'adresses et l'allocation d'un tableau de n adresses pour les structures générées par ApiGen.

Une autre source de gain en efficacité est le fait que les symboles de fonction ne sont plus représentés par un objet spécifique `AFun`, mais directement par la classe de l'opérateur. Ainsi, lors du calcul de la valeur de hachage d'un objet, il est nécessaire de calculer celle de l'objet représentant le symbole de fonction dans le cas des structures générées par ApiGen, alors que ce calcul est effectué lors de la compilation pour les structures générées par Gom.

Les opérations de construction et de hachage étant les opérations les plus coûteuses et les plus communes lors de la manipulation d'arbres, car chaque transformation implique une nouvelles construction, les simplifications et améliorations apportées à ces procédures expliquent le gain en efficacité.

4.3.3. Filtrage pour Tom

L'outil Gom est particulièrement utile lorsqu'il est utilisé conjointement avec Tom. Gom est alors utilisé pour fournir une implémentation pour un type de données abstrait qui sera utilisé dans un programme Tom.

À partir d'une description de la structure de données, Gom génère non seulement l'implantation en Java de cette structure, mais aussi un *mapping* qui est un ancrage formel utilisé par Tom pour manipuler les données.

L'utilisateur peut alors écrire des constructions de filtrage sur les classes de l'arbre de syntaxe abstrait ainsi produit, et Tom compile le tout vers Java.

L'ancrage généré par Gom pour une structure de donnés permet d'utiliser les sortes et opérateurs algébriques directement dans du code Tom. En particulier, les noms des opérateurs et des champs sont préservés, ce qui rend l'intégration transparente. Afin de rendre cette intégration plus simple encore, le langage Tom fournit une construction **%gom** qui permet de spécifier directement au niveau du source Tom la signature algébrique à utiliser. Cette signature est alors communiquée au compilateur Gom, qui génère l'implantation et l'ancrage pour Tom.

4.4. Représentants canoniques

On a présenté en section 4.2.2 pourquoi il est souhaitable d'intégrer des invariants au sein même de la structure de données décrite par une signature algébrique.

La description et l'intégration des invariants dans la structure de données, plutôt que laisser à l'utilisateur la charge de s'assurer que ces invariants sont bien respectés permet de donner des garanties sur les objets manipulés.

Notre approche pour la description des invariants que la structure de données doit respecter est d'essayer de fournir un maximum de souplesse à l'utilisateur, tout en permettant aussi une expression à « haut niveau » de ces invariants. Cela s'insère dans les lignes directrices du développement de Tom, qui fournit des constructions pour exprimer

du filtrage équationnel et des stratégies (donc plutôt de « haut niveau »), tout en laissant la possibilité à l'utilisateur de revenir aux niveaux les plus bas, et d'utiliser toutes les ressources du langage hôte, qu'il soit Java, C ou Caml.

On permettra alors à l'utilisateur de Gom la description des invariants de la structure de données en Java, mais aussi en utilisant les constructions de filtrage de Tom.

4.4.1. Hooks

Cette description se fait par le biais de *hooks*, qui permettent de modifier le comportement des opérations sur la structure de données pour chaque opérateur.

Pour illustrer l'utilisation et l'écriture des *hooks*, nous nous appuierons sur l'exemple des lois de De Morgan considérées 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'implémenter 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 de *hook* de Gom permet de définir des actions arbitraires à exécuter avant (ou en lieu et place) la fonction originale de création d'un opérateur. Ces actions peuvent être décrites par n'importe quelle construction Java ou Tom, et autorisent ce code Tom à utiliser des constructions pour spécifier la fonction de normalisation.

Pour permettre la définition de *hooks*, on ajoute à la syntaxe de Gom les définitions des productions $\langle HookDefinition \rangle$ et $\langle HookOperation \rangle$:

$$\begin{array}{lcl} \langle HookDefinition \rangle & ::= & \langle OperatorName \rangle \textbf{:} \langle HookOperation \rangle \ \textbf{\{} \ \langle TomCode \rangle \ \textbf{\}} \\ \langle HookOperation \rangle & ::= & (\textbf{make} \mid \textbf{make\_insert}) \textbf{(} \ [\langle Identifier \rangle] \ (\textbf{,} \ \langle Identifier \rangle)^* \ \textbf{)} \end{array}$$

Un *hook* est attaché à une définition d'opérateur, et permet d'étendre ou redéfinir la fonction de construction de cet opérateur. Suivant le type du *hook*, qui est soit **make**, soit **make_insert**, celui-ci s'appliquera à un opérateur algébrique, ou variadique. En fait, la distinction entre **make** et **make_insert** n'est pas réellement nécessaire, car on ne peut pas utiliser **make** pour un *hook* d'opérateur variadique, et inversement, **make_insert** n'a pas de sens lorsqu'on s'intéresse à un opérateur algébrique. Cependant, cette distinction permet de rendre explicite la différence entre les *hooks* d'opérateurs algébrique, et ceux qui traitent des opérateur variadiques : tandis que le *hook* **make** prendra des arguments compatibles avec la spécification de l'opérateur algébrique associé, un *hook* **make_insert** prendra deux arguments, le premier ayant le type du domaine de l'opérateur variadique associé et le second celui du co-domaine. La définition du **make_insert** ne modifie alors pas directement l'opération de création d'un opérateur variadique, mais

plutôt l'opération d'insertion d'un nouvel élément dans la liste des arguments d'un opérateur variadique.

Exemple 22. De tels *hooks* peuvent être utilisés pour définir le système 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)
not:make(arg) {
  %match(Bool arg) {
    not(x)  -> { return `x; }
    and(l,r) -> { return `or(not(l),not(r)); }
    or(l,r) -> { return `and(not(l),not(r)); }
  }
}
```

On voit ici qu'il est possible (et d'ailleurs très utile) d'utiliser Tom dans la définition du *hook* pour filtrer sur la structure de données même qui est en train d'être décrite. Cela laisse la possibilité à l'utilisateur de définir les *hooks* comme des règles de réécriture pour obtenir une fonction de normalisation. Si l'exécution de cette définition se termine sans retourner de valeur, alors la fonction de création originale de l'opérateur est utilisée, c'est le cas dans notre exemple lorsque l'argument `arg` est l'une des constantes `True()` ou `False()`. Il est aussi possible d'appeler la fonction `makeReal`, qui correspond à la fonction de création originale.

Lorsqu'il utilise le mécanisme des *hooks*, l'utilisateur doit s'assurer que le système de normalisation définit par les *hooks* est confluent et terminant, car ces propriétés ne seront pas garanties par le compilateur Gom. D'autre part, la combinaison de *hooks* 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 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. Ce langage permettrait de laisser la tâche de la complétion des règles de normalisation ou leur combinaison à son compilateur, ainsi que la vérification de leur terminaison.

On peut considérer Gom comme un composant réutilisable, conçu pour être un outil permettant d'implanter un autre langage (de la même manière que la bibliothèque des ATerm et ApiGen ont été utilisés comme base pour ASF+SDF [JO04]), ou comme composant dans une architecture plus complexe. D'autre part, l'introduction des *hooks*

permettant de modifier le comportement des constructeurs est une idée qui existe aussi dans les langages fonctionnels, comme les types privée de Caml introduits par Pierre Weis [LDG+04], qui permettent de forcer l'utilisateur à définir et utiliser des fonctions de constructions explicites pour les objets d'un type algébrique tout en préservant la possibilité d'utiliser le filtrage. Le langage MOCA [BWH06] est une extension au langage des types inductifs de Caml permettant d'obtenir des fonctions de normalisation pour un type inductif annoté avec des théories qui sont associées aux constructeurs.

4.5. Interactions avec Tom

Nous avons vu en section 4.3.3 que Tom peut être utilisé dans une application utilisant Gom pour filtrer sur les structures d'arbres que Gom permet de représenter. La définition des *hooks* peut aussi faire intervenir Tom, permettant de faciliter leur définition, par l'utilisation d'un langage de plus haut niveau.

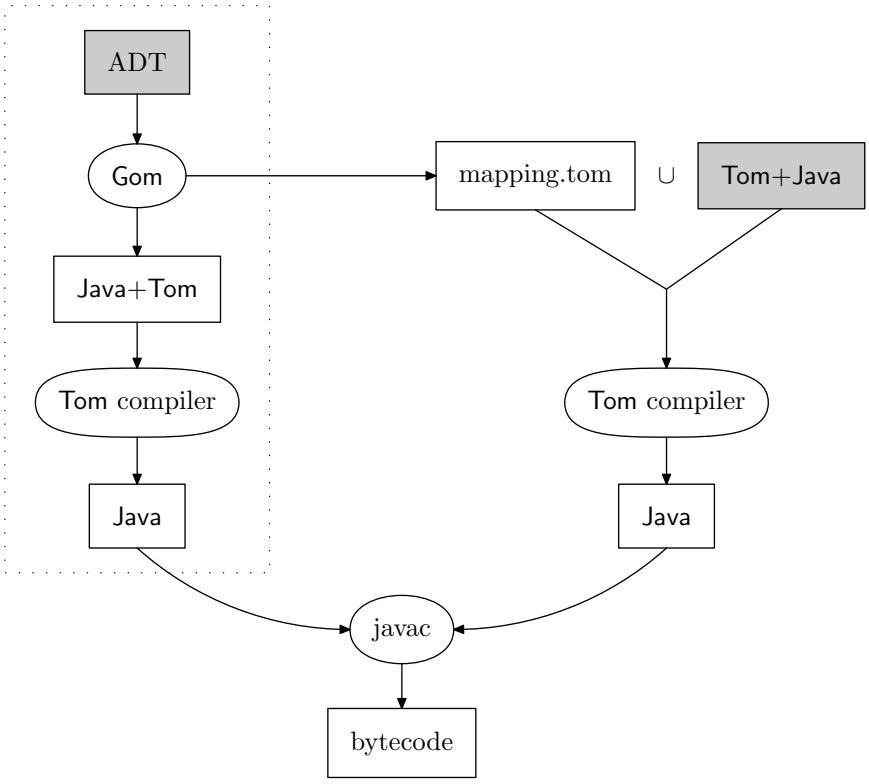


Fig. 4.9: Interactions entre Tom et Gom

Ainsi, le compilateur Gom, à partir d'une description de structure de données et de ses invariants (décrits par des *hooks* utilisant le langage Tom), va produire un ensemble de fichiers Java. Ce compilateur est conçu pour générer l'implémentation de la structure de

données en Java et Tom, et appelle le compilateur Tom pour traduire les constructions Tom des hooks en Java. Comme il est explicité dans la figure 4.9, le compilateur produit aussi le *mapping* pour la structure, qui est utilisé dans l'application Tom « cliente ».

4.5.1. Un exemple plus élaboré : le calcul des structures

Afin d'illustrer l'expressivité de Gom et de montrer combien le fait de fournir des fonctions de normalisation peut rendre le développement d'applications complexes plus facile, nous allons développer un exemple réaliste d'utilisation. Le calcul des structures [Gug02] est un formalisme qui généralise le calcul des séquents en introduisant l'*inférence profonde* : contrairement au calcul des séquents, le calcul des structures n'utilise pas la notion de connecteur principal, mais à l'instar de la réécriture de termes autorise l'application des règles d'inférence à toutes les profondeurs dans une formule.

Nous avons implanté un prouveur efficace et sûr pour le système BV du calcul des structures. L'implémentation originale s'est faite à la suite d'une collaboration avec Ozan Kahramanoğulları, visant à explorer les possibilités d'une implantation utilisant Tom du système BV. Cette implémentation avait alors été écrite en utilisant Tom et ApiGen, mais utilisait l'approche décrite ci-dessous, en implémentant des invariants que la structure de données devait préserver, afin de restreindre l'impact du non déterminisme du calcul sur la recherche de preuve.

Ainsi, certaines règles ont pu être promues au titre d'invariants de la structure de donnés, permettant une implémentation plus simple et efficace des règles du calcul. Cette promotion s'est faite en modifiant les règles originales du calcul pour séparer les aspects calculatoires et de représentation des données des règles de déduction. Ces invariants et règles ont alors été prouvés corrects vis-à-vis du calcul original, conduisant à l'écriture d'un prouveur automatique efficace et correct [KMR05a].

C'est la difficulté technique qu'a représenté l'implantation et le maintien des invariants de la structure de données dans le cadre de cette implémentation qui a conduit à la conception de Gom, ainsi que le temps passé à rechercher et corriger les différents problèmes conduisant au non respect des invariants. Ceci était rendu particulièrement difficile à cause de la nécessité d'implanter l'inférence profonde, chaque application d'une règle de déduction pouvant briser les invariants à toute position dans l'arbre de preuve.

4.5.2. L'approche

Pour implémenter un prouveur pour une logique particulière, il est nécessaire de raffiner la stratégie d'application des règles du calcul. Ceci est particulièrement vrai dans le cas du calcul des structures, car la présence de l'inférence profonde et d'opérateurs associatifs-commutatifs introduisent un fort non-déterminisme. Ceci est encore aggravé par la non-confluence du calcul.

On décrit ici brièvement le système BV du calcul des structures, pour montrer ensuite comment Tom et Gom peuvent aider à fournir une implémentation concise, robuste et efficace de la recherche de preuve dans ce système.

Associativité

$$\langle \boldsymbol{R}; \langle \boldsymbol{T} \rangle; \boldsymbol{U} \rangle \approx \langle \boldsymbol{R}; \boldsymbol{T}; \boldsymbol{U} \rangle$$
$$[\boldsymbol{R}, [\boldsymbol{T}]] \approx [\boldsymbol{R}, \boldsymbol{T}]$$
$$(\boldsymbol{R}, (\boldsymbol{T})) \approx (\boldsymbol{R}, \boldsymbol{T})$$

Fermeture par contexte

si $R = T$ alors $S\{R\} = S\{T\}$ et $\overline{R} = \overline{T}$

Commutativité

$$[\boldsymbol{R}, \boldsymbol{T}] \approx [\boldsymbol{T}, \boldsymbol{R}]$$
$$(\boldsymbol{R}, \boldsymbol{T}) \approx (\boldsymbol{T}, \boldsymbol{R})$$

Unités

$$\langle \circ; \boldsymbol{R} \rangle \approx \langle \boldsymbol{R}; \circ \rangle \approx \langle \boldsymbol{R} \rangle$$
$$[\circ, \boldsymbol{R}] \approx [\boldsymbol{R}]$$
$$(\circ, \boldsymbol{R}) \approx (\boldsymbol{R})$$

Négation

$$\overline{\circ} \approx \circ$$
$$\overline{\langle R; T \rangle} \approx \langle \overline{R}; \overline{T} \rangle$$
$$\overline{[R, T]} \approx (\overline{R}, \overline{T})$$
$$\overline{(R, T)} \approx [\overline{R}, \overline{T}]$$
$$\overline{\overline{R}} \approx R$$

Singleton

$$\langle R \rangle \approx [R] \approx (R) \approx R$$

FIG. 4.10: Relation d'équivalence pour le système BV.

Les atomes dans le système BV sont notés par $a, b, c, \ldots$ Les structures sont quant à elles identifiées par $R, S, T, \ldots$ et générées par :

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

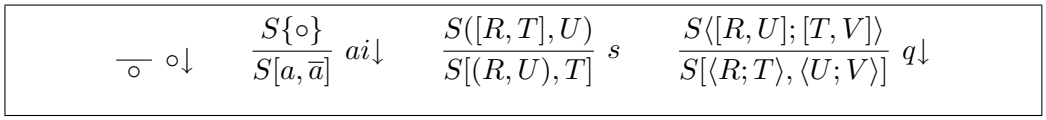
où $\circ$, l'*unité*, n'est pas un atome. On appelle
- $\langle S; \ldots; S \rangle$ une *structure seq*,
- $[S, \ldots, S]$ une *structure par*, et
- $(S, \ldots, S)$ une *structure copar*.

$\overline{S}$ est la négation de la structure S.

Une structure R est appelée *vraie structure par* si $R = [R_1, R_2]$ avec $R_1 \neq \circ$ et $R_2 \neq \circ$. On notera $\boldsymbol{R}$, $\boldsymbol{T}$ et $\boldsymbol{U}$ les séquences finies et non vides de structures. Un *contexte* de structure, noté à la manière de $S\{\ \}$, est une structure comportant un trou. On utilise cette notation pour exprimer les règles de déduction du système BV, et on omettra les accolades lorsqu'il n'y a pas d'ambiguïté.

Les règles du système BV sont simples lorsque l'on considère une relation d'équivalence entre les termes de BV. Les structures *seq*, *par* et *copar* sont associatives, *par* et *copar* étant aussi commutatives. De plus, $\circ$ est un élément neutre pour les structures *seq*, *par* et *copar* et une structure *seq*, *par* ou *copar* constituée d'une seule sous-structure est équivalente à cette sous-structure. Cette relation d'équivalence est détaillée dans la figure 4.10. Alors, les règles de déduction du système BV peuvent être exprimées par les règles de la figure 4.11.

La présence des contextes dans la définition des règles du calcul fait que les règles de réécriture correspondantes peuvent être appliquées non seulement en tête d'une structure, mais aussi sur chaque sous-terme d'une structure, implémentant alors l'inférence profonde. Cette inférence profonde peut alors — combinée avec l'associativité, la commutativité et $\circ$ comme élément neutre pour les structures *seq*, *par* et *copar* — produire une grande quantité de non-déterminisme dans le calcul.



$$\frac{}{\circ}\ \circ\downarrow \qquad \frac{S\{\circ\}}{S[a,\overline{a}]}\ ai\downarrow \qquad \frac{S([R,T],U)}{S[(R,U),T]}\ s \qquad \frac{S\langle[R,U];[T,V]\rangle}{S[\langle R;T\rangle,\langle U;V\rangle]}\ q\downarrow$$

FIG. 4.11: Le système BV

Une implémentation d'un prouveur automatique dans le système BV du calcul des structures suivant strictement cette description devra gérer ce non-déterminisme, et manipuler un très grand espace de recherche, conduisant à un système inefficace [Kah04].

L'approche utilisée par Gom et Tom va être d'identifier les représentants canoniques (ou représentants préférés) pour chaque classe d'équivalence dans les structures, puis d'implanter une procédure de normalisation pour les structures conduisant à la sélection des représentants canoniques en utilisant les *hooks* de Gom. Ce processus requiert de définir tout d'abord la structure de données, puis de définir la normalisation. Cette normalisation nous assurera que toutes les unités $\circ$ dans les structures *seq*, *par* et *copar* sont supprimés, puisque $\circ$ est un élément neutre pour ces structures. On s'assurera aussi que les structures qui sont manipulées sont *aplaties*, ce qui correspond à sélectionner un représentant canonique pour l'associativité de *seq*, *par* et *copar*. De plus, les sous-termes des structures *par* et *copar* seront ordonnés en utilisant un ordre total sur les structures, pour prendre en compte la commutativité de ces structures.

Lorsqu'il faudra implémenter les règles du calcul, il sera nécessaire de prendre en compte le fait que le prouveur ne manipule que des représentants canoniques. Cela conduit à des règles de calcul plus simples, et autorise à effectuer certaines nouvelles optimisations sur les règles.

4.5.3. Structure de données

On doit tout d'abord donner une description syntaxique de la structure de données que le prouveur dans le système BV va utiliser, pour fournir une représentation en termes d'objets des structures *seq*, *par* et *copar* ($\langle R;T\rangle$, $[R,T]$ et (R,T)). Dans notre implémentation, on considérera ces constructeurs comme des opérateurs unaires, prenant chacun comme argument une liste de structures. En utilisant Gom, la structure de données peut alors être décrite par la signature de la figure 4.12. Pour représenter les structures du système BV, on définit tout d'abord quelques atomes constants. Parmi eux, la constante `o()` va être utilisée pour représenter l'unité $\circ$. L'opérateur `neg` construit la négation de son argument, tandis que la définition `Struc = par(parl:StrucPar)` définit un opérateur unaire `par` de sorte `Struc` qui prend comme unique argument un élément de la sorte `StrucPar`. Les opérateurs `cop` et `seq`, correspondant aux structures *copar* et *seq* respectivement sont définis de la même manière, avec des arguments de sortes `StrucCop` et `StrucPar` respectivement.

Les sortes `StrucSeq`, `StrucPar` et `StrucCop` sont alors définies comme ne comportant chacune qu'un seul opérateur variadique, respectivement `concSeq`, `concPar` et `concCop`, prenant chacun un nombre arbitraire d'arguments de sorte `Struc`. Ainsi, en combinant

```
module Structures
abstract syntax
Struc = o()
      | a()
      | b()
      | c()
      | d()
      | neg(a:Struc)
      | cop(copl:StrucCop)
      | par(parl:StrucPar)
      | seq(seql:StrucSeq)
StrucPar = concPar( Struc* )
StrucCop = concCop( Struc* )
StrucSeq = concSeq( Struc* )
```

FIG. 4.12: Signature Gom permettant de représenter les structures du système BV

les opérateurs `par` et `concPar`, il devient possible de représenter la structure $[a, [b, c]]$ par `par(concPar(a(),b(),c()))`. Les structures (R, T) et $\langle R; T\rangle$ sont représentées de manière similaire, en utilisant `cop`, `seq`, `concCop` et `concSeq`.

Note A. Il est possible d'utiliser `par(concPar(a(),b(),c()))` pour représenter la structure $[a, [b, c]]$, utilisant ainsi une représentation aplatie. Cependant, avec la description de la figure 4.12, on peut aussi utiliser des structures `par` emboîtées comme par exemple `par(concPar(a(),par(concPar(b(),c()))))`, faire intervenir des éléments neutres ou des structures à un seul argument.

4.5.4. Les invariants

En utilisant la définition de structure de données de la figure 4.12, on manipule des objets, comme `par(concPar())`, qui ne correspondent pas nécessairement aux structures que l'on veut décrire. Le problème posé par des objets comme `par(concPar())` s'ajoute à celui qui est présenté dans la note A : cet objet ne correspond directement à aucun objet dans le langage des structures du système BV. On peut toutefois le considérer égal au neutre pour l'opérateur [ ], $\circ$.

La possibilité de représenter de différentes manières des structures équivalentes pour le système BV pousse à choisir un de ces représentants comme représentant canonique.

Exemple 23. Les structures `par(concPar(a()))` et `cop(concCop(a()))` sont toutes deux équivalentes à la structure `a()`, car $\langle R\rangle \approx [R] \approx (R) \approx R$.

On définit alors le représentant canonique (ou préféré) en s'assurant que :

– [], ⟨⟩ et () sont réduits lorsqu'ils contiennent une seule sous-structure :

$$\begin{aligned}\texttt{par(concPar(x))} &\rightarrow \texttt{x}\\ \texttt{cop(concCop(x))} &\rightarrow \texttt{x}\\ \texttt{seq(concSeq(x))} &\rightarrow \texttt{x}\end{aligned}$$

– Les structures imbriquées sont aplaties, en utilisant les règles :

$$\begin{aligned}&\texttt{par}(\texttt{concPar}(\texttt{a}_1,\ldots,\texttt{a}_\texttt{i},\texttt{par}(\texttt{concPar}(\texttt{x}_1,\ldots,\texttt{x}_\texttt{n})),\texttt{b}_1,\ldots,\texttt{b}_\texttt{j}))\\ &\quad\rightarrow \texttt{par}(\texttt{concPar}(\texttt{a}_1,\ldots,\texttt{a}_\texttt{i},\texttt{x}_1,\ldots,\texttt{x}_\texttt{n},\texttt{b}_1,\ldots,\texttt{b}_\texttt{j}))\\ &\texttt{cop}(\texttt{concCop}(\texttt{a}_1,\ldots,\texttt{a}_\texttt{i},\texttt{cop}(\texttt{concCop}(\texttt{x}_1,\ldots,\texttt{x}_\texttt{n})),\texttt{b}_1,\ldots,\texttt{b}_\texttt{j}))\\ &\quad\rightarrow \texttt{cop}(\texttt{concCop}(\texttt{a}_1,\ldots,\texttt{a}_\texttt{i},\texttt{x}_1,\ldots,\texttt{x}_\texttt{n},\texttt{b}_1,\ldots,\texttt{b}_\texttt{j}))\\ &\texttt{seq}(\texttt{concSeq}(\texttt{a}_1,\ldots,\texttt{a}_\texttt{i},\texttt{seq}(\texttt{concSeq}(\texttt{x}_1,\ldots,\texttt{x}_\texttt{n})),\texttt{b}_1,\ldots,\texttt{b}_\texttt{j}))\\ &\quad\rightarrow \texttt{seq}(\texttt{concSeq}(\texttt{a}_1,\ldots,\texttt{a}_\texttt{i},\texttt{x}_1,\ldots,\texttt{x}_\texttt{n},\texttt{b}_1,\ldots,\texttt{b}_\texttt{j}))\end{aligned}$$

– Les sous-termes de *par* et *copar* sont triés selon un ordre total $<$:

$$\begin{aligned}&\texttt{concPar}(\ldots,\texttt{x}_\texttt{i},\ldots,\texttt{x}_\texttt{j},\ldots) \rightarrow \texttt{concPar}(\ldots,\texttt{x}_\texttt{j},\ldots,\texttt{x}_\texttt{i},\ldots) \text{ if } \texttt{x}_\texttt{j} < \texttt{x}_\texttt{i}\\ &\texttt{concCop}(\ldots,\texttt{x}_\texttt{i},\ldots,\texttt{x}_\texttt{j},\ldots) \rightarrow \texttt{concCop}(\ldots,\texttt{x}_\texttt{j},\ldots,\texttt{x}_\texttt{i},\ldots) \text{ if } \texttt{x}_\texttt{j} < \texttt{x}_\texttt{i}\end{aligned}$$

Cette notion de forme canonique nous permet de vérifier efficacement si deux objets représentent la même structure modulo l'associativité, la commutativité, les éléments neutres et les règles de réduction des différents connecteurs. Cette vérification est de fait très efficace, car elle correspond à une comparaison de pointeurs, deux objets d'une même classe d'équivalence étant représentés par le même représentant canonique, donc le même objet en mémoire grâce au partage maximal.

Le premier invariant à maintenir est la réduction des singletons pour les structures *seq*, *par* et *copar*. Lorsque l'utilisateur essaye de construire un `seq`, `par` ou `cop` avec une liste vide de structures, alors la fonction de création doit retourner l'unité `o()`. Sinon, si la liste contient un seul élément, elle doit retourner l'élément lui même. Dans les autres cas, cette fonction de création construira la structure demandée. Comme tous les objets manipulés par l'utilisateur sont en forme normale (sont des représentants canoniques), il n'est pas nécessaire dans le cas de cet invariant de traiter le cas d'une liste de structures en argument contenant l'unité ou un seul élément, car cela est rendu impossible par l'invariant sur la liste. Ce comportement est implanté comme un *hook* pour les opérateurs `seq`, `par` et `cop`.

Le premier *hook* de la figure 4.13 implémente la réduction des singletons pour l'opérateur `par` en utilisant le filtrage de Tom. Le filtrage sur les opérateurs de liste de Tom est utilisé pour examiner la liste passée en argument du constructeur de `par`. Si la liste n'est pas un singleton, alors la fonction originale de construction est implicitement appelée. Cette fonction de création originale n'est accessible que depuis la définition du hook correspondant à l'opérateur. Ainsi, l'utilisateur de la structure de données ne peut l'utiliser, mais utilisera la fonction comportant le hook, qui met en place les invariants. Les opérateurs `cop` et `seq` sont munis de *hooks* similaires.

Les hooks permettant de normaliser les listes de structures sont plus complexes. Il est tout d'abord nécessaire d'utiliser un ordre total sur les structures. Cet ordre peut

```
...
par:make(l) {
  %match(StrucPar l) {
    concPar() -> { return `o(); }
    concPar(x) -> { return `x; }
  }
}
cop:make(l) {
  %match(StrucCop l) {
    concCop() -> { return `o(); }
    concCop(x) -> { return `x; }
  }
}
seq:make(l) {
  %match(StrucSeq l) {
    concSeq() -> { return `o(); }
    concSeq(x) -> { return `x; }
  }
}
...
```

FIG. 4.13: Les *hooks* pour `par`, `cop` et `seq`.

être implanté par une fonction extérieure[3]. Toutefois, les structures générées par Gom comportent une implantation d'un ordre total très efficace, rendu disponible par l'implantation de l'interface `Comparable` par toutes les classes correspondant aux opérateurs. Il est donc raisonnable dans ce cas d'utiliser cet ordre, car aucune condition particulière n'est requise sur l'ordre utilisé (il doit toutefois être total).

On peut donc définir les hooks pour les opérateurs variadiques `concSeq`, `concPar` et `concCop`. Ceux-ci sont définis en surchargeant la fonction d'insertion en tête dans la liste d'arguments de l'opérateur variadique, en utilisant **make_insert**.

Le *hook* pour `concSeq` est le plus simple, car les structures $\langle\rangle$ sont seulement associatives, avec $\circ$ leur élément neutre. Le hook correspondant doit donc supprimer les unités et aplatir les `seq` imbriqués. L'implantation de ce hook en figure 4.14 utilise une fois encore le filtrage de Tom pour examiner la forme de l'élément à insérer dans la liste d'arguments d'un opérateur `concSeq`. Si celui-ci est une unité, la liste de départ est retournée, et si c'est une structure `seq`, sa liste interne d'arguments est concaténée à la liste d'arguments de l'opérateur. Les hooks pour `concCop` et `concPar`, présentés en figure 4.15 sont similaires, mais examinent aussi le second argument : la liste dans laquelle on insère l'élément, afin d'implanter une insertion triée selon l'ordre interne à la struc-

[3]La seule contrainte est que cette fonction doit être déclarée statique, et être accessible. Elle peut être écrite en utilisant le filtrage de Tom.

```
StrucSeq = concSeq( Struc* )
concSeq:make_insert(e,l) {
  %match(Struc e) {
    o() -> { return l; }
    seq(concSeq(subL*)) -> { return `concSeq(subL*,l*); }
  }
}
```

FIG. 4.14: Le *hook* pour l'opérateur `concSeq`

ture. Ce *hook* utilise `` `concPar(L*,l*) `` pour effectuer un appel récursif à la fonction de création (utilisant la construction ` de Tom, présentée en section 2.2.4). Il utilise aussi l'appel à la fonction `realMake`, qui correspond à l'opération de construction originale.

Le calcul des structures vérifie aussi les lois de De Morgan pour la négation. On peut donc aussi écrire un *hook* pour l'opérateur de négation `neg`, qui applique les lois de De Morgan de manière similaire au *hook* de l'exemple 22. L'application de ce hook garantit que dans les structures manipulées seuls les atomes peuvent être en position négative. Cela simplifie l'application des règles de déduction, en supprimant le besoin de propager les négations.

4.5.5. Les règles du calcul

Une fois la structure de données définie, ainsi que ses invariants, il est possible de définir les règles du calcul elles-même comme des règles de réécriture sur ces objets dans un programme Tom. Ces règles sont appliquées de manière répétée sur la structure d'entrée du prouveur, calculant tous les successeurs jusqu'à l'obtention du *but* à atteindre (en général, l'unité $\circ$).

Ces règles sont exprimées en utilisant le filtrage de Tom sur la structure de données générée par Gom. Elle sont assez simples, car la relation d'équivalence sur les structures est intégrée dans la structure de données par l'intermédiaire des invariants. Cependant, afin d'obtenir un prouveur efficace, il est nécessaire d'imposer certaines restrictions sur l'application des règles, prenant en compte l'intégration de la relation d'équivalence au sein même de la structure de données. Ces restrictions, ainsi que la preuve d'équivalence du système obtenu vis-à-vis du système BV original sont le résultat d'un travail commun avec Ozan Kahramanoğulları [KMR05a].

Définition 27 (Équivalence)**.** *Deux systèmes $\mathcal{S}$ et $\mathcal{S}'$ sont équivalents si pour chaque preuve d'une structure T dans le système $\mathcal{S}$, il existe une preuve de T dans le système $\mathcal{S}'$, et vice versa.*

L'élaboration du nouveau système de règles pour le calcul des structures, qui prend en compte les invariants de la structure de données, et permet de réduire le non-déterminisme du calcul, se fait par raffinements successifs du système original, en prouvant l'équivalence du système raffiné avec le système qu'il raffine.

```
concCop:make_insert(e,l) {
  %match(Struc e) {
    o() -> { return l; }
    cop(concCop(L*)) -> { return `concCop(L*,l*); }
  }
  %match(StrucCop l) {
    concCop(head,tail*) -> {
      if (0 < e.compareTo(`head)) {
        return `realMake(head,concCop(e,tail*));
      }
    }
  }
}
concPar:make_insert(e,l) {
  %match(Struc e) {
    o() -> { return l; }
    par(concPar(L*)) -> { return `concPar(L*,l*); }
  }
  %match(StrucPar l) {
    concPar(head,tail*) -> {
      if(0 < e.compareTo(`tail)) {
        return `realMake(head,concPar(e,tail*));
      }
    }
  }
}
```

FIG. 4.15: Les *hooks* pour les opérateurs `concPar` et `concCop`

On supprime tout d'abord les équations de la relation d'équivalence concernant les unités. Le système obtenu s'applique aux structures en forme normale par rapport au système correspondant à l'orientation des règles concernant la négation et l'unité de la figure 4.10 de la gauche vers la droite. On se donne aussi une restriction sur ces règles, qui correspond à limiter le non-déterminisme introduit par la commutativité des structures *par* et *copar*, en rendant l'application des règles s et $q\downarrow$ « paresseuse » : la structure W dans les règles de la figure 4.16 ne peut pas être une *vraie structure par*. C'est donc un atome, ou une structure *copar* ou *seq*. Ceci définit alors le système BVul [4].

Propriété 6. *Les systèmes* BV *et* BVul *sont équivalents.*

Démonstration. Le système BV est équivalent au système BVul dans lequel on supprime la restriction sur la forme des structures W, car les règles d'inférence du système BVul

[4] BV sans égalités pour l'unité et « *lazy* ».

$$\frac{}{[a,\overline{a}]}\ ax \qquad \frac{S([R,W],T)}{S[(R,T),W]}\ s_1$$

$$\frac{S\{R\}}{S[R,[a,\overline{a}]]}\ ai_1{\downarrow} \quad \frac{S\{R\}}{S(R,[a,\overline{a}])}\ ai_2{\downarrow} \quad \frac{S\{R\}}{S\langle R;[a,\overline{a}]\rangle}\ ai_3{\downarrow} \quad \frac{S\{R\}}{S\langle[a,\overline{a}];R\rangle}\ ai_4{\downarrow}$$

$$\frac{S\langle[R,T];[U,V]\rangle}{S[\langle R;U\rangle,\langle T;V\rangle]}\ q_1{\downarrow} \quad \frac{S\langle R;T\rangle}{S[R,T]}\ q_2{\downarrow} \quad \frac{S\langle[W,T];U\rangle}{S[W,\langle T;U\rangle]}\ q_3{\downarrow} \quad \frac{S\langle T;[W,U]\rangle}{S[W,\langle T;U\rangle]}\ q_4{\downarrow}$$

FIG. 4.16: Le système BVul

autorisent l'unité $\circ$ à être totalement supprimé du langage des structures BV. On peut alors simuler chacune des règles d'inférence d'un système dans l'autre.

En ajoutant la restriction sur W, qui ne peut pas être une *vraie structure par*, on doit montrer que lorsque W est une *vrai structure par*, une dérivation équivalente à celle qui a lieu sans la restriction est possible dans le système BVul. Le cas de la règle $q_4{\downarrow}$ est analogue au cas de la règle $q_3{\downarrow}$, on donne donc les dérivations pour s_1 et $q_3{\downarrow}$:

$$\begin{array}{l} \dfrac{S([R,T,V],U)}{S[([R,U],T),V]}\ s_1 \\ \dfrac{S[[(R,T),U],V]}{S[(R,T),[U,V]]}\ s_1 \\ \qquad = \end{array} \qquad \begin{array}{l} \dfrac{S\langle[R,V,T];U\rangle}{S[R,\langle[V,T];U\rangle]}\ q_3{\downarrow} \\ \dfrac{S[R,[V,\langle T;U\rangle]]}{S[[R,V],\langle T;U\rangle]}\ q_3{\downarrow} \\ \qquad = \end{array}$$

□

On doit alors supprimer les équations de la commutativité des structures *par* et *copar*, afin d'obtenir un système de déduction manipulant uniquement des représentants canoniques. On va aussi introduire une restriction sur l'application des règles permettant de réduire le non-déterminisme lors de la recherche de preuve, en évitant de développer les branches non prouvables lorsqu'il est facile de les identifier. On utilisera pour exprimer ces restrictions la notion l'ensemble d'atomes d'une structure.

Définition 28 (Atomes)**.** *Soit une structure S. La notation $\mathsf{at}\,S$ représente l'ensemble de tous les atomes apparaissant dans S.*

Définition 29 (Restrictions)**.** *Le système de la figure 4.17, appelé système* BVci*. Les équations de la commutativité et de l'unité ne s'appliquent pas aux objets de ce système.*

De plus, on impose les restrictions suivantes sur l'application des règles :

- *dans les règles s_{11a}, s_{12a}, s_{13a}, s_{14a}, s_{11b}, s_{12b}, s_{13b}, s_{14b} on a $\mathsf{at}\,\overline{R} \cap \mathsf{at}\,W \neq \emptyset$;*
- *dans les règles s_{15a}, s_{16a}, s_{15b}, s_{16b} on a $\mathsf{at}\,\overline{(R,U)} \cap \mathsf{at}\,W \neq \emptyset$;*
- *dans les règles $q_{11}{\downarrow}$, $q_{12}{\downarrow}$ on a $\mathsf{at}\,\overline{R} \cap \mathsf{at}\,T \neq \emptyset$ et $\mathsf{at}\,\overline{U} \cap \mathsf{at}\,V \neq \emptyset$;*
- *dans les règles $q_{31}{\downarrow}$, $q_{32}{\downarrow}$, $q_{33}{\downarrow}$, $q_{34}{\downarrow}$ on a $\mathsf{at}\,\overline{W} \cap \mathsf{at}\,T \neq \emptyset$;*
- *dans les règles $q_{41}{\downarrow}$, $q_{42}{\downarrow}$, $q_{43}{\downarrow}$, $q_{44}{\downarrow}$ on a $\mathsf{at}\,\overline{W} \cap \mathsf{at}\,U \neq \emptyset$.*

Cette limitation permet de n'appliquer ces règles d'inférence que si la structure résultante présente des possibilités d'interaction, et donc potentiellement peut permettre d'atteindre le résultat.

Théorème 4. *Les systèmes* BV *et* BVci *sont équivalents.*

Démonstration. Les règles d'inférence du système BVci sont des instances du système BV. La preuve de l'autre direction se fait par analyse de cas sur l'application des règles d'inférence du système BVul avec commutativité [KMR05a]. L'équivalence du système BVci avec contraintes et du système BV découle alors des résultats présentés par Ozan Kahramanoğulları [Kah06]. □

Le système BVci est alors utilisable comme base pour l'implantation d'un prouveur dans le système BV du calcul des structures, en utilisant une structure de données fournissant des représentants canoniques pour les classes d'équivalence des structures. en utilisant la construction **%match** de Tom, l'implantation de ces règles est la traduction directe des règles d'inférence dans le langage défini par la signature algébrique, en omettant le contexte. Les restrictions sur l'application des règles sont implantées en utilisant des fonctions auxiliaires Java, ce qui permet une écriture efficace et concise.

4.5.6. Stratégie d'application et recherche de preuve

L'inférence profonde est décrite en utilisant le langage de stratégies et traversées d'arbres qui sera présenté et détaillé dans le chapitre 6. Tous les successeurs de l'application d'une règle de déduction sont collectés, et le processus est réitéré sur ces successeurs jusqu'à l'obtention du but.

Cependant, il est nécessaire d'utiliser une stratégie fine de recherche de preuve, l'espace de recherche pouvant être très grand. L'utilisation de Tom est alors d'une grande valeur, car il est possible de décrire, en utilisant Tom et Java, une stratégie fine et évolutive, alors que les langages algébriques usuels n'autorisent que les stratégies en profondeur d'abord, ou en largeur d'abord. Dans ELAN par exemple, la recherche est implémentée en utilisant un mécanisme de *backtracking*, qui est une bonne approche pour implanter une recherche en profondeur d'abord. Cependant, si cette stratégie est efficace en espace, une telle stratégie peut conduire à entrer dans des branches infinies, et donc peut ne pas terminer même si une preuve existe. D'un autre coté, la recherche en largeur d'abord, comme dans MAUDE termine lorsqu'une preuve existe, mais la quantité de mémoire nécessaire peut être conséquente.

Une implantation de la recherche de preuve dans le système BV pour MAUDE, utilisant la stratégie de recherche en largeur d'abord a été écrite [Kah04]. Cette version implante un système proche du système BVci, mais considère des structures modulo associativité-commutativité et utilise le filtrage associatif-commutatif du langage. D'autre part, les conditions présentées en définition 29 sont relaxées car difficile à exprimer dans le langage. Cette version est uniquement capable de prouver de petits problèmes, mais explose en mémoire dès que les structures à prouver deviennent importantes.

$$\frac{}{[a,\overline{a}]}\ ax$$

$$\frac{S\{R\}}{S[R,a,\overline{a}]}\ ai_{11}\downarrow \qquad \frac{S\{R\}}{S[a,\overline{a},R]}\ ai_{12}\downarrow \qquad \frac{S\{R\}}{S[a,R,\overline{a}]}\ ai_{13}\downarrow$$

$$\frac{S\{R\}}{S(R,[a,\overline{a}])}\ ai_{21}\downarrow \quad \frac{S\{R\}}{S([a,\overline{a}],R)}\ ai_{22}\downarrow \quad \frac{S\{R\}}{S\langle R;[a,\overline{a}]\rangle}\ ai_3\downarrow \quad \frac{S\{R\}}{S\langle[a,\overline{a}];R\rangle}\ ai_4\downarrow$$

$$\frac{S([R,W],T)}{S[(R,T),W]}\ s_{11a} \quad \frac{S([R,W],T)}{S[(T,R),W]}\ s_{12a} \quad \frac{S([R,W],T)}{S[W,(R,T)]}\ s_{13a} \quad \frac{S([R,W],T)}{S[W,(T,R)]}\ s_{14a}$$

$$\frac{S([(R,U),W],T)}{S[(R,T,U),W]}\ s_{15a} \quad \frac{S([(R,U),W],T)}{S[W,(R,T,U)]}\ s_{16a}$$

$$\frac{S[([R,W],T),P]}{S[(R,T),P,W]}\ s_{11b} \quad \frac{S[([R,W],T),P]}{S[(T,R),P,W]}\ s_{12b}$$

$$\frac{S[([R,W],T),P]}{S[W,P,(R,T)]}\ s_{13b} \quad \frac{S[([R,W],T),P]}{S[W,P,(T,R)]}\ s_{14b}$$

$$\frac{S[([(R,U),W],T),P]}{S[(R,T,U),P,W]}\ s_{15b} \quad \frac{S[([(R,U),W],T),P]}{S[W,P,(R,T,U)]}\ s_{16b}$$

$$\frac{S\langle[R,T];[U,V]\rangle}{S[\langle R;U\rangle,\langle T;V\rangle]}\ q_{11}\downarrow \quad \frac{S[\langle[R,T];[U,V]\rangle,P]}{S[\langle R;U\rangle,P,\langle T;V\rangle]}\ q_{12}\downarrow$$

$$\frac{S\langle R;T\rangle}{S[R,T]}\ q_{21}\downarrow \quad \frac{S\langle R;T\rangle}{S[T,R]}\ q_{22}\downarrow \quad \frac{S[\langle R;T\rangle,P]}{S[R,P,T]}\ q_{23}\downarrow \quad \frac{S[\langle R;T\rangle,P]}{S[T,P,R]}\ q_{24}\downarrow$$

$$\frac{S\langle[W,T];U\rangle}{S[W,\langle T;U\rangle]}\ q_{31}\downarrow \frac{S\langle[W,T];U\rangle}{S[\langle T;U\rangle,W]}\ q_{32}\downarrow \frac{S[\langle[W,T];U\rangle,P]}{S[W,P,\langle T;U\rangle]}\ q_{33}\downarrow \frac{S[\langle[W,T];U\rangle,P]}{S[\langle T;U\rangle,P,W]}\ q_{34}\downarrow$$

$$\frac{S\langle T;[W,U]\rangle}{S[W,\langle T;U\rangle]}\ q_{41}\downarrow \frac{S\langle T;[W,U]\rangle}{S[\langle T;U\rangle,W]}\ q_{42}\downarrow \frac{S[\langle T;[W,U]\rangle,P]}{S[W,P,\langle T;U\rangle]}\ q_{43}\downarrow \frac{S[\langle T;[W,U]\rangle,P]}{S[\langle T;U\rangle,P,W]}\ q_{44}\downarrow$$

FIG. 4.17: Le système BVci

Requètes	MAUDE	Gom+Tom
$[a, b, (\neg a, \neg b)]$	1 ms	4 ms
$[a, (\neg a, [\neg b, (b, [c, \neg c])])]$	70 ms	5 ms
$[a, d, (\neg a, \neg c), (\neg b, \neg d), (b, c)]$	**33 s**	142 ms
$[a, b, d, (\neg b, c), (\neg a, \neg d, \neg c)]$	**57 s**	30 ms
$[[(a, b), \neg a, \neg b], [(c, [(a, b), \neg a]), \neg b, \neg c]]$	**Overflow**	54 ms

L'utilisation de la stratégie de recherche globale permet d'éviter l'explosion combinatoire en explorant en priorité les branches « prometteuses » dans le calcul.

Nous avons implémenté une stratégie de recherche de preuve appelée *recherche globale*. Celle-ci combine les avantage de la recherche en profondeur d'abord et de la recherche en largeur d'abord. Étant donnée une liste ordonnée d'éléments, on sélectionne le premier terme, et calcule l'ensemble de ses successeurs en appliquant les règles à toutes les positions. Implanter une stratégie de recherche en largeur d'abord peut être fait en insérant les éléments de cet ensemble à la fin de la liste. Pour une stratégie en profondeur d'abord, il suffit d'insérer les éléments produits en tête de liste. Dans notre cas, les éléments de l'ensemble sont insérés dans la liste suivant un ordre particulier, qui implémente une heuristique caractérisant les structures *intéressantes* qui doivent être explorées en priorité car elle peuvent conduire à une preuve de manière efficace.

Ce mécanisme est alors répété jusqu'à l'obtention du but, et donc d'une preuve, ou à celle d'une liste d'éléments vide, auquel cas le théorème n'admet pas de preuve.

4.6. Synthèse

Les résultats présentés dans ce chapitre sont assez pratiques : la conception et l'implantation d'un langage de description de structures de données algébriques avec invariants. Cette description est alors compilée vers Java, produisant une implantation efficace et sûre de cette structure de donnée, qui de plus est prête à être utilisée en conjonction avec Tom.

4.6.1. Modules dans Gom

Le langage Gom comprend une construction de modularité : **imports**, permettant de réutiliser une spécification algébrique existante. Cependant, celle-ci est très limitée, car il est interdit d'étendre une sorte existante, ou d'ajouter des *hooks* pour des opérateurs définis dans le module importé.

Ces limitations sont nécessaires pour garantir les propriétés des fonctions de normalisation, mais pourraient être relaxées lorsque l'on peut prouver que la composition des fonctions de normalisation est possible.

Une dimension de modularité orthogonale serait de donner la possibilité d'annoter une spécification sans *hooks* avec une ou des théories à attacher aux opérateurs, et donc de générer les *hooks* correspondant au calcul de formes canoniques pour les classes d'équivalence modulo ces théories.

4.6.2. Le problème des phases

Un problème que ne résout pas Gom est celui des AST des phases de compilation. Un compilateur comme Gom fonctionne par transformations successives d'un AST. Les AST d'entrée et ceux de sortie sont très similaires : on peut rencontrer la plupart des nœuds de l'entrée en sortie, mais certains nœuds sont absents de la sortie, tandis que de nouveaux apparaissent, qui ne peuvent pas être présents en entrée. Par exemple, a la sortie du parseur de Tom, on trouvera des nœuds `Match` correspondant aux constructions de filtrage. Après compilation, ces nœuds ne doivent plus être présents, mais des nœuds `CompiledMatch` les remplacent. Cependant, les nœuds concernant le langage hôte restent inchangés. Actuellement, ces deux opérateurs sont membres d'une même sorte `Instruction`, et il est possible de construire un AST contenant les deux types de nœuds. Le compilateur pendant qu'il transforme l'AST d'entrée crée de tels arbres.

Il est important de pouvoir garantir que chaque phase de compilation produit des AST de la forme voulue. On peut par exemple vouloir s'assurer que l'AST sortant du compilateur ne comprend plus de nœuds `Match`. Ces garanties ne peuvent être fournies par un typage fort, car il est nécessaire de parfois autoriser la construction de tels arbres. On ne peut pas non plus dupliquer entièrement la signature, car la fonction de traduction devrait alors traiter de nombreux cas triviaux.

Un langage d'expressions de modularité permettrait pour ce problème de spécifier les caractéristiques des AST en entrée et sortie de phase de compilation, et de générer un vérificateur de conformité avec cette spécification. Ainsi, il deviendrait possible d'annoter chaque phase du compilateur avec les spécifications de ses entrées et sorties, et de vérifier au cours de l'exécution que cette spécification est respectée.

5. Les opérateurs associatifs et leur représentation

"What *is* a Caucus-race ?" said Alice ; not that she wanted much to know, but the Dodo had paused as if it thought that *somebody* ought to speak, and no one else seemed inclined to say anything.
"Why," said the Dodo, "the best way to explain it is to do it."

(Lewis Carroll)

L'objectif de ce chapitre est de montrer comment les opérateurs variadiques introduits lors de la présentation du langage Gom, ainsi que le filtrage associatif que permet Tom sur ces structures, présenté en section 2.2.3, peuvent être connectés aux structures de termes modulo associativité et élément neutre.

En effet, les structures concrètes que sont les opérateurs variadiques, ainsi que le filtrage de liste (ou filtrage associatif) du langage Tom sont des structures pratiques dans le cas de la programmation, et se prêtent à une implémentation efficace. Cependant, les algorithmes et structures abstraites que l'on veut manipuler sont souvent décrites comme des structures associatives avec élément neutre.

Il est alors nécessaire de bien expliciter les liens entre ces deux types de représentations, afin d'une part de justifier le qualificatif d'« associatif » donné au filtrage de liste de Tom, ainsi que de permettre l'établissement de preuves de correction des algorithmes décrits en utilisant ce filtrage et concernant des structures associatives avec élément neutre.

5.1. Opérateurs associatifs et opérateurs variadiques

Il s'agit tout d'abord de définir ce que l'on entend par opérateurs associatifs et opérateurs variadiques.

5.1.1. Signature associative

Définition 30 (Opérateur associatif)**.** *Un opérateur binaire $f : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ est associatif si il satisfait la proposition suivante :*

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

Définition 31 (Neutre à gauche)**.** *La constante e, élément de $\mathcal{T}$ est neutre à gauche pour l'opérateur binaire $f : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ si elle satisfait la proposition :*

$$\forall x \in \mathcal{T}; f(e, x) = x$$

Définition 32 (Neutre à droite)**.** *La constante $e \in \mathcal{T}$ est neutre à droite pour l'opérateur binaire $f : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ si elle satisfait la proposition :*

$$\forall x \in \mathcal{T}; f(x, e) = x$$

Définition 33 (Neutre)**.** *La constante $e \in \mathcal{T}$ est élément neutre pour l'opérateur binaire $f : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ si elle est neutre à gauche et à droite pour cet opérateur.*

Afin de définir les signatures contenant des symboles associatifs, nous allons alors étendre la définition 1 aux signatures contenant des opérateurs associatifs et neutres.

Définition 34 (Signature associative)**.** *Une signature associative est composée d'un ensemble de symboles de fonction $\mathcal{F}$, chacun étant associé à un entier naturel par la fonction arité, $ar : \mathcal{F} \to \mathbb{N}$, ainsi que d'un ensemble $\mathcal{A}$ contenant une liste de couples de symboles de fonction de $\mathcal{F}$, (f, e), avec f binaire, et e constante, tels que f est associatif, et e est son élément neutre. On notera cette signature $\mathcal{F}_\mathcal{A}$, et l'ensemble des termes engendrés par cette signature $\mathcal{T}(\mathcal{F}_\mathcal{A})$.*

On notera E la théorie équationnelle engendrée par les équations d'associativité et de neutralité des opérateurs de $\mathcal{A}$.

On peut alors définir une relation d'équivalence $=_E$ sur les termes de $\mathcal{T}(\mathcal{F}_\mathcal{A})$ telle que les termes égaux modulo associativité et élément neutre sont égaux.

5.1.2. Signature variadique

Une signature variadique est une signature algébrique comme spécifié dans la définition 1, enrichie par la présence d'opérateurs variadiques.

Définition 35 (Opérateur variadique)**.** *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, et on peut voir la définition d'un opérateur variadique f_v comme la définition d'une famille d'opérateurs algébriques $\{f_n | n \in \mathbb{N}, ar(f_n) = n\}$.*

Définition 36 (Signature variadique)**.** *Une signature variadique est composée d'un ensemble de symboles de fonctions $\mathcal{F}$, chacun étant associé à un entier naturel par la fonction arité, $ar : \mathcal{F} \to \mathbb{N}$, ainsi que d'un ensemble $\mathcal{F}_v$ de symboles de fonction variadiques.*

On notera cette signature $\mathcal{F} \bullet \mathcal{F}_v$, et l'ensemble des termes engendrés par cette signature $\mathcal{T}(\mathcal{F} \bullet \mathcal{F}_v)$.

On peut définir des notions similaires à l'associativité et élimination des éléments neutres pour les opérateurs variadiques, respectivement l'aplatissement et la suppression des vides.

Définition 37 (Aplatissement)**.** *Soit f_v un opérateur variadique. Celui satisfait l'équation d'aplatissement si :*

$$\begin{array}{l} \forall t_1, \ldots, t_n; \\ \quad f_v(t_1, \ldots, t_i, \ldots, t_n) \quad = \quad f_v(t_1, \ldots, t_{i-1}, {t'}_1, \ldots, {t'}_{n'}, t_{i+1}, \ldots, t_n) \\ \qquad\qquad\qquad\qquad\quad \textit{si } t_i = f_v({t'}_1, \ldots, {t'}_{n'}). \end{array}$$

Définition 38 (Supression des vides)**.** *Soit f_v un opérateur variadique. Celui-ci satisfait l'équation de suppression des vides si :*

$$\begin{array}{l} \forall t_1, \ldots, t_n; \\ \quad f_v(t_1, \ldots, t_{i-1}, f_v(), t_{i+1}, \ldots, t_n) \quad = \quad f_v(t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n) \\ \forall t \textit{ tel que } \nexists v_1, \ldots, v_k; t = f_v(v_1, \ldots, v_k), \\ \quad f_v(t) \qquad\qquad\qquad\qquad\qquad\quad = \quad t. \end{array}$$

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

5.2. Représentants canoniques

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, et d'autre part d'aplatissement et élimination des vides.

5.2.1. Représentants canoniques pour les termes associatifs

On oriente les équations d'associativité et élément neutre de $\mathcal{T}(\mathcal{F}_\mathcal{A})$, en un système R dans lequel on considère tous les opérateurs comme algébriques. Pour une signature $\mathcal{F}_\mathcal{A}$ avec $\mathcal{A} = \{(f, e)\}$, on a :

$$R = \left\{ \begin{array}{lcl} f(e, x) & \rightarrow & x \\ f(x, e) & \rightarrow & x \\ f(f(x, y), z) & \rightarrow & f(x, f(y, z)). \end{array} \right.$$

Ce système décrit comment calculer les représentants canoniques pour les termes contenant des opérateurs associatifs ou neutres. On notera $\mathcal{T}(\mathcal{F}_\mathcal{A}) \downarrow$ l'ensemble des représentants canoniques pour les termes de $\mathcal{T}(\mathcal{F}_\mathcal{A})$.

Note A. Ce système est terminant (ordre lexicographique sur les arguments de f). Ses paires critiques étant joignables, il est confluent.

5.2.2. Représentants canoniques pour les termes variadiques

On oriente les équations d'aplatissement et élimination des vides de $\mathcal{T}(\mathcal{F} \bullet \mathcal{F}_v)$, en un système R_v. Pour une signature $\mathcal{F} \bullet \mathcal{F}_v$ avec $\mathcal{F}_v = \{f_v\}$, on a :

$$R_v = \begin{cases} f_v(t_1, \ldots, t_{i-1}, f_v(), t_{i+1}, \ldots, t_n) & \rightarrow & f_v(t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n) \\ f_v(t_1, \ldots, t_i, \ldots, t_n) & \rightarrow & f_v(t_1, \ldots, t_{i-1}, t'_1, \ldots, t'_{n'}, t_{i+1}, \ldots, t_n) \\ & & \text{si } t_i = f_v(t'_1, \ldots, t'_{n'}) \\ f_v(t) & \rightarrow & t. \end{cases}$$

On considère encore une fois un filtrage algébrique, sans théorie, lorsque l'on applique le système de réécriture. Ici, même si l'on considère un nombre infini de règles, le filtrage algébrique est toujours décidable, car pour chaque terme à filtrer, le nombre de règles applicables est fini.

On notera $\mathcal{T}(\mathcal{F} \bullet \mathcal{F}_v) \downarrow$ l'ensemble des représentants canoniques pour les termes de $\mathcal{T}(\mathcal{F} \bullet \mathcal{F}_v)$.

5.3. Passer de terme associatif à terme variadique

Nous avons défini une notion de représentants canoniques pour les termes associatifs ainsi que pour les termes variadiques. Nous nous intéressons maintenant à la manière de représenter une signature associative par une signature variadique équivalente.

Définition 39 (Interprétation variadique des signatures)**.** *Soit une signature associative $\mathcal{F}_\mathcal{A}$. On définit l'interprétation variadique de $\mathcal{F}_\mathcal{A}$, notée $\mathcal{F} \bullet \mathcal{F}_v = [\![\mathcal{F}_\mathcal{A}]\!]$ comme la signature $\mathcal{F} \bullet \mathcal{F}_v$, avec $\mathcal{F}$ égale à la partie non associative de $\mathcal{F}_\mathcal{A}$, $\mathcal{F}_\mathcal{A} \setminus \mathcal{A}$, et $\mathcal{F}_v$ contenant un opérateur f_v pour chaque couple (f, e) de $\mathcal{A}$.*

Définition 40 (Interprétation variadique des termes)**.** *L'interprétation variadique des termes de $\mathcal{T}(\mathcal{F}_\mathcal{A})$, $[\![\mathcal{T}(\mathcal{F}_\mathcal{A})]\!]$ est définie en utilisant une fonction auxiliaire $[\ ]$ prenant en entrée un terme associatif, et retournant une liste de termes variadiques.*

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

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

La définition de la fonction $[\![\ ]\!]$ n'est pas directe. En effet, on doit traduire « l'épine dorsale » d'un terme contenant une suite de f en une liste d'arguments pour l'opérateur variadique f_v. Ceci est nécessaire afin de pouvoir prouver l'injectivité de cette fonction $[\![\ ]\!]$.

Théorème 5 (Injectivité $[\![\ ]\!]$). *La fonction de représentation variadique $[\![\ ]\!]$ est injective.*

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

Démonstration. La preuve se fait par induction sur la structure de t_1 et t_2 en considérant comme hypothèse d'induction :

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

□

5.4. Liens entre termes variadiques et termes associatifs

Jusqu'ici, nous avons simplement posé les définitions des différents objets qui sont manipulés : les termes associatifs sont courants dans les spécifications algébriques, et les termes variadiques, ou listes, sont usuellement considérés pour l'implantation et la représentation de tels objets algébriques.

Nous voulons alors établir une relation entre les termes associatifs et leur interprétation variadique, puis montrer que les formes normales modulo associativité et élément neutre d'une part, et aplatissement et suppression des neutres d'autre part sont liées de la même manière. Le diagramme de la figure 5.1 présente cet objectif de manière informelle.

On définit les fonctions de normalisation effectuant « peignage » et élimination des éléments neutres pour les termes associatifs, ainsi que aplatissement et élimination des vides pour les termes variadiques.

Définition 41 (Peigne). *La fonction peigne : $\mathcal{T}(\mathcal{F}_\mathcal{A}) \to \mathcal{T}(\mathcal{F}_\mathcal{A})$ effectue la normalisation de tout terme de $\mathcal{T}(\mathcal{F}_\mathcal{A})$ en appliquant les règles d'associativité associées aux opérateurs associatifs.*

Définition 42 (Suppression neutres). *La fonction $supprA : \mathcal{T}(\mathcal{F}_\mathcal{A}) \to \mathcal{T}(\mathcal{F}_\mathcal{A})$ effectue la normalisation de tout terme de $\mathcal{T}(\mathcal{F}_\mathcal{A})$ en appliquant les règles d'élimination des éléments neutres de la signature.*

Définition 43 (Aplatissement). *La fonction aplat : $\mathcal{T}(\mathcal{F} \bullet \mathcal{F}_v) \to \mathcal{T}(\mathcal{F} \bullet \mathcal{F}_v)$ effectue la normalisation de tout terme de $\mathcal{T}(\mathcal{F} \bullet \mathcal{F}_v)$ en appliquant les règles d'aplatissement associées aux éléments de $\mathcal{F}_v$.*

Définition 44 (Suppression vides). *La fonction $supprV : \mathcal{T}(\mathcal{F} \bullet \mathcal{F}_v) \to \mathcal{T}(\mathcal{F} \bullet \mathcal{F}_v)$ effectue la normalisation de tout terme de $\mathcal{T}(\mathcal{F} \bullet \mathcal{F}_v)$ en appliquant les règles d'élimination des éléments vides de la signature.*

On peut alors énoncer les propriétés de commutation.

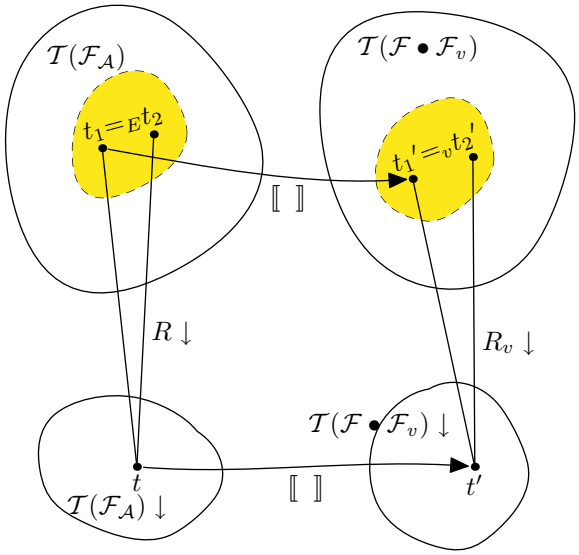


FIG. 5.1: Représentation informelle des liens entre termes associatifs et termes variadiques

Théorème 6 (Commutations)**.** *Les fonctions peigne et aplat commutent avec la fonction de représentation variadique.*

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

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

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

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.

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

On peut résumer ce théorème par le diagramme suivant :

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

On a donc montré que les représentants canoniques par les fonctions de normalisation associées aux théories des objets de $\mathcal{T}(\mathcal{F}_\mathcal{A})$ et des objets associés par la fonction $[\![\ ]\!]$ sont eux aussi associés par la fonction $[\![\ ]\!]$.

Les fonctions *peigne*, *supprA*, *aplat* et *supprV* sont définies comme des fonctions transformant tour à tour des objets de $\mathcal{T}(\mathcal{F}_\mathcal{A})$ et $\mathcal{T}(\mathcal{F} \bullet \mathcal{F}_v)$. Le diagramme commutatif établi jusque là, pour être utile, doit être complété par la relation entre l'équivalence modulo associativité et élément neutre $=_E$ et ces fonctions.

Théorème 7 (Complétude $supprA \circ peigne$)**.**

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

Théorème 8 (Correction $supprA \circ peigne$)**.**

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

Théorème 9 (Complétude $supprV \circ aplat$)**.**

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

Théorème 10 (Correction $supprV \circ aplat$)**.**

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

L'utilisation de ces théorèmes nous permet de prouver le diagramme de la figure 5.2.

Les termes de $\mathcal{T}(\mathcal{F}_\mathcal{A})$ 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 normalisation par aplatissement et élimination des vides vers un représentant canonique dans $\mathcal{T}(\mathcal{F} \bullet \mathcal{F}_v)$ 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.

5.5. Preuves, utilisant Coq

Nous n'avons pas donné dans les sections précédentes de preuves détaillées pour les différents théorèmes. L'un des intérêts de ce travail, effectué en collaboration avec Damien Doligez, outre le fait de formaliser les liens entre opérateurs associatifs avec élément neutre et opérateurs variadiques, est d'avoir utilisé un assistant de démonstration automatique pour formaliser le problème ainsi que développer les démonstrations.

Nous ne décrirons pas ici l'ensemble des preuves, mais porterons notre attention essentiellement sur la formalisation du problème, ainsi que des différents théorèmes.

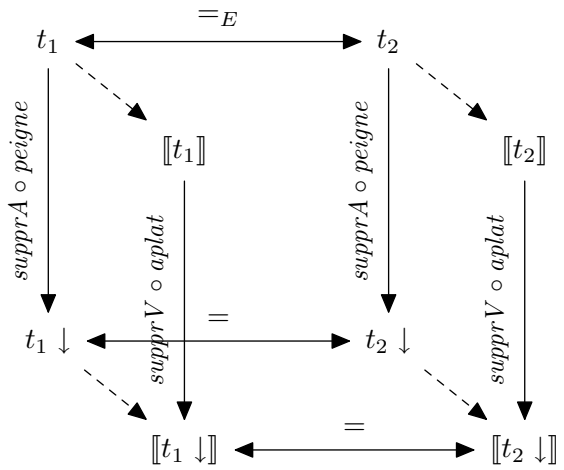


FIG. 5.2: Liens entre classes d'équivalence et formes normales sur $\mathcal{T}(\mathcal{F}_{\mathcal{A}})$ et $\mathcal{T}(\mathcal{F} \bullet \mathcal{F}_v)$

5.5.1. Les fonctions de normalisation et transformation

On commence par définir l'algèbre de termes associative $\mathcal{T}(\mathcal{F}_{\mathcal{A}})$. On considère ici que la signature ne contient qu'un seul opérateur associatif `F`, et un seul élément neutre pour `F`, `E`. Les autres membres de la signature sont abstraits par un constructeur `C` prenant un entier en argument. Ajouter ici des constructeurs algébriques prenant plusieurs arguments ne complique pas les preuves, ajoutant simplement des cas triviaux aux traitements récursifs. Il est alors plus simple de l'éliminer.

```
Inductive T : Set :=
  | E : T
  | C : nat -> T
  | F : T -> T -> T
.
```

On peut alors définir les fonctions de normalisation `peigne` et `supprT`. Elles sont obtenues comme des points fixes sur le type `T` représentant les termes associatifs.

```
Fixpoint peigne (t : T) : T :=
  match t with
  | E => E
  | C x => C x
  | F x y => peigne2 x (peigne y)
  end
with peigne2 (t1 : T) (t2 : T) {struct t1} : T :=
  match t1 with
  | E => F t1 t2
  | C x => F t1 t2
  | F x y => peigne2 x (peigne2 y t2)
  end
```

```
.

Fixpoint supprT (t : T) : T :=
  match t with
    | E => E
    | C x => t
    | F x y =>
       match supprT x with
       | E => supprT y
       | xx =>
          match supprT y with
          | E => xx
          | yy => F xx yy
          end
       end
  end
.
```

On définit alors le type `U` comme le type des représentants variadiques pour `T`. On représente l'opérateur variadique comme un constructeur prenant en argument une liste de termes de `U`. On conserve un constructeur `Ev` représentant l'élément neutre. Celui-ci doit être identifié avec l'opérateur variadique sans argument : on prendra garde dans les fonctions de normalisation d'identifier ces deux termes. Il est cependant plus utile de le conserver dans la définition de `U` afin de simplifier les démonstrations.

```
Inductive U : Set :=
  | Ev : U
  | Cv : nat -> U
  | Fv : Ulist -> U
with Ulist : Set :=
  | Cons : U -> Ulist -> Ulist
  | Nil : Ulist
.
```

On définit alors la fonction `UofT`, qui est l'implémentation de la fonction $[\![\ ]\!]$, et doit faire correspondre à tout terme de `T` un terme de `U`.

```
Fixpoint UofT (t : T) : U :=
  match t with
    | E => Ev
    | C x => Cv x
    | F x y => Fv (Cons (UofT x) (UofT2 y))
  end
with UofT2 (t : T) : Ulist :=
  match t with
  | F x y => Cons (UofT x) (UofT2 y)
```

```
  | E => Cons Ev Nil
  | C x => Cons (Cv x) Nil
  end
.
```

Cette définition est concordante avec celle de la fonction ⟦ ⟧ donnée en définition 40, et prend garde de traduire un enchaînement à droite de `F` en une liste d'arguments pour le constructeur `Fv`.

On montre alors que cette fonction est injective, en utilisant un lemme légèrement plus fort introduisant l'injectivité de la fonction `UofT2` qui correspond à [ ].

```
Lemma inj2 : forall t1 t2,
  (UofT t1 = UofT t2 -> t1 = t2) /\ (UofT2 t1 = UofT2 t2 -> t1 = t2).

Theorem injection : forall t1 t2,
  UofT t1 = UofT t2 -> t1 = t2.
```

On définit aussi les fonctions d'aplatissement `aplat` et d'élimination des vides, `supprU`. Celles-ci sont construites de manière similaire aux fonctions `peigne` et `supprT`. Ces définitions sont cependant légèrement plus compliquées. En effet, les termes variadiques étant constitués d'une liste d'arguments associée au constructeur `Fv`, la construction récursive de la fonction effectuant l'aplatissement doit être annotée par l'argument décroissant lors de l'appel récursif, afin de permettre au système de vérifier que cette fonction termine bien.

```
Fixpoint aplat_list (us tl : Ulist) {struct us} : Ulist :=
  match us with
  | Nil => tl
  | Cons x us' => aplat2 x (aplat_list us' tl)
  end
with aplat2 (x : U) (us : Ulist) {struct x} : Ulist :=
  match x with
  | Ev => Cons x us
  | Cv _ => Cons x us
  | Fv us1 => aplat_list us1 us
  end
.

Definition aplat (u : U) : U :=
  match u with
  | Ev => Ev
  | Cv x => Cv x
  | Fv us => Fv (aplat_list us Nil)
  end
.
```

La définition de la fonction `supprU` éliminant les vides est construite comme celle de la fonction `supprT`. Elle doit aussi prendre en compte le fait que le constructeur `Ev` est utilisé pour remplacer la liste vide `Fv` avec une liste vide en argument.

On peut alors établir les différents théorèmes.

5.5.2. Les théorèmes

Les premiers théorèmes à établir sont les résultats de commutation entre les différentes fonctions de normalisation avec la fonction de représentation variadique.

```
Theorem comm1 : forall t : T, aplat (UofT t) = UofT (peigne t).
Theorem comm2 : forall t : T, supprU (UofT t) = UofT (supprT t).
```

On doit alors établir la connexion entre les fonctions de normalisation *peigne* et *supprA* et la relation d'équivalence $=_E$ engendrée par les axiomes d'associativité et élément neutre.

Cette théorie équationelle est donnée par la définition inductive suivante, spécifiant que `equa` est une relation réflexive, symétrique, transitive, stable par substitution, et satisfaisant les équations d'associativité de `F`, avec `E` son élément neutre.

```
Inductive equa : T -> T -> Prop :=
  | refl : forall t : T, equa t t
  | sym : forall t1 t2 : T, equa t1 t2 -> equa t2 t1
  | trans : forall t2 t1 t3 : T,
             equa t1 t2 -> equa t2 t3 -> equa t1 t3
  | subst : forall t1 t2 t3 t4,
             equa t1 t3 -> equa t2 t4 -> equa (F t1 t2) (F t3 t4)
  | assoc : forall t1 t2 t3 : T,
             equa (F (F t1 t2) t3) (F t1 (F t2 t3))
  | neutre_l : forall t : T, equa (F E t) t
  | neutre_r : forall t : T, equa (F t E) t
.
```

On peut alors montrer les résultats de correction et complétude des fonctions *peigne* et *supprA* par rapport à la relation d'équivalence $=_E$.

```
Theorem completT :
  forall t1 t2 : T,
    equa t1 t2 -> supprT (peigne t1) = supprT (peigne t2).

Theorem correct_peigne : forall t : T, equa t (peigne t).
Theorem correct_supprT : forall t : T, equa t (supprT t).
```

On montre aussi les résultats équivalents pour les fonctions *aplat* et *supprV* sur les éléments de $\mathcal{T}(\mathcal{F} \bullet \mathcal{F}_v)$ et la relation d'équivalence $=_E$.

```
Theorem completU : forall t1 t2 : T,
  equa t1 t2 -> supprU (aplat (UofT t1)) = supprU (aplat (UofT t2)).
```

```
Theorem correct_aplat :
  forall t1 : T, exists t2 : T,
    aplat (UofT t1) = UofT t2 /\ equa t1 t2.

Theorem correct_supprU :
  forall t1 : T, exists t2 : T,
    supprU (UofT t1) = UofT t2 /\ equa t1 t2.
```

La correction des fonctions *aplat* et *supprV* s'obtient en utilisant les résultats de commutation ainsi que les corrections respectives des fonctions *peigne* et *supprA*, tandis que la complétion s'obtient grâce aux résultats de commutation et au théorème `completT`.

Nous ne donnons pas ici les preuves des différents théorèmes dans le système Coq. La plupart d'entre elles se font par induction sur la structure des termes, en utilisant de nombreux lemmes intermédiaires, comme le fait que les différentes fonctions sont idempotentes, ainsi que les propriétés des termes « peignés ». Les scripts de preuve complets sont disponibles dans l'annexe A.1.

5.6. Synthèse

Nous avons montré dans ce chapitre 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 mise 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, comme ceux qui sont 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.

Une dernière étape serait de certifier que l'algorithme de filtrage « de liste » que génère le compilateur Tom pour des filtres de listes comme décrits en section 2.2.3, pour montrer qu'il est correct et complet par rapport au filtrage associatif avec élément neutre. Ceci permettrait de certifier l'implémentation de réécriture modulo associativité et élément neutre.

6. Stratégies de réécriture réflexives en Java

"Begin at the beginning", the King said, gravely, "and go till you come to the end ; then stop."

(Lewis Carroll)

6.1. Langages de stratégies

La définition de *stratégies* pour l'application des règles est d'un intérêt tout particulier pour les systèmes à base de règles. En effet, la combinaison de règles pour former des transformations complètes requiert de pouvoir exercer un certain contrôle sur l'application des règles.

Les systèmes à base de règles appliquent traditionnellement les règles suivant une stratégie de contrôle standard, qui dépend du système. Cependant, il est généralement nécessaire de contrôler plus finement l'application des règles lors du développement d'applications ou de transformations complexes. C'est pourquoi les systèmes de transformations basés sur la réécriture ont adopté différents mécanismes pour contrôler l'application des règles.

6.1.1. Ce qu'on veut pouvoir exprimer

L'objectif des stratégies est de permettre de spécifier les radicaux sur lesquels il faut ou ne faut pas appliquer les règles. Cette spécification permet de séquencer les applications des règles de réécriture, rendant ainsi confluent un système de réécriture qui peut ne pas l'être. La stratégie de contrôle standard des systèmes basés sur la réécriture est habituellement la stratégie *leftmost-innermost*. Avec cette stratégie, le radical situé le plus à gauche et le plus profond dans le terme est systématiquement sélectionné pour l'application d'une règle de réécriture, puis le processus est répété jusqu'à l'obtention d'un point fixe. D'autres stratégies communes sont *TopDown* et *BottomUp*, pour lesquelles les règles sont appliquées du radical le moins profond dans le terme vers le radical le plus profond, généralement en donnant la priorité aux radicaux de gauche, et inversement. La stratégie *leftmost-innermost* est la plus intéressante, car lorsque le système de réécriture est terminant et confluent, alors les formes normales sont égales aux termes obtenus par le calcul sous cette stratégie.

De plus, le contrôle de la réécriture, c'est-à-dire le contrôle de la manière d'appliquer les règles doit être dissocié des règles elles-mêmes, afin de permettre de raisonner séparément sur les règles de transformation et la façon d'effectuer ces transformations.

Traversée et transformation d'arbres

Pour des applications comme la compilation et la transformation de programmes, il est nécessaire de pouvoir exercer un contrôle plus fin sur l'application des règles. Par exemple, le traitement des déclarations et utilisations des variables peut nécessiter de rechercher à partir de la racine une déclaration de variable, pour ensuite appliquer un traitement particulier au sous-arbre de cette déclaration, comme propager l'information de type de cette variable, ou effectuer une propagation de constante.

On a alors besoin de pouvoir spécifier de manière souple la manière de rechercher les radicaux sur lesquels appliquer les règles, et dans quel ordre appliquer ces règles. On doit aussi être capable de spécifier des séquences d'actions à exécuter. Les stratégies *TopDown*, *BottomUp* et *leftmost-innermost* sont alors des instances de ces stratégies de traversée.

Gérer le non-déterminisme

Un autre type de problème qui nécessite de pouvoir contrôler l'application des règles est la gestion du non-déterminisme. Il y a deux sources de non-déterminisme lors de l'application d'un système de règles :

- le non-déterminisme du filtrage, lorsque l'on considère du filtrage équationnel non-unitaire, avec des théories associatives, associatives-commutatives, *etc.*
- le non-déterminisme introduit par la possibilité d'appliquer une ou des règles à différentes positions dans un terme.

Les stratégies de traversées permettent de limiter l'impact de la seconde source, en spécifiant quels radicaux doivent être réduits, et par quelles règles. Cependant, il est parfois nécessaire d'explorer tous les résultats atteignables par l'application d'une règle à quelque position que ce soit, ou sur un sous ensemble des positions. C'est le cas par exemple lorsque l'on veut faire de la vérification de modèles, comme dans l'exemple de la vérification du protocole d'authentification à clés publiques Needham-Schroeder présenté en section 4.3.2, ou de la recherche de preuves comme dans le calcul des structures présenté en section 4.5.1.

Les stratégies communes pour gérer le non-déterminisme sont :

- Le *backtracking*, propre à Prolog, ELAN, permettant d'implanter des recherches en profondeur d'abord, c'est-à-dire que l'on commence à explorer les successeurs du terme obtenu par réécriture d'un radical à la recherche d'un résultat satisfaisant. Si ce résultat n'est pas obtenu, alors le système reviendra en arrière, et explorera les successeurs de l'application d'une règles sur un autre radical ou de l'application de la même règle sur le même radical, mais avec une solution de filtrage différente, dans le cas de filtrage non-unitaire.
- La recherche dite « en largeur d'abord », comme utilisé dans Murphi, fonctionne, à l'inverse, en collectant tout d'abord les successeurs du terme couramment examiné par l'application d'une règle à n'importe quelle position. Si le but est trouvé dans cet ensemble, la recherche s'arrête. Sinon, les successeurs directs de chaque élément de l'ensemble sont collectés et examinés.

Ces stratégies sont généralement les seules implantées dans les systèmes à base de règles comme Prolog ou ELAN. Elles doivent être combinées avec des stratégies de traversée pour limiter le nombre de successeurs à chaque étape, en éliminant des radicaux ou des règles qui ne peuvent pas mener au résultat cherché. Toutefois, ces stratégies ont chacune des inconvénients : une stratégie en profondeur d'abord peut ne pas terminer si le système considéré comporte des branches de calcul qui ne terminent pas, tandis qu'une recherche en largeur d'abord peut consommer énormément d'espace mémoire, ce qui se produit lorsque le non-déterminisme est important au niveau de chaque règle, par exemple lorsque l'on utilise du filtrage modulo associativité-commutativité.

On peut aussi considérer des stratégies plus complexes, comme la recherche *globale* présentée en section 4.5.6, qui permet d'utiliser une heuristique pour décider de l'ordre d'exploration des différentes branches de calcul basée sur un ordre sur les termes.

6.1.2. Tour d'horizon

De nombreux langages ont proposé des constructions permettant de spécifier des stratégies ou tactiques contrôlant l'application de règles. Ces constructions ont été introduites principalement pour les systèmes basés sur la réécriture, ainsi que dans le domaine de la transformation de programmes.

Les parseurs d'arbres

Cette approche a été développée dans le domaine de la génération de code, dans l'objectif de permettre l'écriture facile de générateurs de code dans un compilateur. Dans cette approche, les règles sont décrites par les règles d'une grammaire d'arbres utilisée pour recouvrir un arbre avec les règles applicables, et exécuter les actions correspondantes. Un ordre de traversée de l'arbre est ainsi dérivé de la spécification de la grammaire d'arbre.

Cette technique est utilisée par le générateur de parseurs ANTLR [PQ95] pour générer des traversées d'arbres, appelées *tree walkers* à partir d'une grammaire d'arbre. Les transformations d'arbre ou traductions sont spécifiées en associant des actions aux productions de la grammaire. De cette spécification est généré un ensemble de fonctions mutuellement récursives qui définissent une traversée en une passe de l'arbre.

Il n'est pas possible de définir une traversée qui parcours un même nœud de multiples fois. De plus, les règles de transformation sont intégrées aux productions de la grammaire, et la séparation des règles et des stratégies n'est pas supportée. Les traversées génériques ne sont pas non plus possibles, et il est nécessaire de spécifier une grammaire complète pour les arbres.

Maude

Le langage de spécification MAUDE se base sur la logique de réécriture. Le contrôle de l'application des règles est rendu possible par la réflexivité du système, qui rend possible la modification du système de réécriture utilisé par un système de réécriture utilisant ces aspects réflexifs.

Elan

Le langage ELAN est un langage de spécification qui a été conçu au sein du projet Protheo, et implanté par Marian Vittek et décrit tout d'abord dans sa thèse de doctorat [Vit94]. Celui-ci introduisait la notion de stratégie. Cette notion à ensuite été étendue par Peter Borovanský [Bor98] en un langage de stratégies très expressif, qui a été intégré à l'interpréteur ELAN. Un compilateur efficace pour le langage ELAN « de Marian » a été développé simultanément par Pierre-Étienne Moreau [Mor99], et étendu pour supporter une grande partie du langage de stratégies décrit par Peter Borovanský.

Le langage Elan de base : Le langage de spécification ELAN [Vit94, BKK+98] a introduit la notion de langage de stratégies, en utilisant un langage de combinateurs permettant à l'utilisateur de composer des stratégies.

Dans cette approche, une spécification consiste en un ensemble de règles de réécriture non nommées, un ensemble de règles de réécriture nommées ainsi qu'un ensemble de définitions de stratégies. Les règles non nommées sont appliquées sur tous les termes avec la stratégie *leftmost-innermost* dès que possible. Les stratégies correspondent à des programmes qui tentent de transformer un terme en un autre. Elles peuvent réussir ou échouer dans cette tâche, et lorsqu'elles réussissent, le résultat de leur application est le terme transformé. Les stratégies élémentaires sont les règles de réécriture nommées, identifiées grâce à leur nom, ainsi que l'identité `id` et l'échec `fail` qui respectivement laisse le terme inchangé ou échoue toujours.

Ainsi, la règle nommée `inc` permet d'incrémenter un nombre.

```
[inc] N => N+1  end
```

Ces stratégies élémentaires sont combinées pour former des stratégies plus complexes en utilisant un certain nombre de combinateurs de stratégies. Ces combinateurs permettent de sélectionner l'ordre d'application des règles, ainsi que de gérer les multiples résultats dûs au non-déterminisme, c'est-à-dire de choisir certaines solutions dans l'ensemble des solutions possibles. Les principaux combinateurs sont :

- la *séquence* de stratégies, `s1 ; s2` pour laquelle on tente tout d'abord d'appliquer la stratégie `s1`, puis d'appliquer `s2` à tous les résultats en cas de succès.
- l'opérateur `dk`, pour *don't know choose*, `dk(s1,...,sn)` qui donne tous les résultats de l'application de toutes les stratégies `s1`,...,`sn` sur un terme ;
- l'opérateur `dc one`, pour *don't care choose one*, `dc one(s1,...,sn)` qui donne un seul des résultats de l'application d'une stratégie `sk` qui n'échoue pas, la méthode de choix de la stratégie `sk` étant non-déterministe ;
- La *répétition*, `repeat*(s)` et sa variante `repeat+(s)` qui répètent l'application d'une stratégie, jusqu'à ce qu'elle ne soit plus applicable. `repeat*(s)` n'échoue jamais, et renvoie l'identité si `s` n'est pas applicable sur le sujet, alors que `repeat+(s)` impose que `s` puisse être appliqué au moins une fois.

Le non-déterminisme est généralement dû au filtrage équationnel, modulo associativité-commutativité, ou au non-déterminisme des clauses `where` qui constituent des gardes

à l'application des règles. Le langage permet d'être plus précis dans la gestion du non-déterminisme, en introduisant les combinateurs :

- `dc(s1,...,sn)` qui donne tous les résultats de l'application d'une stratégie `sk` qui n'échoue pas sur un terme. Cette stratégie `sk` est choisie de manière non-déterministe ;
- `first(s1,...,sn)` qui donne tous les résultats de l'application de la première stratégie `sk` qui n'échoue pas ;
- `first one(s1,...,sn)` qui donne le premier résultat de la première stratégie `sk` qui n'échoue pas ;
- `iterate*(s)`, qui correspond à `dk(id, s, s ;s, s ;s ;s,...)`, et donc énumère les résultats des applications successives de la stratégie `s`.

Ainsi, pour énumérer les nombres entiers, une stratégie `enumerate` peut être définie par :

```
[] enumerate => iterate*(dc one(inc)) end
```

Appliquée sur l'entier 0, cette stratégie énumérera l'ensemble des nombres entiers.

Les stratégies élémentaires, les règles nommées, ne s'appliquent qu'à la tête du terme sur lequel elles sont appliquées, de même que les combinateurs, qui ne permettent pas de traverser les termes. Les traversées sont encodées dans les clauses `where` des définitions de règles, qui permettent d'instancier des variables locales au membre droit de la règle en utilisant le résultat de l'application d'une stratégie à l'une des variables de filtrage. Ce mécanisme, bien que souple, ne permet pas de séparer la définition des règles de celle des stratégies, car il faut connaître les stratégies pour écrire les clauses `where` des règles, et ces règles ne sont pas réutilisables avec d'autres stratégies.

Exemple 24. Ainsi, pour incrémenter tous les nombres contenus dans des arbres crées à l'aide de deux constructeurs f et g binaires, en réutilisant la définition de la stratégie `inc`, on devrait ainsi écrire :

```
[recinc] f(x,y) => f(u,v) where u:=(dc(inc,recinc))u
                          where v:=(dc(inc,recinc))v
                          end
[recinc] g(x,y) => g(u,v) where u:=(dc(inc,recinc))u
                          where v:=(dc(inc,recinc))v
                          end
```

Il faut alors définir une règle effectuant la traversée des termes pour chaque symbole de la signature algébrique manipulée.

Le langage Elan étendu : Le système de stratégies d'ELAN, bien que puissant, comportait certaines lacunes. En particulier, il était impossible de spécifier des traversées de termes sans en encoder une partie dans les règles elles-mêmes, ou de définir des stratégies d'évaluation comme l'appel par nom ou par valeur.

Peter Borovanský, dans sa thèse de doctorat [Bor98], définit un langage de stratégies « étendu ». Dans celui-ci les stratégies sont vues comme des règles de réécriture, ce qui permet des définitions récursives.

Exemple 25. On peut définir la répétition jusqu'à l'obtention d'un point fixe, ou la boucle conditionnelle, avec une condition sans effet de bord avec les définitions récursives :

```
strategies
  try(s)      = first one(s, id)
  repeat(s)   = try(s ; repeat(s))
  while(c, s) = try(test(c); s; while(c, s))
```

Deux nouveaux mécanismes sont fournis pour décrire des traversées de termes : une stratégie `normalize(R1,...Rn)` qui normalise un terme suivant les règles `R1,...Rn` en utilisant la stratégie *leftmost-innermost*, ainsi que des *opérateurs de congruence*. Les opérateurs de congruence définissent un ensemble de stratégies élémentaires dépendant de la signature des termes employée. Pour chaque opérateur F d'arité n, un combinateur de stratégies de la forme `F(s1,...sn)` est fourni. Il s'applique aux termes de la forme $F(t_1, \ldots, t_n)$ en appliquant chaque stratégie `si` au terme t_i correspondant, ce qui constitue une généralisation de ce qui est présenté dans l'exemple 24. Une utilisation typique de ces opérateurs est de définir l'opérateur `map(s)` sur les listes, qui sont représentées par la liste vide `nil` et l'insertion par un simple espace, par la définition de stratégie `[map(s)]t = [dc one(nil, cons(s,map(s)))]t`, dans laquelle `[s]t` représente l'application de la stratégie `s` au terme `t`.

Ce langage très riche définit de nombreux combinateurs additionnels, permettant de gérer le non-déterminisme des règles et ensembles de règles de manière très fine. Ceci, combiné avec les définitions récursives de stratégies et le mécanisme de *backtracking* offre une grande expressivité. Ce langage de stratégies offre par ailleurs des aspects réflexifs, et il est possible, dans une certaine mesure, d'utiliser des règles de réécriture pour transformer des stratégies, les règles étant quant à elles statiques. Il est aussi possible de considérer les termes de manière non typée, comme application d'un symbole à une liste d'arguments pour définir des stratégies plus génériques.

Cependant, ce langage, bien que très complet et expressif, est aussi très difficile à utiliser, et les coûts des différentes constructions, tant en temps qu'en espace, sont très difficile à évaluer.

Stratego

L'approche par combinaison de stratégies élémentaires, en autorisant les stratégies récursives a inspiré la conception du langage de transformations de programmes Stratego [VBT98]. La syntaxe et l'ensemble des combinateurs sont différents, mais les idées derrière ces deux langages de stratégies sont similaires. L'apport de Stratego est de simplifier de manière drastique le langage de stratégies, en limitant le nombre de combinateurs basiques à seulement 7, et en supprimant le typage des termes manipulés. Ceci est fait au détriment de la gestion du non-déterminisme, car les règles de Stratego sont décrites avec un filtrage unitaire, et le *backtracking* est uniquement local aux opérateurs de choix. Ces limitations ont été introduites dans le but de simplifier le langage de stra-

tégies, son implantation et son efficacité, et aussi car le domaine de la transformation de programmes ne nécessite pas une gestion fine du non-déterminisme.

Ces stratégies de base sont proches de celles présentées pour ELAN, la composition séquentielle, les choix déterministe et non-déterministe, l'identité, l'échec, auxquels on ajoute le test et la négation. Le test `test(s)` permet de tester si la stratégie `s` s'applique ou échoue, et renvoie l'identité en cas de réussite. La négation `not(s)` fait l'inverse, et renvoie l'identité si l'application de `s` échoue, et échoue sinon. La définition récursive de stratégies est rendue possible par un opérateur de récursion $\mu x(s)$, qui permet d'utiliser le nom x pour désigner la stratégie récursive dans la définition de s. La définition de la stratégie `repeat(s)` est alors `repeat(s) =` μx`(try(s ; x))`.

On peut remarquer que contrairement à ELAN, le choix non-déterministe de Stratego ne définit qu'un *backtracking* local, et non global. Cela signifie que lorsque le choix a été fait, il ne pourra pas être remis en cause par un échec plus tard dans le calcul.

Un autre apport du langage Stratego est la définition de combinateurs de stratégies permettant la définition de traversées d'arbres génériques, s'ajoutant aux opérateurs de congruence. L'opération fondamentale pour les traversées de termes est alors l'application d'une stratégie sur un sous-terme direct donné d'un terme. La stratégie `i(s)` applique la stratégie `s` au fils en position i dans le sujet, et échoue si i est supérieur à l'arité du symbole de tête du sujet. De plus, trois opérateurs de congruence générique sont ajoutés, permettant la définition de traversées d'arbre génériques :

- `All(s)`, qui applique la stratégie `s` à tous les fils du terme sujet. La stratégie échoue si l'une des applications échoue, mais n'échoue jamais sur une constante (il n'y a pas de sous-terme sur lequel l'application de `s` échoue).
- `One(s)`, qui applique la stratégie `s` au premier fils du sujet pour lequel cette application n'échoue pas. Elle échoue s'il n'y a pas de fils sur lequel l'application de `s` n'échoue pas ; elle échoue toujours sur une constante (il n'y a pas de sous-terme sur lequel l'application de `s` réussit).
- `Some(s)`, qui est un hybride entre `All` et `One`, applique `s` sur tous les fils du sujet pour lesquels cette application n'échoue pas, et échoue s'il n'y a pas au moins un fils sur lequel la stratégie `s` a pu être appliquée.

La combinaison de ces stratégies de décomposition de termes génériques et de l'opérateur de récursion permet de décrire simplement les stratégies comme `BottomUp(s)` ou `TopDown(s)`, respectivement par μx`(All(`x`) ; s)` et μx`(s ; All(`x`))`, mais aussi de construire des stratégies de parcours spécifiques à une application. Par contre, il est difficile d'explorer un espace de recherche.

ASF+SDF

L'environnement de transformations de programmes ASF+SDF suggère une autre manière de résoudre le problème des traversées de termes, en proposant des fonctions de traversée générique [BKV03]. Ces fonctions peuvent être des traversées des racines aux feuilles des arbres, ou inversement, et peuvent aussi traverser tous les nœuds d'un arbre ainsi qu'arrêter la traversée à une certaine profondeur ou quand un motif particulier est trouvé.

Cette approche distingue trois types de traversées, selon qu'elles décrivent une transformation, un accumulateur, permettant de récolter de l'information dans l'arbre sans le modifier, ou une traversée ayant ces deux caractéristiques.

Exemple 26. Pour illustrer l'utilisation de ces fonctions de traversée, définissons une fonction incrémentant toutes les valeurs entières contenues dans un arbre. La sorte `TREE` ainsi que les constructeurs des arbres sont définis par ailleurs dans un module `Tree-syntax`. La fonction `inc` est appliquée sur tous les nœuds de l'arbre de manière *BottomUp*, et l'application se poursuit après la réussite d'une application.

```
module Tree-inc
imports Tree-syntax
exports
context-free syntax
inc(TREE) -> TREE {traversal(trafo,bottom-up,continue)}
equations
[1] inc(N) = N + 1
```

La transformation `inc` est alors définie, et ainsi, sur l'entrée : `inc(f(g(f(1,2),3),4))`, on obtient en sortie `inc(f(g(f(2,3),4),5))`.

Dans cette approche, seules les notions de traversée sont traitées, et les notions de répétition, conditions et contrôle doivent être traitées au cœur des règles de transformation elles-mêmes. Ainsi, les règles et le contrôle de leur application sont intimement liés, limitant les possibilités de réutilisation. Surtout, les règles et stratégies étant liées, il est difficile de raisonner sur les règles indépendamment de leur stratégie d'application. Ce point est sujet a discussion, car en effet, nombre de transformations sont constituées de règles n'ayant pas de propriétés particulières lorsque considérées séparément, mais qui possèdent des propriétés comme terminaison et confluence lorsqu'elles sont utilisées sous une stratégie d'application particulière. Cela explique aussi pourquoi la séparation des règles et du contrôle n'est pas un facteur suffisant pour entraîner une réutilisabilité systématique des règles et stratégies : il est courant que des règles n'aient un intérêt que lorsqu'elles sont utilisées avec une stratégie particulière.

La définition des fonctions de traversée génériques a l'intérêt essentiel de permettre au langage de rester simple et facile à utiliser, tout en couvrant les besoins de traversée d'arbres et les besoins spécifiques à l'analyse et transformation de programmes, comme la collecte d'information dans un AST.

JJTraveler

La bibliothèque JJTraveler [Vis01] a pour objectif d'adapter le modèle de stratégies de traversée composables proposé pour Elan et Stratego à un langage orienté objet, comme Java. Il se base sur le constat que la méthode habituelle pour décrire des traversées d'arbres dans les langages orientés objet est généralement le patron de conception *visiteur*. Les limitations principales de la traversée d'arbres avec des visiteurs sont d'une

part le manque de contrôle sur la traversée et d'autre part la difficulté de combiner plusieurs visiteurs entre eux. En effet, la stratégie de parcours est soit intégrée au code du visiteur lui-même, soit dans la méthode `accept` des classes représentant l'arbre. Le premier cas impose la redéfinition du parcours pour chaque visiteur, tandis que dans le second cas, aucun contrôle sur la traversée ne peut être exercé, mis à part le choix entre plusieurs stratégies prédéfinies. La combinaison de stratégies de parcours est aussi impossible. Une approche pour séparer la programmation des visiteurs de celle des stratégies de parcours peut être d'utiliser la programmation orientée aspects [KLM+97], et de faire de la stratégie de parcours un aspect distinct des visiteurs.

JJTraveler propose une solution à ces limitations par l'introduction d'un ensemble de combinateurs de visiteurs (*visitor combinators*), pouvant être utilisés pour construire de nouveaux visiteurs à partir de visiteurs existants. Ces combinateurs réutilisables et génériques sont inspirés des primitives de stratégies de Stratego. Le support pour les combinateurs de visiteurs par une structure d'arbre se fait par l'implantation d'une interface `Visitable` qui décrit non seulement comment visiter un nœud, mais aussi comment accéder à ses sous-termes, et correspond à exposer la structure de l'arbre ou du composite qui l'implante.

Cet environnement définit aussi une méthode d'implantation de visiteurs sûrs pour les types lorsque des implémentations typées d'arbres sont utilisées, permettant la définition de visiteurs préservant le type des nœuds visités.

Limites

L'approche que nous avons suivie pour la conception et le développement du langage de stratégies présenté dans ce chapitre est proche de celles proposées par Stratego pour ce qui est de la définition des stratégies, ainsi que de celle de JJTraveler pour la conception objet.

Notre objectif est multiple :

- avoir la même expressivité que Stratego, ainsi que la possibilité de définir facilement des stratégies ;
- permettre de gérer le non-déterminisme comme dans le langage ELAN, tout en préservant une interface typée pour les termes ;
- intégrer le tout dans le langage Java, de manière élégante ;
- considérer des traits réflexifs, pour permettre la transformation de stratégies à haut niveau.

L'approche de Stratego est séduisante, car elle offre une syntaxe et une sémantique claires, centrées sur la programmation par stratégies. Cependant, elle présente un langage à part entière, ce qui rend l'intégration dans un projet existant difficile. Le langage, bien que simple pour ce qui concerne les stratégies, est toutefois difficile à maîtriser dans sa globalité. Finalement, le langage et les stratégies sont non typés.

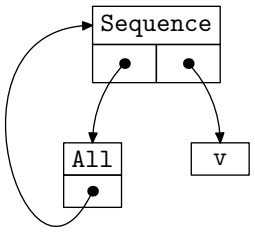
La bibliothèque de stratégies composables JJTraveler, ainsi que les patrons de conception attachés fournissent une base pour la programmation stratégique dans les langages orientés objet. Bien que basées sur les combinateurs de stratégies de Stratego, les stratégies basiques de JJTraveler peuvent être difficiles à composer. En particulier, les dé-

finitions récursives des stratégies doivent être implantées par un graphe au niveau des objets visiteurs. Ce graphe est obtenu par des astuces de programmation qui rendent ces définitions peu sûres. Ainsi, la librairie définit des combinateurs complexes comme autant de briques de base.

Exemple 27. L'opérateur de stratégie correspondant à la stratégie *BottomUp* est défini comme `BottomUp(v) = Sequence(All(BottomUp(v)),v)`, et donc revient à appliquer tout d'abord la stratégie `v` sous la stratégie *BottomUp* à tous les sous-termes du terme courant avant de l'appliquer en tête. Il est défini dans la librairie JJTraveler par la classe suivante, qui instancie le graphe de stratégie :

```
package jjtraveler;

public class BottomUp extends Sequence {
    public BottomUp(Visitor v) {
      super(null,v);
      first = new All(this);
    }
}
```



La définition dans la librairie est équivalente à la définition formelle, mais doit utiliser les mécanismes d'héritage ainsi que des pointeurs `null` initialisés après coup pour définir le graphe.

La bibliothèque fournit ainsi un grand nombre de briques de base (`All`, `One`, `Sequence`, `TopDown`, `DoWhileSuccess`, ...), qui peuvent être exprimées par une combinaison des combinateurs principaux, car il est difficile de définir de nouvelles stratégies lors de l'utilisation.

C'est en s'appuyant sur les patrons à la base de cette librairie que nous définissons notre langage de stratégie et son implantation. Les principales extensions de notre librairie (que l'on appellera MuTraveler) sont :

- la compatibilité avec les visiteurs de JJTraveler, c'est-à-dire qu'une structure de données conçue pour être utilisée avec l'une est utilisable avec l'autre ;
- un opérateur de récursion explicite `Mu` permettant de simplifier la définition de stratégies récursives ;
- la possibilité de manipuler les stratégies comme des objets algébriques, filtrer et transformer les stratégies ;
- une construction de déclaration de stratégies « utilisateurs » par filtrage ;
- la gestion des positions de traversée de termes, et la possibilité de les utiliser pour gérer le non-déterminisme ;
- des opérateurs de congruence et de construction permettant de spécifier des motifs et des règles réflexives.

6.2. Rendre l'opérateur de récursion μ explicite

Notre première contribution est de rendre l'opérateur de récursion utilisé pour les définitions de stratégies entièrement explicite.

Les définitions des stratégies composées récursives de Stratego [VB98, VBT98], comme celles du langage ELAN [Bor98] et JJTraveler [Vis01] utilisent un opérateur de récursion explicite μ. Stratego et ELAN autorisent aussi des définitions de stratégies récursives comme illustré dans l'exemple 25. Il n'est pas possible d'utiliser de telles définitions dans un langage comme Java. Pour les rendre possibles et faciles à utiliser, il est nécessaire de rendre l'opérateur de récursion μ explicite au niveau du langage de stratégies. Une fois cet opérateur μ rendu explicite, il sera possible de décrire une librairie de stratégies composées tout aussi complète que JJTraveler en fournissant simplement les quelques briques de base, ainsi que les définitions formelles des stratégies composées.

Les combinateurs de stratégies élémentaires, aussi appelés briques de bases, sont les seuls à posséder une implantation concrète dans la librairie, en tant que classes définissant un comportement particulier dans leur méthode `visit`. Ils sont décrits dans la figure 6.1.

Combinateur	Sémantique
`Identity()`	Ne fait rien, et retourne le terme d'origine
`Fail()`	Échoue toujours
`All(`v`)`	Applique v à tous les fils directs en séquence
`Choice(`v_1`,`v_2`)`	Applique v_1, puis v_2 si v_1 échoue
`Sequence(`v_1`,`v_2`)`	Applique v_1, puis v_2. Échoue si l'un des deux échoue
`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
`One(`v`)`	Applique v à tous les fils dans l'ordre, jusqu'à une réussite
`IfThenElse(`c`,`v_1`,`v_2`)`	Applique c, si c réussit, applique v_1, sinon v_2

FIG. 6.1: Les combinateurs élémentaires

6.2.1. La μ-expansion

Afin d'implémenter les définitions récursives de stratégies, utilisant l'opérateur μ pour exprimer la récursion, on a besoin d'une notion de variable de stratégie. On introduit alors un combinateur `MuVar` d'arité 1, dont l'argument est un nom à utiliser dans une définition de stratégie. On se donne aussi une fonction `mu` qui permet de représenter l'opérateur de récursion.

Exemple 28. La définition de la stratégie *BottomUp* de l'exemple 27, qui se définit formellement par $BottomUp(v) = \mu x.Sequence(All(x), v)$ devient une fonction qui prenant en argument une stratégie v retourne la stratégie $BottomUp(v)$:

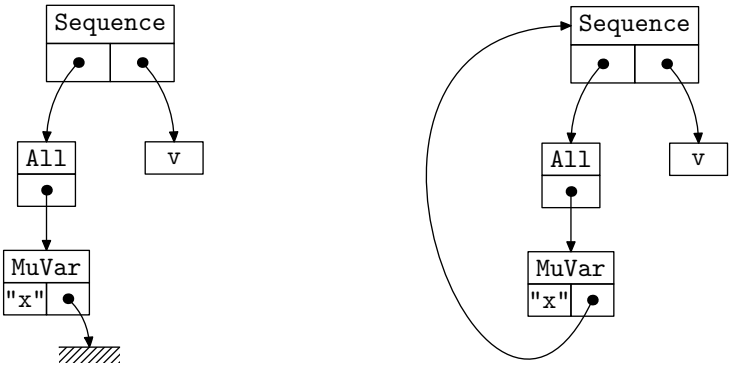
```
Strategy BottomUp(Strategy v) {
```

```
  return `mu(MuVar("x"), Sequence(All(MuVar("x")),v));
}
```

On suppose ici que chaque stratégie élémentaire est accessible par le biais d'une fonction de construction. Le « ` » est un service de Tom qui permet simplement d'appeler ces fonctions. On peut alors utiliser dans un programme Java une écriture qui est très proche de l'écriture formelle.

Cette stratégie peut alors être utilisée dans un programme pour visiter un terme `u`, appliquant une stratégie `s` sur tous les nœuds : `t = `BottomUp(s).visit(u);`.

Le combinateur `MuVar` est particulier, car il permet d'encoder l'appel récursif. Cet opérateur est d'arité 1 du point de vue de l'utilisateur, mais possède un pointeur vers l'objet de stratégie que cette variable représente, dans notre exemple vers la stratégie `Sequence(All(MuVar("x")),v)` résultat. Le graphe obtenu comme résultat de l'appel de la fonction `BottomUp` est alors celui de la figure 6.2b, qui provient de la μ-expansion de la représentation sous forme d'arbre de la figure 6.2a.



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

(b) Après μ-expansion : Graphe représentant *BottomUp*

FIG. 6.2: Graphes de stratégies avec le combinateur `MuVar`

(a) Nœud `MuVar` en mémoire

(b) Représentation compacte

FIG. 6.3: Convention de représentation des variables

Par la suite, nous utiliserons la notation présentée dans la partie 6.3b de la figure 6.3 pour représenter les nœuds `MuVar`. Le pointeur associé au nœud `MuVar` est initialement un pointeur `null`. La fonction `mu`, qui prend comme arguments une variable — un nœud `MuVar` — et un arbre représentant une stratégie, lequel contient des nœuds `MuVar`, parcours cet arbre à la recherche de variables de même nom que son premier argument, et

dont le pointeur n'est pas encore initialisé. Elle fait alors pointer ces nœuds vers la tête de l'arbre de stratégie, construisant le graphe.

6.2.2. Stratégies visitables

Cette fonction `mu` implémente une traversée d'arbre, à la recherche de nœuds particuliers (`MuVar`), afin de leur appliquer une transformation locale, mettant à jour les pointeurs cachés de ces nœuds. Il est alors tout naturel de vouloir l'implémenter en utilisant une stratégie de transformation. Les objets représentant les stratégies ne doivent alors pas être simplement des visiteurs, mais aussi des objets visitables au même titre que les termes, dont les sous-termes sont les stratégies qu'ils composent.

On peut alors envisager d'utiliser un parcours de l'arbre représentant la stratégie à l'aide d'une stratégie *BottomUp*. Le premier problème est de s'assurer que cette traversée ne prenne pas en compte les définitions récursives qui sont déjà en place, par exemple dans la stratégie `Repeat(BottomUp(. . .))`, car cela conduirait à des boucles infinies. Les nœuds `MuVar` étant explicites, il suffit de faire en sorte que le champ de l'objet `MuVar` contenant le pointeur ne fasse pas partie de l'interface `Visitable`. Ainsi, pour la stratégie de parcours, les nœuds `MuVar` n'ont qu'un seul sous-terme, qui est le nom de la variable.

L'autre problème que pose la traversée de cet arbre de stratégies est qu'il est nécessaire de construire une stratégie *BottomUp* pour décrire la traversée, stratégie qui devrait elle-même utiliser le mécanisme de μ-expansion implanté par la fonction `mu`. Pour assurer l'amorçage de la librairie, le processus de μ-expansion doit alors utiliser une stratégie *BottomUp* particulière, dans laquelle le graphe de stratégie est déjà explicite. On utilisera pour définir la fonction `mu` une stratégie `BottomUp` privée utilisant un encodage similaire à celui qui est présenté dans l'exemple 27. Cette stratégie est strictement privée à la définition de l'opération de μ-expansion, et n'est pas accessible à l'utilisateur de la librairie. La stratégie alors appliquée sous cette stratégie sera paramétrée par les deux arguments de la fonction `mu` : la variable et l'expression de stratégie. Cette stratégie fait pointer sur chacun des nœuds de l'expression de stratégie vers l'expression entière lorsqu'elle est un nœud `MuVar` de même nom que l'argument et que son pointeur est `null`.

Un des avantages de cette méthode est que l'implantation de la μ-expansion est simple, car il n'est pas nécessaire de traiter des problèmes d'alpha-conversion. En effet, les expressions de stratégies sont closes, et l'utilisation d'une fonction `mu Java` pour la μ-expansion implique que l'expansion se fait au niveau du lieur le plus profond d'abord, ce qui correspond à une stratégie d'appel par nom. Les variables à l'intérieur d'une expression sont donc correctement liées au niveau du μ le plus proche, une variable pour laquelle on a déjà donné une valeur au pointeur associé n'étant plus modifiée.

6.3. Filtrage et transformation de stratégies

Nous avons rendu l'opération de création du graphe de stratégie explicite par l'introduction d'une opération de μ-expansion. On peut alors représenter tous les opérateurs

additionnels de la librairie JJTraveler non plus par des classes de la librairie, mais simplement par des appels à cette μ-expansion, comme le montre l'exemple 28. Cette extension est aussi plus sûre, car le nombre de classes implantant le comportement des combinateurs est réduit au strict nécessaire. Un extrait des définitions de stratégies d'utilité courante est donnée dans la figure 6.4.

Stratégie	Sémantique
`Try(v)`	`Choice(v,Identity())`
`TopDown(v)`	`mu(MuVar("x"),Sequence(v,All(MuVar("x"))))`
`BottomUp(v)`	`mu(MuVar("x"),Sequence(All(MuVar("x")),v))`
`OnceBottomUp(v)`	`mu(MuVar("x"),Choice(One(MuVar("x")),v))`
`OnceTopDown(v)`	`mu(MuVar("x"),Choice(v,One(MuVar("x"))))`
`Repeat(v)`	`mu(MuVar("x"),Try(Sequence(v,MuVar("x")))`
	`mu(MuVar("x"),Sequence(All(MuVar("x")),`
	`Try(Sequence(v,MuVar("x")))))`

FIG. 6.4: Extrait de la librairie de stratégies

6.3.1. Les stratégies sont des termes : un mapping pour Tom

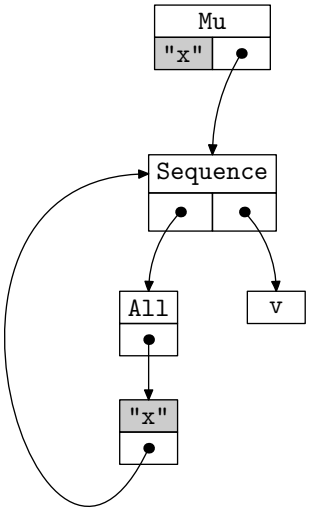
Toutes les stratégies sont construites à partir des combinateurs basiques, de l'opérateur de récursion et du combinateur `MuVar`. Ainsi, ces constructions de stratégies peuvent être manipulées comme des arbres, mis à part les pointeurs que contiennent les nœuds `MuVar`, qui sont cachés à l'utilisateur. On peut alors envisager d'écrire des règles de réécriture permettant de transformer de telles constructions, afin de les analyser, les optimiser ou encore les compiler.

Cependant, il est nécessaire de pouvoir aussi utiliser le filtrage sur les μ-expressions avant expansion afin de permettre la réalisation de transformations complexes de stratégies.

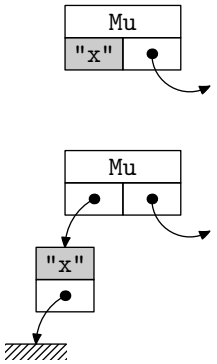
6.3.2. Opérateur μ encore plus explicite

On doit faire de μ un constructeur du langage de stratégies, pour pouvoir l'utiliser lors du filtrage, c'est-à-dire le rendre lui aussi explicite, comme l'on a rendu explicite la définition de variables de stratégies. La structure objet d'une stratégie composée comme la stratégie *BottomUp* des exemples précédents sera alors équivalente à l'AST qui correspond à sa déclaration. Le nœud `Mu` possède deux sous-termes, qui sont la variable qui est liée par l'opérateur μ et la stratégie dans laquelle cette variable est liée, comme représenté dans la figure 6.5, en utilisant la convention d'écriture de la figure 6.3.

Le premier argument d'un μ, la variable, est représentée, comme on le voit dans la figure 6.5b par un nœud `MuVar` comportant un pointeur initialisé à `null`. Une fois cet objet `Mu` introduit pour représenter explicitement les opérateurs μ de la description des



(a) Graphe représentant *BottomUp*



(b) Représentation d'un nœud `Mu`

FIG. 6.5: La structure objet de la représentation explicite de *BottomUp*

stratégies composées récursives, on doit adapter la procédure de μ-expansion à cette nouvelle représentation.

La sémantique des combinateurs `Mu` et `MuVar` est simple, et permet aux graphes représentant les stratégies avec μ explicite d'être interprétés de la même manière que le graphe original :

- `MuVar("x")` appliqué à un terme t retourne le résultat de l'application de la stratégie pointée par cette variable à t ou échoue si celle-ci échoue. Si le pointeur du nœud `MuVar` est `null`, alors une exception est levée. Ce cas est exceptionnel, car les stratégies pour être évaluées doivent être des termes clos.
- `Mu(MuVar("x"),s)` appliqué à un terme t est utilisé pour déclencher la μ-expansion, supprimant le besoin d'une fonction statique `init`. Si la μ-expansion n'a pas encore été faite sur `s`, alors elle est effectuée, puis le résultat de l'application de `s` à t est retourné, ou un échec si cette application échoue.

C'est l'évaluation d'une stratégie contenant un nœud `Mu` qui maintenant déclenche la μ-expansion. On utilise le pointeur du nœud `MuVar` qui est le premier argument du `Mu` afin de marquer les stratégies pour lesquelles la μ-expansion a déjà été faite : initialisé à `null`, celui-ci pointera vers le nœud `Mu` correspondant après l'opération de μ-expansion.

La μ-expansion n'est donc réalisée qu'une fois qu'elle est réellement nécessaire à l'évaluation d'une stratégie. N'étant plus déclenchée par l'évaluation d'une fonction, il faut alors prendre garde aux problèmes d'alpha-conversion, afin de résoudre correctement les liens. Ainsi, la procédure de μ-expansion est plus complexe que celle qui est présentée en section 6.2.2, car il faut aussi traiter les nœuds `Mu` qui peuvent être sous-termes du nœud pour lequel la procédure est déclenchée, et qui n'ont pas encore été expansés. L'expansion se fait alors en parcourant l'expression de stratégie depuis la racine jusqu'aux feuilles en conservant les déclarations de noms de variables et nœuds associés dans une

pile, pour lier les occurrences de ces variables au nœud associé à la variable de même nom la plus récemment entrée dans la pile.

```
Pile pile;
Fonction expanse(noeud) {
  si noeud est un Mu {
    empile noeud sur pile;
    expanse(variable de noeud)
    expanse(fils de noeud)
    dépile pile;
  } sinon si noeud est un MuVar(nom) {
    si pointeur de noeud ≠ null {
      s ←cherche(nom, pile);
      pointeur de noeud ←s;
    }
  } sinon {
    pour chaque fils de noeud {
      expanse(fils);
    }
  }
}
```

FIG. 6.6: Algorithme de μ-expansion (pseudo-code)

Cet algorithme, décrit dans la figure 6.6, utilise une pile pour mémoriser les couples variable, expression de stratégie dont les déclarations sont traversées. La fonction `cherche` parcourt la pile de la dernière entrée à la première et renvoie la valeur de l'expression de stratégie associée à la première occurrence de la variable trouvée.

6.3.3. Conséquences

Le fait de rendre l'opérateur de récursion totalement explicite dans une expression de stratégie permet d'une part de pouvoir construire de telles expressions de stratégie par une simple traduction de la description formelle de la stratégie et d'autre part de manipuler les expressions de stratégie comme des termes. Il est possible non seulement d'utiliser du filtrage pour examiner une stratégie, mais aussi de traverser et transformer une stratégie en utilisant une autre stratégie. La procédure de μ-expansion est un exemple de telle transformation. Cependant, parce que la μ-expansion est l'opération qui rend une stratégie exécutable en résolvant les liens des variables, celle-ci doit utiliser une stratégie particulière, codée directement en Java.

La possibilité de traverser, filtrer et construire des expressions de stratégies comme des termes algébriques permet alors de spécifier des règles de transformation de stratégies. On peut par exemple spécifier par une traversée de l'arbre de stratégie et filtrage une

méthode d'affichage du graphe de stratégie, ce qui peut être utile pour expliquer ou documenter des stratégies complexes. La figure 6.7 présente le graphe d'une stratégie utilisée dans l'optimiseur du compilateur Tom. Cet affichage est obtenu par la traversée du graphe par une stratégie générant une représentation textuelle.

On peut aussi utiliser ces trait réflexifs des stratégies pour définir des stratégies modifiant leur comportement pendant le parcours, ou encore des fonctions transformant ou optimisant les expressions de stratégies.

6.4. Rendre les stratégies utilisables

Nous avons jusque-là développé un langage de stratégies permettant de représenter des stratégies de traversée d'arbre génériques. Cette représentation, utilisant des stratégies élémentaires, dont des opérateurs `All`, `One` et `Omega` permettant d'accéder de manière générique aux sous-termes et 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.

6.4.1. Comment définir des stratégies « utilisateur »

Le langage de stratégies permet de spécifier la manière d'appliquer des transformations dans un arbre. Ces transformations sont décrites par des stratégies spécifiées par l'utilisateur du système, à la manière des règles nommées du langage Elan, décrites en section 6.1.2.

Dans l'expression de stratégie présentée en figure 6.7, les nœuds grisés correspondent aux stratégies utilisateur qui décrivent les différentes optimisations effectuées par le compilateur, tandis que les autres nœuds sont les combinateurs qui décrivent la procédure d'optimisation.

Note A. La stratégie présentée en figure 6.7 est représentée avec toutes les stratégies composées affichées sous la forme de leur graphe de stratégies élémentaires. Cette stratégie dont le graphe est complexe est programmée en combinant des stratégies prédéfinies. Ainsi, la première branche de la séquence principale est décrite dans le programme par :

```
InnermostId(ChoiceId(RepeatId((NopElimAndFlatten())),NormExpr()))
```

Dans laquelle `NopElimAndFlatten()` et `NormExpr()` construisent les stratégies « utilisateur » qui respectivement éliminent les instructions inutiles, et normalisent les expressions booléennes. On voit ici que la règle `NopElimAndFlatten()` est réutilisée à trois reprises, dans des contextes différents. En effet, dans certains cas elle est utilisée sous une stratégie `TopDown` pour normaliser, mais elle est aussi utilisée sous une stratégie `OnceTopDown` lorsque l'on sait dans le processus d'optimisation que la règle ne doit être appliquée qu'une seule fois, après fusion des blocks et des tests qui peuvent l'être.

La définition de telles transformations se fait dans le cadre de la librairie JJTraveler par l'écriture d'une classe Java qui étend une classe spécifique à la structure de données

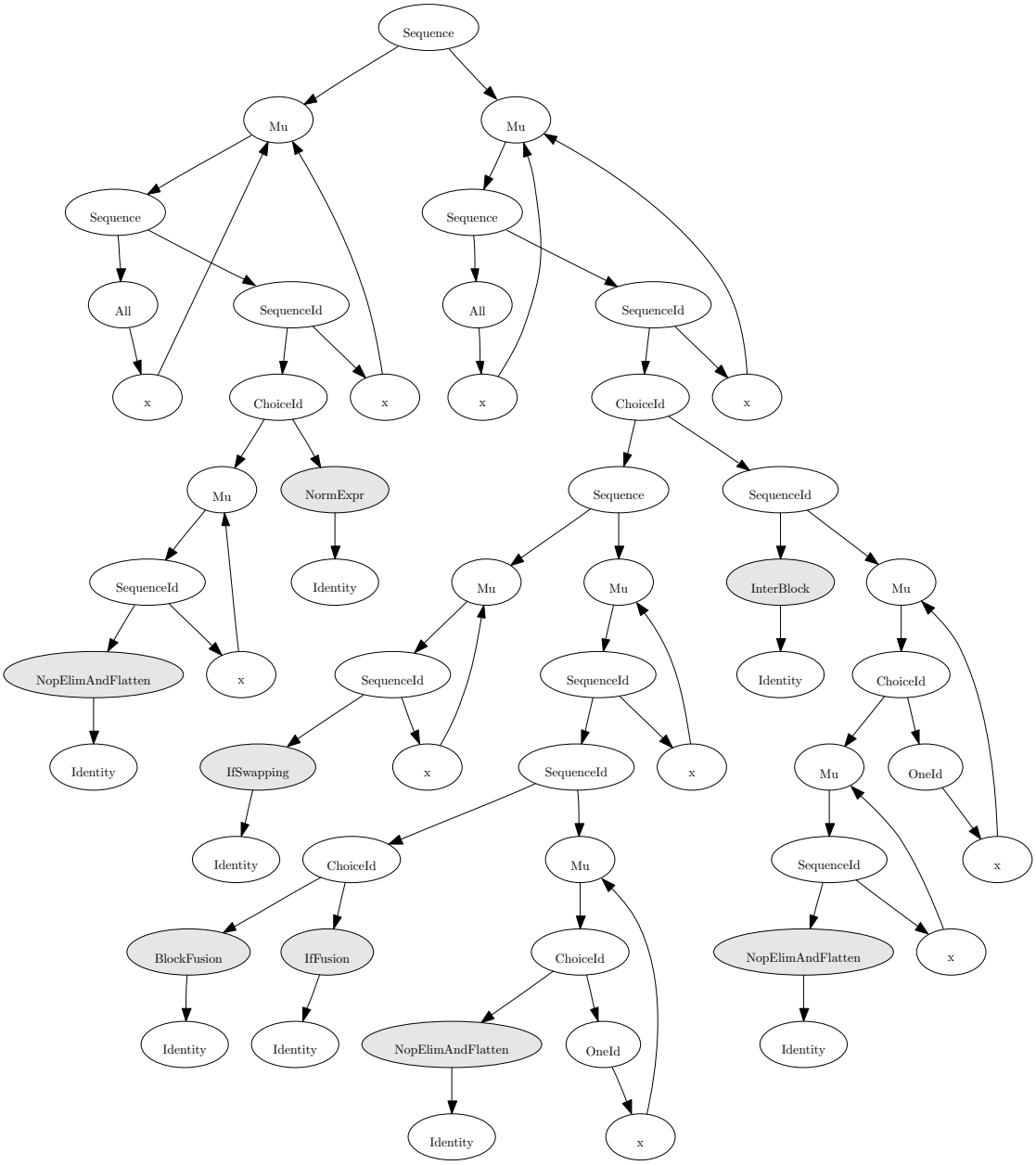


FIG. 6.7: Graphe de la stratégie d'optimisation du compilateur Tom

traversée. Nous appellerons par la suite cette classe dépendant de la structure de données `Fwd`. Celle-ci prend en argument une stratégie, généralement `Identity()` ou `Fail()`, et définit des méthodes de visite préservant les types de la structure de données, dont le comportement par défaut est de déléguer l'appel à la stratégie argument. L'utilisateur peut alors redéfinir une de ces méthodes spécifiques à un type dans la structure de données pour obtenir un comportement particulier. On peut voir en figure 6.7 que les stratégies utilisées pour la définition de l'optimiseur du compilateur Tom étendent le comportement de la stratégie identité. Ainsi, lorsque la règle ne peut pas être appliquée, la stratégie effectue l'identité, et ne change pas le terme courant, ni n'échoue.

Une construction de déclaration de stratégies

Pour pouvoir programmer une telle stratégie utilisateur, il est nécessaire de connaître les détails de fonctionnement interne de la bibliothèque, comme illustré en figure 6.8. La structure de données utilisée avec les stratégies est généralement une structure générée automatiquement pas un outil comme Gom. Ainsi, il n'est pas nécessaire que l'utilisateur définisse lui-même la classe `Fwd` constituant la base des stratégies utilisateur. Cependant, il reste nécessaire de connaître les patrons de conception dirigeant la librairie de stratégie pour développer les stratégies.

Pour rendre le développement de stratégies plus facile, nous définissons une extension du langage Tom par une construction **%strategy** permettant de définir une stratégie de manière abstraite. Cette construction, illustrée en figure 6.8 par l'encodage d'une règle de réécriture $f(x,x) \to g(x)$, pour laquelle on suppose f et g de sorte T, permet une écriture concise et abstraite de la règle. La définition qui suit `visit T` explicite le comportement de la stratégie lorsqu'elle visite un nœud de sorte `T`, en utilisant le filtrage sur les termes comme dans l'instruction **%match**. On peut définir une construction **visit** pour chaque sorte de l'algèbre de termes. Cette stratégie étend l'échec, ce qui signifie qu'elle échoue lorsque la règle définie ne peut pas s'appliquer, et donc n'échoue pas uniquement sur les radicaux qui filtrent $f(x,x)$.

Cette construction **%strategy** permet aussi de décrire des stratégies paramétrées, les paramètres étant passés comme arguments lors de la construction de la règle. La stratégie une fois déclarée peut être utilisée à la manière des combinateurs élémentaires dans l'écriture d'une expression de stratégie.

La stratégie définie à l'aide de la construction **%strategy** a le même statut que les combinateurs pour la librairie de stratégie. Elle est donc elle-même visitable et il est possible d'utiliser une construction de filtrage sur cette stratégie. La procédure de μ-expansion traite ainsi les sous-termes d'une stratégie utilisateur qui sont de type `Strategy`, ce qui permet d'écrire des versions spécialisées des briques de base, ainsi que des stratégies dont les arguments représentent la suite du calcul pouvant être récursives.

Cela est utile pour décrire une stratégie réutilisable effectuant un appel récursif explicite non pas sur la brique de base elle-même, mais sur la stratégie de parcours elle-même.

Stratégie $f(x,x) \rightarrow g(x)$ utilisant JJTraveler et Tom pour le filtrage :

```
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); }
    }
  }
}
```

Stratégie $f(x,x) \rightarrow g(x)$ utilisant **%strategy** :

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

FIG. 6.8: Exemple de stratégie codant une règle de réécriture

6.4.2. Des stratégies basées sur l'identité comme échec

Dans la librairie MuTraveler, comme dans JJTraveler, l'échec d'une stratégie est représenté par la levée d'une exception de type `jjtraveler.VisitFailure`. Les exceptions étant un moyen de détourner le flot de contrôle normal d'un programme, cet encodage est adapté à la plupart des situations, car il permet de représenter le *backtracking* local pour les combinateurs comme le choix.

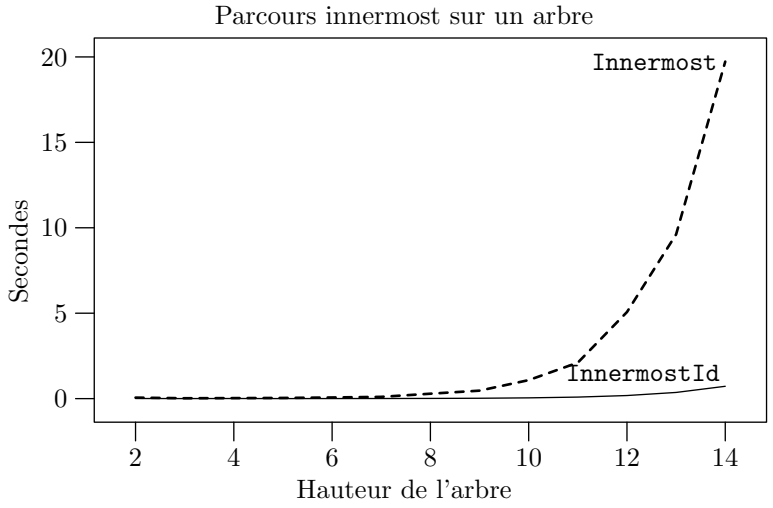
Cette approche, bien que souple, est en général inefficace, car la pose d'un nouveau point de choix se fait par l'intermédiaire d'une construction `try ... catch` qui place un point de choix sur la pile d'exécution. Cette construction, dans le modèle d'exécution de Java, n'est pas très coûteuse lorsqu'aucune exception n'est levée. Cependant, le coût d'exécution augmente lorsque l'exception est effectivement levée.

Des problèmes d'efficacité se font alors sentir lorsque la stratégie utilisateur échoue souvent, car alors de nombreuses exceptions seront levées et rattrapées au cours du parcours. Ceci est pourtant un cas commun lorsque l'on considère des règles de transformation. La stratégie décrite en figure 6.8 par exemple échouera sur tout nœud qui n'est pas de la forme $f(x,x)$. Si cette stratégie est appliquée sous la stratégie `Innermost`, alors au moins autant d'exceptions qu'il y a de nœuds dans l'arbre seront levées, conduisant à une stratégie inefficace.

Afin de pallier ce problème, nous avons introduit la notion de stratégies basées sur l'identité. Beaucoup d'applications ne nécessitent pas une gestion très fine des échecs,

Combinateur	Sémantique
`ChoiceId(`v_1`,`v_2`)`	Applique v_1, puis v_2 si v_1 retourne l'identité
`SequenceId(`v_1`,`v_2`)`	Applique v_1, puis v_2. Échoue si l'une des deux applications retourne l'identité
`OneId(`v`)`	Applique v à tous les fils dans l'ordre, tant que v retourne l'identité
`AllId(`v`)`	Applique v à tous les fils dans l'ordre, retourne l'identité si l'un des sous-termes n'est pas changé
`TryId(v)`	`v`
`RepeatId(v)`	`mu(MuVar("x"),SequenceId(v,MuVar("x")))`
`BottomUpId(v)`	`mu(MuVar("x"),SequenceId(All(MuVar("x")),v))`
`TopDownId(v)`	`mu(MuVar("x"),SequenceId(v,All(MuVar("x"))))`
`OnceBottomUpId(v)`	`mu(MuVar("x"),ChoiceId(OneId(MuVar("x")),v))`
`InnermostId(v)`	`mu(MuVar("x"),Sequence(All(MuVar("x")),` `SequenceId(v,MuVar("x"))))`

(a) Combinateurs élémentaires et stratégies communes basées sur l'identité



(b) Application d'un système de réécriture sous stratégie basée sur l'identité

FIG. 6.9: Stratégies basées sur l'identité

car elles correspondent à calculer un point fixe à l'application d'une stratégie. On peut dans ce cas considérer que renvoyer l'identité, c'est-à-dire ne pas modifier le terme, peut représenter un échec de la stratégie. On décide donc d'encoder systématiquement l'échec de l'application d'une stratégie par l'identité, en introduisant des opérateurs de composition dits « basés sur l'identité ». Les nouveaux opérateur ainsi introduits sont présentés en figure 6.9a. La stratégie de l'optimiseur de Tom de la figure 6.8 en utilise certaines pour encoder l'application jusqu'à un point fixe de stratégies comme l'élimination des instructions inutiles. Ces combinateurs basiques basés sur l'identité utilisent alors un simple test d'égalité (qui est effectué en temps constant dans le cas des structures de données générées par Gom grâce au partage maximal) pour tester l'échec. Les stratégies employées sous ces combinateurs doivent alors étendre l'identité, ou être gardés par un `Try`.

Le gain en efficacité est très important lorsqu'il s'agit d'appliquer un système de réécriture de manière répétée pour obtenir un point fixe. Le graphe de la figure 6.9b présente le temps d'exécution de la normalisation sous stratégie *leftmost-innermost* du système

$$\begin{array}{lcl} a & \rightarrow & b \\ b & \rightarrow & c \\ g(c) & \rightarrow & c \end{array}$$

appliqué à des termes constitués d'un certain nombre d'étages de f binaires, et dont les feuilles sont des $g(a)$. Ce système est encodé comme étendant l'échec, et appliqué sous la stratégie `Innermost`, puis encodé comme étendant l'identité, et appliqué sous la stratégie `InnermostId`. Pour de petites hauteurs de l'arbre, la différence de temps d'exécution n'est pas importante. Lorsque la hauteur de l'arbre augmente, le nombre de nœuds augmente de façon exponentielle, exposant alors le coût de la gestion des exceptions par rapport à celui des tests d'égalité qui les remplacent dans le cas des stratégies basées sur l'identité.

Ce gain d'efficacité se fait au détriment de l'expressivité. En effet, si l'on ne peut plus distinguer un échec de l'identité, certaines applications deviennent plus difficiles à programmer. Par exemple, pour encoder de manière efficace une stratégie recherchant un sous-terme particulier, on peut utiliser l'échec comme une exception pour stopper le flot normal d'exécution lorsque ce nœud particulier est trouvé, évitant ainsi de parcourir tous les nœuds restants.

Cette distinction entre l'échec de l'application d'une stratégie et le fait de considérer l'identité comme une condition d'arrêt est aussi présent dans le langage MGS [GM01]. Ce langage de simulation de processus biologiques par règles de transformation de collections topologiques introduit deux types de point fixe pour l'application de ces transformations :

- « `T[fixrule](c)` » applique une transformation `T` jusqu'à ce qu'aucune règle ne s'applique ;
- « `T[fixpoint](c)` » applique une transformation `T` jusqu'à atteindre un point fixe, c'est-à-dire que l'application de la transformation laisse la collection `c` inchangée.

L'ajout de ces combinateurs de stratégie « basés dur l'identité » permet d'exprimer de manière plus simple et efficace les stratégies de normalisation communes en réécriture,

comme la normalisation *innermost* ou l'obtention de point fixe.

6.4.3. Savoir où l'on en est : les positions

Parmi les objectif que nous nous étions fixés en section 6.1.1, le problème de la gestion du non-déterminisme n'a pas encore été traité. Nous avons vu en section 6.1.2 que le langage de stratégies de Stratego, qui a fortement inspiré notre langage, n'offre pas de mécanisme de gestion du non-déterminisme.

Une première solution applicable en Tom est d'utiliser une structure d'ensembles, comme les `Collections` de Java pour manipuler des ensembles de résultats et d'expliciter, en utilisant les opérations sur ces ensembles, la manière de gérer les résultats non-déterministes. Cette solution est suffisante pour traiter les problèmes d'atteignabilité dans un système de réécriture, comme dans l'exemple de la vérification du protocole de Needham-Schroeder présenté en section 4.3.2, lorsque les règles de réécriture s'appliquent toujours en tête d'un terme.

Nous avons ajouté dans notre système de stratégies la notion de position, ainsi que des moyens de gérer et utiliser les positions pour la programmation de stratégies complexes, ainsi que permettant de gérer le non-déterminisme, ainsi que de manipuler les positions.

La stratégie Oméga

Les positions et les stratégies associées sont principalement basées sur la stratégie `Omega` décrite en figure 6.1. Cette stratégie n'est pas nouvelle, et était déjà présente dans la description des combinateurs de stratégie de Stratego [VBT98] comme décrit en section 6.1.2. Elle permet d'appliquer son argument à un sous-terme donné du sujet. Alors, une composition de `Omega` imbriqués peut être utilisé pour encoder l'accès à une position donnée dans un arbre.

La nouveauté ici est de donner accès à la position courante dans le terme, par l'intermédiaire d'une méthode `getPosition()` accessible depuis les actions de chaque construction **%strategy**. Cette méthode renvoie un objet représentant la position courante du parcours. Ainsi, une stratégie lorsqu'elle traverse un terme, connaît en permanence le chemin qu'il faut parcourir depuis la racine du terme sur lequel elles est appliquée pour atteindre le nœud qu'elle visite. L'objet représentant la position peut alors être stocké pour être ensuite réutilisé, ou utilisé pour obtenir l'une des stratégies associées à cette position. Cet objet n'est pas lié à la stratégie, et ne sera pas modifié lorsque la stratégie continuera le parcours. On peut donc utiliser ces positions pour mémoriser les radicaux sur lesquels les règles peuvent être appliquées, et utiliser les stratégies associées pour appliquer ces règles et sélectionner les positions à utiliser.

Ces stratégies associées à une position `p` sont :

– `p.getOmega(s)`, qui retourne une stratégie appliquant la stratégie `s` à la position représentée par `p`. Si `p` correspond à une position ω, l'application de `p.getOmega(s)` à un terme t retourne t' une copie de t dans lequel le sous-terme en position ω est remplacé par l'application de `s` à ce sous-terme, $t' = t[\omega \hookleftarrow \mathtt{s}(t_{|\omega})]$. Pour une position

$\omega = \{1, 2\}$, qui correspond à la position de c dans le terme $f(a, f(b, c))$, on obtient la stratégie `Omega(1,Omega(2,v))` ;
- `p.getReplace(r)`, qui retourne une stratégie appliquant la règle $x \to$ `r` à la position ω représentée par `p`. Appliquée à un terme t, cette stratégie retourne alors $t[\omega \hookleftarrow$ `r`$]$. Comme la précédente, elle échoue si la position n'existe pas dans le terme ;
- `p.getSubterm()`, qui retourne une stratégie dont l'application a pour résultat, lorsqu'appliquée sur un terme t le sous-terme de t à la position ω représentée par `p`, $t_{|\omega}$;
- `p.getOmegaPath(s)`, qui retourne une stratégie appliquant la stratégie `s` à tous les radicaux sur le chemin de la position représentée par `p` jusqu'à la racine.

Ces stratégies échouent toutes lorsque la position voulue n'existe pas dans le terme auquel elle est appliquée.

Maintien d'une référence à la position

Afin de permettre de récupérer la position courante dans le terme traversé, il est nécessaire que toute stratégie ait accès à un objet représentant cette position. De plus, les combinateurs permettant de traverser un terme doivent modifier la valeur de cet objet position pendant la traversée.

Cela se fait en maintenant dans chaque combinateur élémentaire de stratégie (incluant ceux qui sont générés pour chaque construction **%strategy**) une référence vers un objet `Position`. De plus, les combinateurs `All` et `One`, ainsi que `Omega` doivent appeler des méthodes `down(i)` et `up` sur cet objet, lorsque la stratégie descend dans le sous-terme en position i et après cette visite.

Afin de pouvoir utiliser ces fonctionnalités, il est nécessaire de s'assurer que tous les combinateurs de la stratégie partagent le même objet `Position`. Une méthode statique `init` est ainsi fournie, qui utilise une stratégie *BottomUp* pour initialiser la référence vers un objet `Position` au niveau de tous les combinateurs, comme l'illustre la figure 6.10. De plus, la méthode `apply` permet d'appliquer une stratégie sur un sujet, en prenant garde à initialiser les positions si cela s'avère nécessaire. Cet objet `Position` maintient une pile des index des positions traversées, qui est mise à jour par les méthodes `down(i)` et `up()`, qui respectivement place i sur la pile et supprime la tête de la pile.

La méthode `getPosition()` retourne lors du parcours une copie de cette position partagée (ainsi que de la pile associée), afin de s'assurer que la position récupérée par l'utilisateur ne sera plus modifiée. C'est l'exploitation de cette pile qui permet alors de construire les différentes stratégies décrites en section 6.4.3.

Étant donné que la librairie est constituée d'un très petit nombre de combinateurs basiques, et que les extensions de la librairie consistent en des définitions de combinaisons courantes de ces combinateurs, le support des positions s'étend tout naturellement à toutes les stratégies de la librairie.

Utilité pratique

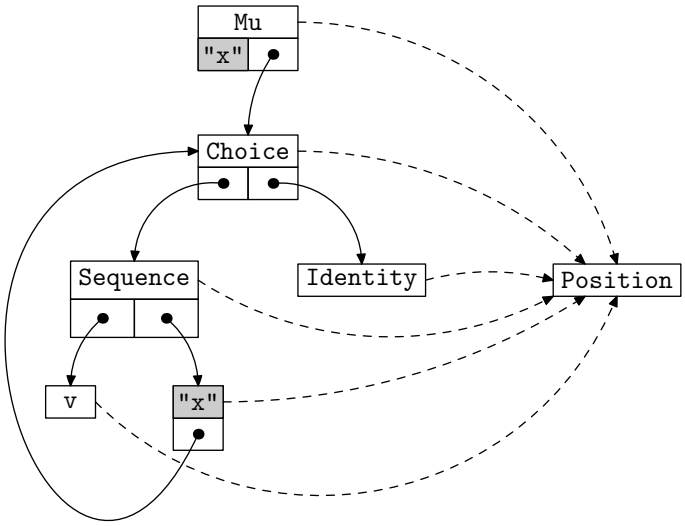


FIG. 6.10: Partage d'un objet représentant la position par tous les combinateurs d'une stratégie `Repeat(v)`

La gestion des positions lors des traversées d'arbre permet d'une part d'exprimer des stratégies complexes, pour lesquelles l'action à effectuer au niveau des différends nœuds ne dépend pas seulement d'information relative au nœud lui-même, mais aussi de sa position dans l'arbre.

D'autre part, cette notion de position permet de gérer le non-déterminisme de l'application des règles de manière fine. Dans des applications comme la recherche de preuves dans un système logique, il est nécessaire de pouvoir étudier différents chemins, correspondant à différentes manières de choisir les radicaux sur lesquels les règles sont appliquées, comme on l'a vu à propos du calcul des structures dans la section 4.5.1. On peut traiter un problème d'atteignabilité dans un système de réécriture en calculant ainsi les successeurs d'un terme donné par l'application d'une règle à une position quelconque en utilisant une stratégie. Cette stratégie, paramétrée par un ensemble (une `Collection` en `Java`), va, au lieu de modifier le terme lui-même lorsqu'un radical où appliquer une règle est trouvé, placer le terme résultant à l'application de cette règle dans l'ensemble.

Une telle stratégie n'est pas exprimable dans le langage de stratégies de Stratego ou JJTraveler, car l'information disponible lors de la visite est locale au radical. Le résultat de l'application d'une règle à une position donnée peut être calculé, mais pas l'ensemble de résultats généré par plusieurs applications de règles.

Exemple 29. Considérons le système de réécriture $\mathcal{R}$ défini par :

$$\mathcal{R} = \begin{cases} f(x) & \to & f(s(x)) \\ f(s(x)) & \to & f(f(x)) \\ s(s(x)) & \to & f(x). \end{cases}$$

La stratégie `Collect` prend en argument un ensemble, décrit par une `Collection`, et

place dans cet ensemble les successeurs du terme sur lequel elle est appliquée par l'application d'une règle de $\mathcal{R}$. Une référence vers le terme sur lequel elle est appliquée lui est aussi passé en argument, afin de pouvoir utiliser la stratégie de remplacement décrite en section 6.4.3 pour calculer le successeur. Celui-ci est alors ajouté dans l'ensemble des termes atteignables.

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

Cette stratégie `Collect` est alors appliquée de manière répétée sous une stratégie *BottomUp* jusqu'à l'obtention du terme à atteindre.

```
  public boolean reach(T start, T end) {
    // crée les ensembles de résultats, result conserve tous les termes
    // atteints, et c1 ceux de la dernière génération (n-1)
    Collection result = new HashSet();
    Collection c1 = new HashSet();
    // on commence avec l'ensemble de résultats {start}
    c1.add(start);
    // itère la recherche
    for(int i=1 ; i<MAXITER ; i++) {
      // c2 contiendra les termes de la génération n
      Collection c2 = new HashSet();
      Iterator it = c1.iterator();
      // itère sur les termes de la génération n-1
      while(it.hasNext()) {
        // applique la stratégie Collect, place les résultats dans c2
        ‘BottomUp(Collect((Term)it.next(),c2)).apply(c2);
      }
      // supprime de c2 les termes déjà rencontrés précédemment
      c2.removeAll(result);
      // la génération n devient le sujet, on calculera n+1
      c1 = c2;
```

```
    // marque les termes de la génération n comme déjà rencontrés
    result.addAll(c2);
    // on s'arrête si le but est trouvé dans la génération n
    if(result.contains(end)) {
      return true;
    }
  }
  return false;
}
```

L'utilisation de Java au sein de programmes Tom comportant des stratégies permet de se servir des fonctionnalités du langage pour gérer l'ensemble des termes atteignables. On peut alors, comme c'est le cas dans la fonction `reach` éliminer les termes déjà rencontrés lors d'une itération précédente pour éviter d'explorer plusieurs fois des branches de calcul similaires, ou encore implémenter une stratégie de recherche particulière, comme la *recherche globale* présentée en section 4.5.6.

La gestion des positions permet aussi de générer des traces fines de l'application d'un système de réécriture. Les environnements de réécriture comme Elan ou Maude permettent d'implanter des prouveurs ou de procédures de décision pour différentes théories en utilisant la réécriture, éventuellement sous stratégie. Cependant, les preuves alors produites ne peuvent pas être vérifiées ; le système répond « vrai » ou « faux », mais ne fournit en général pas de témoin ou de justificatif. Quang Huy Nguyen, dans sa thèse de doctorat [Ngu02], présente une méthode permettant d'utiliser les traces de réécriture (mémorisant les règles appliquées ainsi que les positions de ces applications) pour certifier des preuves obtenues par réécriture dans le système Coq. Il a dû étendre le compilateur Elan afin d'obtenir ces traces. Le support des positions dans le langage de stratégies de Tom permet de produire des traces en restant au niveau « utilisateur », sans avoir à modifier le compilateur, tout en ayant la possibilité d'utiliser des tactiques de recherche de preuve fines, comme la recherche globale.

6.5. Toujours plus d'expressivité

On a maintenant la possibilité de définir des stratégies de manière aisée, et de permettre la gestion du non-déterminisme au niveau des stratégies. Les stratégies définies par l'utilisateur utilisent le filtrage pour sélectionner les actions à exécuter sur les différents nœuds.

Bien que les expressions de stratégie soient réflexives, on ne peut qu'agir sur l'expression de stratégie spécifiant le parcours, et non sur les règles elles-mêmes, qui sont compilées par le compilateur Tom.

Nous présentons ici comment il est possible de spécifier filtrage et règles de réécriture au niveau même des stratégies, obtenant ainsi un environnement de spécification par règles en Java réflexif.

6.5.1. Opérateurs de congruence

Les opérateurs de stratégie de congruence, introduits pour le langage ELAN en section 6.1.2, permettent de spécifier des stratégies ayant un comportement particulier en fonction de la forme du terme auquel elles sont appliquées. Pour chaque opérateur f de la signature, un combinateur de stratégie `_f(s1,...,sn)` est fourni. Appliqué sur un terme de la forme $f(t_1, \ldots, t_n)$, celui-ci retourne le terme composé par les applications de chaque `si` au sous-terme t_i correspondant, c'est-à-dire $f(\texttt{s1}[t_1], \ldots, \texttt{sn}[t_n])$, en notant $\texttt{s}[t]$ l'application de la stratégie `s` au terme t. Si le terme n'est pas de la forme $f(t_1, \ldots, t_n)$, alors la stratégie échoue.

Note B. Nous utilisons la notation `_f` pour représenter la stratégie de congruence correspondant au symbole f de la signature, car les stratégies étant des termes au niveau du langage Tom, nous ne pouvons utiliser la même écriture pour le symbole de la signature et la stratégie de congruence, car Tom ne permet pas la *surcharge*.

De tels opérateurs de congruence ne peuvent pas être génériques, comme le sont les combinateurs `All` et `One`, mais dépendent de la signature de termes considérée. Lorsque l'implantation de la structure de termes est décrite dans le langage Gom, le compilateur génère non seulement l'implantation de cette structure, comme présenté dans le chapitre 4, mais aussi une implantation pour chaque opérateur de congruence associé.

Note C. Le cas des arguments de type *builtin* dans une signature est particulier. En effet, les stratégies ne peuvent pas être appliquées sur les types *builtin*, et les stratégies correspondant aux champs *builtin* dans l'opérateur de congruence ne sont jamais utilisées, mais remplacées par l'identité.

Les opérateurs de congruence étant directement disponible lorsque la structure de termes est générée par Gom, ils peuvent être utilisés pour exprimer des stratégies comme l'itération sur les éléments d'une liste, lorsqu'elle est représentée par exemple par la liste vide `Nil()` et l'insertion en tête `Cons(head:Element,tail:List)`.

Exemple 30. La stratégie `map(s)`, qui applique une stratégie `s` sur tous les éléments d'une liste peut être définie en utilisant les opérateurs de congruence par :

```
%gom {
  module List
  abstract syntax
  Element = | a() | b() | c()
  List = Nil()
       | Cons(head:Element,tail:List)
}

Strategy map(Strategy s) {
  return `mu(MuVar("x"),Choice(_Nil(),_Cons(s,MuVar("x"))));
}
```

Les opérateurs variadiques de Gom sont une manière de représenter des listes d'éléments. L'opérateur de congruence de ces opérateurs est particulier, car il ne prend qu'une seule stratégie en argument, et applique cette stratégie à tous les fils de l'opérateur. Ainsi, l'opérateur de congruence associé aux opérateurs variadiques fournit une stratégie équivalente au `map` pour les listes représentées par ces opérateurs variadiques.

Exemple 31. On utilisera les définitions de structure de terme et stratégie suivantes :

```
%gom {
  module List
  abstract syntax
  Element = a() | b() | c()
  List = List(Element*)
}

%strategy S() extends Identity() {
  visit Element {
    a() -> { return `b(); }
  }
}
```

L'exécution de l'instruction `` `_List(S()).apply(`List(a(),c(),a())) `` va retourner le terme `List(b(),c(),b())`, correspondant à l'application de la règle $a \to b$ à tous les éléments de la liste `List(a(),c(),a())` ;

Les stratégies de congruence permettent aussi d'exprimer une forme de filtrage lorsque plusieurs stratégies de congruence sont imbriquées.

Exemple 32. Si l'on considère la structure de termes définie par

```
Term = | a() | b() | c()
       | g(t:Term)
       | f(t1:Term,t2:Term)
```

La stratégie `_f(_g(_a(),b()))` retourne l'identité lorsqu'elle est appliquée au terme $f(g(a), b)$, et échoue dans tous les autres cas, implémentant donc le filtrage avec motif clos. On peut aussi appliquer sur l'un des sous-termes une stratégie particulière, afin de transformer certains sous-termes, ou encore mémoriser leurs valeurs.

6.5.2. Opérateurs de construction, et variables

Les stratégies de congruence fournissant le moyen d'exprimer du filtrage au niveau des stratégies, il reste à fournir des stratégies construisant des termes.

Ainsi, pour chaque opérateur f d'arité n de la signature, on fournit un combinateur de stratégie `Make_f(s1,...,sn)`. Ces opérateurs de construction sont particuliers, car le terme sur lequel ils sont appliqués n'est pas utilisé, et peut être `null`. La stratégie

`Make_f(s1,...sn)`, appliquée sur un terme quelconque, renvoie le terme $f(t_1, \ldots, t_n)$, avec pour tout i de 1 à n, t_i le résultat de l'application de `si` sur le terme `null`. Cette stratégie échoue si l'une de ces applications échoue.

Les stratégies qu'un opérateur de construction compose sont appliqués sur un terme `null`, donc pour que l'exécution d'une stratégie de construction n'échoue pas, les sous-stratégies qui la composent doivent être elles-mêmes des stratégies de construction, ou des stratégies n'utilisant pas leur argument.

Pour les opérateurs de construction, comme pour les opérateurs de congruence, le compilateur Gom génère une implantation pour chaque opérateur de la signature de l'opérateur de construction associé, ainsi que le *mapping* Tom permettant de les manipuler.

Exemple 33. En utilisant la définition de signature de l'exemple 32, l'application de la stratégie `Make_f(Make_g(Make_a()),Make_a())` sur un terme quelconque retourne le terme $f(g(a), a)$ comme résultat.

Afin d'implanter à l'aide des opérateurs de congruence et des opérateurs de construction des stratégies implémentant des règles de réécriture, il nous manque des opérateurs permettant de déclarer et instancier des variables, ainsi que de récupérer leur valeur.

De tels opérateurs peuvent être décrits en utilisant la construction **%strategy**, définissant des stratégies utilisant une table de hachage pour mémoriser les instanciations de chaque variable.

Exemple 34. Une implémentation des opérateurs de déclaration et dé-référencement de variables pour les termes de la sorte `Term` définie dans l'exemple 32 peut être donnée par les stratégies `Set` et `Get` suivantes :

```
%strategy Set(env:Map, name:String) extends Identity() {
  visit Term {
    x -> { env.put(name,‘x); }
  }
}
%strategy Get(env:Map, name:String) extends Identity() {
  visit Term {
    _ -> { return (Term) env.get(name); }
  }
}
```

On peut utiliser les stratégies codant des variables avec les opérateurs de congruence et de construction pour décrire des règles de réécriture au niveau des stratégies. Un tel encodage se fait en utilisant les stratégies de congruence et la stratégie `Set` pour définir un motif, avec ses variables instanciées par filtrage. Si cette stratégie de filtrage n'échoue pas, cela signifie que le terme d'origine filtre bien le motif, et il est alors possible de construire le membre droit de la règle en utilisant la stratégie `Get` et les opérateurs de construction. Le motif et le membre droit de la règle sont composés par une séquence,

ce qui fait que la construction du résultat n'est effectuée que lorsque le terme d'entrée filtre le motif, la stratégie encodant la règle de réécriture échouant si ce n'est pas le cas.

Exemple 35. On peut encoder la règle de réécriture $f(g(x), a) \rightarrow f(x, b)$ définie sur l'algèbre de termes de l'exemple 32 par la stratégie :

```
Map env = new HashMap();
Strategy rule = `Sequence(
                  _f(_g(Set(env,"x")),_a()),
                  Make_f(Get(env,"x"),Make_b())
                );
```

Les stratégies `Set` et `Get` partagent une référence vers la même table de hachage, permettant de propager l'information d'instanciation du membre gauche de la règle au membre droit.

Ces stratégies de gestion de variables sont ici très simples, mais il est possible de définir des stratégies de gestion de variables plus fines, permettant par exemple de tester l'égalité entre deux variables, ou de tester l'égalité du terme visité avec la valeur d'une variable, ce qui permet d'implanter du filtrage non-linéaire.

Les expressions de filtrage et réécriture pouvant être définies en utilisant les stratégies de congruence et de construction sont moins puissantes et expressives que celles qu'offre le compilateur Tom, ne permettant que le filtrage syntaxique, éventuellement non-linéaire si les stratégies gérant les variables sont assez fines. Cependant, celles si sont directement des stratégies qui peuvent être composées aisément, et manipulées comme les autres stratégies. On peut alors utiliser filtrage et stratégies pour étudier, traverser et transformer ces règles, offrant alors un environnement de spécification de règles entièrement réflexif intégré au langage Java.

Exemple 36. L'utilisation des traits réflexifs du langage de stratégie permet d'écrire une *méta-règle* $f(X, \_) \rightarrow f(Y, \_) \Rightarrow g(X) \rightarrow g(Y)$. Par exemple, celle ci transforme la règle $f(g(x), a) \rightarrow f(x, b)$ (représentée par une stratégie comme dans l'exemple 35) pour obtenir la règle $g(g(x)) \rightarrow g(x)$. Une telle *méta-règle* est une stratégie qui peut s'écrire :

```
%strategy Rewrite() extends Identity() {
  visit Strategy {
    Sequence(_f(X,_),Make_f(Y,_)) -> {
      return `Sequence(_g(X),Make_g(Y));
    }
  }
}
```

Appliquée sur la stratégie `rule` de l'exemple 35, cette stratégie retourne une stratégie `Sequence(_g(_g(Set(env,"x"))),Make_g(Get(env,"x")))`, les variables `X` et `Y` du motif de la stratégie `Rewrite` ayant été respectivement instanciés par `_g(Set(env,"x"))` et `Get(env,"x")`.

6.5.3. Compilation des stratégies

Les stratégies complexes sont représentées par un graphe d'objets, qui s'appellent les uns les autres pendant la traversée. Les stratégies pouvant maintenant être manipulées comme des termes algébriques, on peut envisager de transformer des expressions de stratégies en des stratégies plus efficaces. En particulier, la composition de stratégies complexes se fait en composant des stratégies elles-mêmes non élémentaires, ce qui peut conduire à des écritures non optimales des stratégies, mais que des règles de transformations simples peuvent améliorer.

Exemple 37. La stratégie `Choice(Try(s1),s2)` est équivalente à la stratégie `Try(s1)`, car `Try(s1)` ne pouvant pas échouer, `s2` ne sera jamais exécutée.

Cependant, dans le cas général, il est difficile de concevoir des règles d'optimisation des stratégies qui préservent la sémantique de ces stratégies, c'est-à-dire l'ordre de parcours.

Une autre voie est de transformer le graphe de stratégie en une seule stratégie de même sémantique. En effet, une fois le graphe de stratégie initialisé, les appels aux différentes méthodes `visit` au sein de la stratégie peuvent tous être résolus statiquement. On peut alors envisager d'effectuer une forme d'évaluation partielle du graphe de stratégie, en remplaçant les appels non récursifs à des méthodes `visit` de combinateurs de stratégie par le code correspondant, ainsi que les appels récursifs provenant d'un combinateur `MuVar` par l'appel récursif correspondant.

Une telle compilation du graphe de stratégie peut être envisagée comme une phase de compilation séparée, mais les stratégies alors exécutées perdraient leurs caractéristiques réflexives, le graphe étant compilé en un seul objet.

Afin de préserver les fonctionnalités réflexives, il faut préserver le graphe de stratégies. On doit alors effectuer la compilation du graphe de stratégie lors de l'exécution de la stratégie, et compiler les stratégies « juste à temps ». Une telle transformation est à l'étude, en collaboration avec Jérémie Delaitre, utilisant une librairie de manipulation de code objet Java, ASM [BLC02]. Celle-ci permet d'extraire et analyser le code objet correspondant aux différentes méthodes `visit` du graphe de stratégie, ainsi que de recomposer une implantation pour la stratégie dans son entier. Le code est alors généré à la volée lorsque la stratégie doit être appliquée.

6.5.4. Vers le ρ-calcul ?

On encode le ρ-calcul avec filtrage unitaire du premier ordre (même si on décompose la flèche des règles, ce n'est pas de l'ordre supérieur) et structures sans superposition, et avec appel par valeur.

La structure est encodée par un opérateur variadique et son opérateur de congruence associé, qui simule la distributivité de la structure.

Le ρ-calcul constitue très probablement un environnement pour permettre d'exprimer la sémantique des stratégies que l'on peut manipuler avec ce langage. En particulier, les aspects réflexifs du langage de stratégie pourraient être explicités en utilisant une sémantique pour les stratégies basées sur ce formalisme, ainsi que certains traits de la

gestion du non-déterminisme, qui peuvent être traités par la structure dans le ρ-calcul. Ainsi, une sémantique formelle du langage de stratégie ainsi que de ses particularités propres pourrait être donnée, constituant une étape vers la certification formelle d'une librairie de stratégies.

6.6. Synthèse

6.6.1. Bilan

Partant de deux langages de stratégies, inspirés de ceux des langages ELAN et Stratego, et en s'appuyant sur l'expérience de la librairie JJTraveler pour ce qui est d'intégrer de tels combinateurs de stratégie au sein d'un langage orienté objets, nous avons construit un langage de stratégies puissant qui s'intègre dans l'environnement de développement Tom.

Ce langage permet d'encoder les fonctionnalités principales de Stratego dans le langage Java, comme les stratégies, la propagation d'information, les règles dynamiques. De plus, en fournissant le moyen de spécifier des stratégies paramétrées, ces paramètres pouvant être soit des stratégies, soit des objets Java, ce langage apporte une grande souplesse dans la définition d'algorithmes complexes. Ainsi, une stratégie peut être paramétrée par un objet représentant un contexte d'exécution, qui peut être modifié pendant le parcours, ou encore permettre d'obtenir des stratégies génériques et réutilisables dans différents contextes.

L'intégration de ce langage de stratégies dans l'environnement de développement Tom permet non seulement de fournir des constructions (**%match** et **%strategy**) autorisant la définition aisée de stratégies au niveau de l'utilisateur, mais aussi des combinateurs de stratégie dépendant de la structure de termes utilisée, qu'ils soient des opérateurs de congruence ou des opérateurs de construction. Ces opérateurs permettent alors d'encoder la réécriture de termes, et une version simple du ρ-calcul dans un environnement Java.

Finalement, en rendant les expressions de stratégie réflexives, permettant à l'utilisateur de modifier les stratégies pendant l'exécution du programme, et aussi en permettant, par la gestion fine des positions de réécriture, de gérer le non-déterminisme, ce langage de stratégie permet d'augmenter l'expressivité de l'environnement.

6.6.2. Encore plus d'expressivité ?

Les stratégies définies en utilisant ce langage de stratégies définissent une forme d'*itérateurs* sur les nœuds des arbres traversés. Cependant, on ne peut pas comme c'est le cas dans les itérateurs classiques, contrôler la manière d'itérer, c'est-à-dire qu'il n'est pas possible par exemple d'exécuter une stratégie *pas-à-pas*, en contrôlant l'exécution d'un nouveau pas depuis l'extérieur du programme. La possibilité de placer une forme de « points de contrôle » pouvant stopper et reprendre l'exécution d'une stratégie permettrait de faciliter la compréhension du comportement de stratégies complexes, par exemple en attachant un déboggueur affichant le graphe de la stratégie et son état ainsi que la position dans le terme, permettant d'exécuter la transformation pas à pas.

Une telle fonctionnalité pourrait être offerte par l'utilisation de co-routines, ou par l'adoption d'un style de programmation par continuations, l'évaluation de la stratégie retournant la stratégie correspondant à l'étape suivante du calcul.

D'autre part, ces stratégies « interruptibles » car le déroulement du calcul peut être contrôlé depuis l'extérieur pourraient permettre d'implanter une forme de *backtracking*, en offrant la possibilité de sauvegarder l'état d'une stratégie à une étape donnée, pour ensuite le restaurer si nécessaire, en modifiant la stratégie pour explorer une autre branche du calcul.

7. Développement de Tom

> We reject kings, presidents, and voting. We believe in rough consensus and running code.
>
> *(Dave Clark)*

Les contributions de cette thèse ont toutes nécessité un certain support de développement. Ainsi, le développement de la méthode de validation de la compilation du filtrage à conduit à intégrer cette validation au sein du compilateur Tom. De même, le compilateur pour le langage Gom à été écrit comme un programme Tom *boostrappé*, puisque utilisant des structures Gom pour représenter les AST. Le langage de stratégies a lui aussi été développé au sein de ce projet, et intégré très tôt dans le processus de développement du compilateur lui-même.

Le compilateur Tom étant un important programme Tom, il a été un environnement de choix pour tester les différents ajouts au langage ou aux outils associés. Le compilateur Tom utilise actuellement Gom pour implémenter les différentes structures de données, dont les AST des programmes à compiler, et utilise de manière intensive le langage de stratégies introduit au chapitre 6 ainsi que le mécanisme de définition de stratégies « utilisateurs » pour chaque phase de compilation.

7.1. Contributions

7.1.1. Contraintes dans la compilation du filtrage

Le filtrage non-linéaire est une particularité des motifs de Tom. Il est absent des langages de motifs de la plupart des langages fonctionnels comme ML ou Haskell.

Les filtres non-linéaires sont particulièrement utiles lorsqu'il s'agit d'explorer des listes, comme on a déjà pu le voir dans les exemples 7 et 8 dans le chapitre 2, en permettant d'éliminer un grand nombre de filtres inutiles produits par le filtrage de liste. La technique utilisée par le compilateur Tom était de rendre le motif linéaire, en attribuant aux différentes occurrences de la variable répétée des noms frais. L'égalité de ces variables une fois instanciées par filtrage est alors vérifiée avant d'exécuter l'action. Le motif présenté en figure 7.1a est compilé de manière similaire au motif de la figure 7.1b.

Cette technique a l'avantage d'être simple : les motifs sont linéarisés, et il est alors possible d'utiliser l'algorithme de compilation du filtrage linéaire pour compiler un motif non-linéaire. Cependant, le code alors généré est inefficace, car tous les filtres pour le motif linéarisé sont explorés complètement. Cela est particulièrement sensible lorsque le

```
%match(List l) {
  conc(_*,x,_*,x,_*) -> {
    // find a
    // duplicated "x"
    print(‘x);
  }
}
```

(a) Recherche de doublons non-linéaire

```
%match(List l) {
  conc(_*,x,_*,y,_*) -> {
    if(‘x.equals(‘y)) {
      print(‘x);
    }
  }
}
```

(b) Recherche de doublons avec test

FIG. 7.1: Recherche de doublons par filtrage non-linéaire, et équivalent utilisant un test

motif comporte de nombreuses sous-listes ainsi qu'une variable répétée un grand nombre de fois.

Exemple 38. Le filtrage de liste de Tom permet de manipuler les chaînes de caractères comme des listes. Une utilisation courante du filtrage de listes sur les séquences de caractères est le découpage d'une ligne en plusieurs sous-chaînes en fonction d'un caractère particulier. On peut ainsi vouloir explorer les différentes entrées d'un fichier `/etc/passwd`, qui est constitué de lignes contenant 7 entrées séparées par le caractère « : ». L'opérateur de liste associé à la sorte `String` étant unique, il n'y a pas d'ambiguïté, et le nom de l'opérateur peut être omis dans l'écriture de filtre, qui peut alors être écrit comme :

```
%match(String entry) {
  (login*,x,pass*,x,uid*,x,gid*,x,name*,x,path*,x,shell*) -> {
    if(':' == ‘x) {
      // some action
    }
  }
}
```

Ce motif cherche 7 champs séparés par un même caractère, et exécute l'action si ce caractère est bien « : ».

Le filtre de l'exemple 38 lorsqu'il est compilé en utilisant la linéarisation du filtre, introduisant alors 5 variables fraîches, et effectuant un test vérifiant que les valeurs attribuées par filtrage à chacune d'elles sont égales avant d'exécuter l'action, produit un algorithme de filtrage très inefficace. En effet, tous les découpages de la chaîne de caractères d'entrée sont explorés entièrement.

On introduit afin de compiler ces filtres de manière efficace une notion de contrainte associée aux variables. Ainsi, à chacune des variables fraîches introduites par la linéarisation du motif est associé une contrainte d'égalité avec la première variable `x`. La compilation de l'algorithme de filtrage est alors faite en tenant compte de ces contraintes : dès

qu'une quantité suffisante d'information est disponible au cours du processus de filtrage, il est vérifié si la contrainte est satisfaite. Ceci permet de ne pas continuer le découpage de la chaîne lorsque le début du découpage n'est pas valide, c'est à dire qu'un champ a été délimité par des caractères différents.

La figure 7.2 illustre le temps d'exécution du motif de l'exemple 38 avec le schéma de compilation sans contraintes, puis en considérant les contraintes d'égalité au niveau des variables. La découpe d'une seule entrée du fichier `/etc/passwd`, par exemple :

```
postfix:*:27:27:Postfix User:/var/spool/postfix:/usr/bin/false
```

prend environ 10 secondes dans le premier cas, pour un temps inférieur à 1 milliseconde dans le second cas, lorsqu'elle est exécutée sur une machine équipée d'un processeur Pentium 4 cadencé à 3 gigahertz.

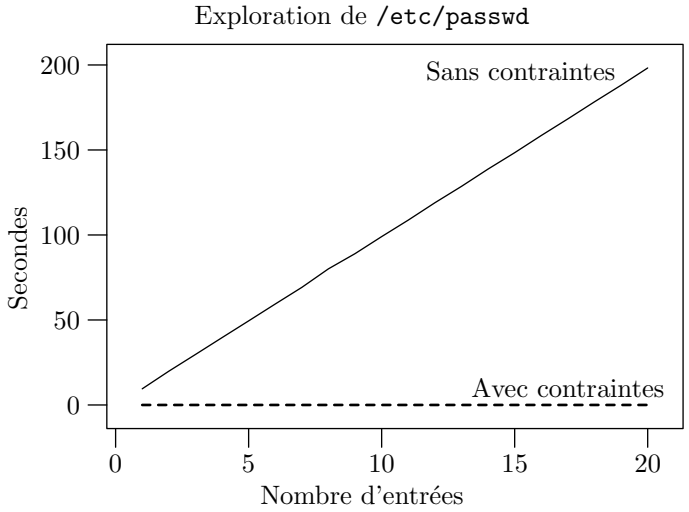


Fig. 7.2: Tests du filtrage associatif non-linéaire avec et sans contraintes

La gestion et la propagation des contraintes d'égalité sur les variables au sein de l'algorithme de filtrage permet d'obtenir des gains d'efficacité substantiels pour les motifs utilisant plusieurs variables non linéaires.

Cependant, le motif de l'exemple 38 peut être rendu plus efficace en considérant non seulement une variable non-linéaire `x`, mais directement la constante « `:` » au sein du motif, comme une constante algébrique. Les premières versions du compilateur ne pouvaient pas effectuer une telle optimisation.

On étend alors la notion de contrainte gérée par le compilateur des contraintes d'égalité entre variables à des contraintes plus générales. Parmi elles, on considérera les contraintes d'égalité d'une variable avec un constante. Le motif de l'exemple 38 peut alors s'écrire en utilisant la constante `':'` en lieu et place des variables `x`. Les contraintes d'égalité à cette constante sont alors générées au sein de l'algorithme de filtrage, permettant de ne par continuer le découpage de l'entrée lorsque le caractère choisi comme séparateur de champs n'est pas « `:` ».

Le motif s'écrit alors :

```
%match(String entry) {
  (login*,':',pass*,':',uid*,':',gid*,':',name*,':',path*,':',shell*)->{
    // some action
  }
}
```

On compare en figure 7.3 l'efficacité du motif de l'exemple 38 compilé avec l'algorithme de propagation de contraintes d'égalité de variables avec la version utilisant aussi des contraintes d'égalité entre variables et constantes. La seconde version s'avère environ deux fois plus rapide, car le découpage correct est trouvé sans essai de découpages incorrects, utilisant 6 occurrences d'un caractère différent de « : ». L'évaluation du second motif est linéaire en la taille des entrées, tandis que l'efficacité du premier dépend de la fréquence et de la répartition des caractères dans les entrées ; si il y a beaucoup de redondance, de nombreux découpages incorrects seront testés.

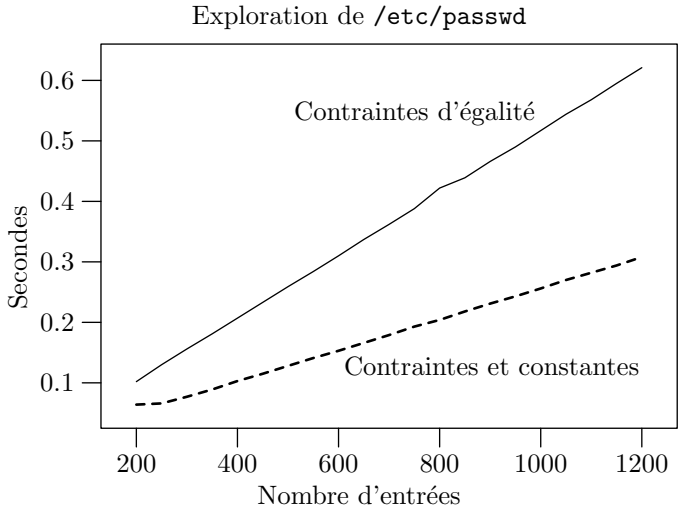


FIG. 7.3: Tests du filtrage associatif non-linéaire avec et sans gestion des constantes

Filtrage avec contraintes. L'introduction tout d'abord de contraintes d'égalité entre variables afin de permettre de mieux compiler les motifs non-linéaires, ainsi que leur propagation au sein de l'algorithme de filtrage compilé lui-même, puis l'ajout de contraintes d'égalité entre variables et constantes permettent l'obtention de gains en efficacité importants, sur des applications d'utilité commune, comme la manipulation de chaînes de caractères.

L'implémentation et la conception de ce mécanisme de gestion de contraintes au niveau de l'algorithme de filtrage ont été faites de manière générique et extensible, fournissant une base pour l'intégration de filtrage avec des contraintes arbitraires. L'objectif est

de pouvoir compiler efficacement des conditions attachées aux motifs comme autant de contraintes sur les variables de filtrage.

7.1.2. Extensions de ApiGen

Avant l'introduction de Gom, ApiGen était l'un des outils clé du développement avec Tom, générant les structures de données utilisées pour représenter les AST du compilateur, ainsi que les structures de données des différents exemples. Certaines extensions du langage de spécification de signatures d'ApiGen ont du être introduites, afin de permettre l'utilisation plus facile de ces structures.

C'est en partie ces extensions qui ont guidé la conception du langage Gom, ainsi que l'expérience acquise en modifiant le compilateur ApiGen qui a dirigé la conception de son compilateur.

Symboles de liste nommés

Une première extension a été de permettre la définition des noms des symboles associatifs. En effet, le langage de spécification original d'ApiGen permet de définir un opérateur de liste en fonction de son domaine ainsi que de son co-domaine. De plus, il n'est possible de définir qu'un seul opérateur de liste par co-domaine, et cette sorte ne peut contenir aucun autre opérateur. Ainsi, la définition d'un opérateur de liste de sorte `List` agrégeant des éléments de sorte `Element` s'écrit :

```
list(List,Element)
```

Le nom de l'opérateur de liste associé était alors dérivé automatiquement du nom du co-domaine, ici `concList`. Nous avons introduit une déclaration de liste nommée, permettant de donner un nom à l'opérateur de liste, afin de simplifier l'écriture des motifs et constructions de termes dans les programmes. La déclaration de la liste précédente peut alors être :

```
named-list(conc,List,Element)
```

Une telle spécification permet alors d'utiliser un nom significatif comme opérateur de liste. Une liste d'instructions dans l'AST d'un programme sera alors identifiée par le constructeur `Sequence` plutôt qu'un constructeur `concInstruction` moins lisible.

Modularité des signatures

Un autre ajout au langage de spécification de signatures d'ApiGen est celui de modules pour les signatures. Les signatures ApiGen sont en effet contenues dans un seul fichier, et ne peuvent pas être réutilisées. Elles contiennent une liste de déclarations de constructeurs et d'opérateurs de listes.

Lorsque l'on doit manipuler une signature de grande taille, comme peut l'être celle qui décrit les AST d'un compilateur, il est nécessaire d'être capable de distinguer les différentes parties de la spécification des AST, par exemple en séparant les instructions du langage de ce qui concerne le typage.

On introduit alors une construction `modulentry` à ApiGen. Celle-ci permet de déclarer un nouveau module, qui éventuellement importe lui même d'autres modules, avec la liste des sortes qu'il contient, suivie par la liste des déclarations d'opérateurs. Le module déclarant les sortes et opérateurs implémentant les entiers de Peano peut alors être :

```
[
  modulentry(
      name("Peano"),
      [],
      [type("Nat"),type("List")],
      [
        constructor(Nat,Zero,Zero()),
        constructor(Nat,Suc,Suc(pred<Nat>)),
        named-list(conc,List,Nat)
      ]
  )
]
```

Les importations de modules sont résolues à la compilation, et leur cohérence vérifiée. Les modules sont alors aplatis avant la compilation. Le code à générer pour chaque module ne dépend que de la « signature » des modules qu'il importe ou qui sont importés par l'un de ces modules, c'est-à-dire le nom du module et des sortes qu'il déclare. Il faut alors calculer la fermeture réflexive et transitive de la relation d'importation de modules afin de déterminer les modules qui doivent être recompilés lors d'un changement.

7.2. Le *meta-quote*

Le développement du compilateur Gom implique l'écriture d'un générateur de code, qui permet de générer les classes Java correspondant à la signature algébrique. L'écriture d'un tel générateur n'est pas aisée dans le langage Java, le code à générer pour chaque élément de l'AST pouvant être long.

Exemple 39. On peut illustrer le problème que pose l'écriture d'un tel générateur par l'écriture d'un générateur simple, dont l'objectif est de produire le programme `HelloWorld` classique, ainsi que de donner un numéro de version.

```
public class GenHello {
  static String version = "v12";
  public static void main(String[] args) {
    String prg = "";

    prg += "public␣class␣Hello␣{\n";
    prg += "\tpublic␣static␣void␣main(String[]␣args)␣{\n";
    prg += "\t\tSystem.out.println(";
```

```
    prg +=  "\"Hello\\n\\tWorld\\t"+version+"\""
    prg += ");\n";
    prg += "\t}\n";
    prg += "}\n";

    System.out.println(prg);
  }
}
```

Le code à générer est ici très simple, mais le programme qui le génère n'est par contre par très facile à comprendre.

On introduit pour faciliter le développement de tels générateurs de code une construction %[...]% permettant de générer des chaînes de caractères formatées. Celle-ci permet l'écriture de chaînes contenant des caractères qui doivent dans le langage hôte être échappées, comme les retour chariots, les tabulations, ou le caractère « \ ».

Le code source placé entre %[...]% est alors traité sans modifications comme une chaîne de caractères, et les échappements corrects sont calculés. Seul le caractère « @ » est interprété différemment, celui-ci permettant d'insérer dans la chaîne formaté par la construction %[...]% le résultat de l'évaluation d'une fonction Java ou le contenu d'une variable. On peut alors écrire le programme précédent de manière plus simple.

Exemple 40. Programme équivalent à celui de l'exemple 39, utilisant la construction *méta-quote* introduite dans Tom :

```
public class GenHello {
  static String version = "v12";
  public static void main(String[] args) {

    String prg = %[
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello\n\tWorld\t@version@");
  }
}
    ]%;

    System.out.println(prg);
  }
}
```

La classe à générer est alors lisible explicitement. Le mécanisme « @ » permet d'intégrer au programme généré la valeur de la variable `version`.

7.2.1. Travaux en rapport

La construction *méta-quote* permet entre autres d'écrire de programmes à l'intérieur de programmes. Les langages Meta-AspectJ [ZHS04] et Java-Java [BVVV05] ont été développés expressément dans le but de faciliter l'écriture de tels programmes générant de nouveaux programmes. Le premier permet par l'introduction de constructions syntaxiques similaires à celles que nous venons d'introduire, ainsi qu'un mécanisme permettant de déclarer des variables contenant des morceaux de programme à générer, de construire des programmes AspectJ au sein de programmes AspectJ, qui est une extension à Java introduisant des mécanismes de programmation par aspects. Java-Java est une extension au langage Java permettant de construire des programmes Java dans le langage Java, en ajoutant une fois encore des constructions similaires : des mots clés délimitant le programme ou morceau de programme à construire, et un mécanisme d'*échappement* permettant d'introduire dans ce programme le contenu de variables ou le résultat de l'évaluation de fonctions.

La différence principale entre ces mécanismes de *méta-programmation* et la construction *méta-quote* introduite dans Tom est que notre construction ne considère le produit que comme une chaîne de caractères, tandis que les mécanismes de Meta-AspectJ et Java-Java construisent un AST typé pour le programme à générer. L'avantage essentiel de la construction d'un AST est que le compilateur lui-même peut vérifier la validité syntaxique du programme généré : le typage interdit par exemple de placer une expression là ou une instruction est attendue. Cela impose l'utilisation d'une machinerie lourde afin de parser les morceaux de programmes à construire, puis effectuer une inférence de types, pour ensuite vérifier leur bonne utilisation. Le mécanisme est alors dépendant du langage des programmes à générer.

Le mécanisme *méta-quote* au contraire est très léger, et ne représente qu'une trentaine de lignes de code au sein du compilateur Tom, car il réutilise des méthodes utilisées par ailleurs, pour gérer le code hôte dans le compilateur. Il ne fournit aucune garantie de validité syntaxique sur le programme qui est généré, mais offre une grande souplesse, car il n'est lié à aucun langage cible. Il peut être utilisé pour générer des programmes Java et Tom, comme il l'est pour le développement du compilateur Gom, mais aussi pour générer les représentations graphiques des stratégies comme celle de la figure 6.7 dans le langage `dot`, ou pour générer des témoins de preuve imprimables et lisibles dans le langage LaTeX pour le vérificateur de compilation présenté au chapitre 3.

7.3. Méthodes

Durant cette thèse, j'ai eu la chance de participer au développement du systèmes Tom, non seulement par les contributions présentées jusque-là, mais aussi en tant que contributeur à un projet « *open-source* ».

Dans ce cadre, j'ai pu participer aux différentes évolutions du projet, ainsi qu'aux phases de décision qui ont conduit à ces évolutions. Ces évolutions font de Tom non seulement un outil de recherche, mais aussi un environnement de développement fiable et robuste.

7.3.1. Compilation de Tom

Le compilateur Tom est écrit en utilisant Tom avec comme langage hôte Java. De plus, il utilise le langage ApiGen décrit en section 4.1.2 pour générer une implantation typée pour les AST manipulés. Les AST ont ensuite été convertis pour être générés par le compilateur Gom.

Utilisation de Apache Ant

Le processus de compilation du projet était alors géré par un ensemble de scripts et de fichiers `Makefile` générés par les outils `autoconf` et `automake`. Ces scripts se chargeaient de relancer les outils comme ApiGen ainsi que d'appeler successivement le compilateur Tom en sa version stable, puis le compilateur `javac` pour obtenir une version compilée de Tom. Cependant, alors que le projet devenait plus important, ces scripts se faisaient de plus en plus difficile à maintenir.

De plus, les utilisateurs de l'environnement hors du projet lui-même devaient à leur tour développer des scripts équivalents pour construire leurs projets utilisant Tom.

Nous avons pour simplifier le processus ainsi que le rendre portable, conçu un nouveau processus de compilation pour le projet, basé sur l'utilisation de l'outil *Apache Ant*[1]. Cet outil permet une spécification plus déclarative du processus de compilation, ainsi que l'intégration plus facile d'outils indépendants. De plus, c'est un outil d'utilisation courante dans la communauté Java, permettant une approche plus facile du projet pour les nouveaux utilisateurs.

Intégration des outils

Afin de simplifier l'utilisation des divers outils fournis par l'environnement Tom, comme le compilateur, ApiGen ou Gom, nous avons défini des modules, appelés *tâches*, pour *Apache Ant* permettant l'utilisation de ces outils avec le même facilité que le compilateur `javac`.

Nous avons ainsi pu redéfinir le processus de *bootstrap* du compilateur, en incluant la génération des structures de données par le compilateur Gom. Ce *bootstrap* permet non seulement de construire les compilateurs Tom et Gom, mais aussi de les tester, ainsi que de tester systématiquement le bon fonctionnement des tâche pour *Ant*.

Un autre intérêt à l'intégration de nos outils dans un environnement de compilation comme *Ant* est qu'il est maintenant possible d'intégrer de nouveaux exemples d'utilisation de ces outils très simplement. Un nouveau dossier est créé pour l'exemple, et la compilation de cet exemple est décrite dans un fichier de quelques lignes. Une fois ceci fait, il est possible de compiler cet exemple avec la version de développement du compilateur, et cet exemple est intégré au jeu de tests de non régression du compilateur ; à chaque *bootstrap*, l'exemple sera compilé par la nouvelle version du compilateur, afin de s'assurer que cette nouvelle version fonctionne bien.

[1] `http://ant.apache.org`

Tests unitaires

Les tests de non régression que constituent les exemples d'utilisation du langage sont très utiles : ils permettent d'être sûr qu'un exemple qui était compilé par la version précédente du compilateur l'est toujours. Une fois couplés avec la procédure de validation de la compilation présentée au chapitre 3, ils permettent en outre de s'assurer que le filtrage est toujours compilé correctement.

Cependant, ces tests ne couvrent pas l'ensemble des fonctionnalités du langage, ni les cas problématiques que peuvent poser certaines constructions. C'est ainsi qu'un ensemble de tests de non régression plus spécifique à été écrit. Ces tests ciblent chacun une fonctionnalité différente du langage. Non seulement nous testons si ces programmes de test sont compilés correctement, mais aussi si leur exécution est bien conforme aux attentes. Pour ceci, nous utilisons la bibliothèque `junit`[2], qui nous permet d'intégrer l'exécution de ces jeux de tests dans la procédure de compilation, *bootstrap* et diffusion des outils.

Ces tests unitaires ne sont pas des tests unitaires dans le sens que leur donne *extreme programming*[3], car ils testent le comportement du compilateur et des programmes compilés par celui-ci. Cependant, ils sont conçus pour vérifier le bon fonctionnement des différents aspects du compilateur et de la librairie.

7.3.2. Plateforme de compilation à *Greffons*

Le compilateur Tom consiste en différentes phases (analyseur syntaxique, vérificateur de types, traduction des constructions apportant du *sucre syntaxique*, compilateur, générateurs de code,...). Il est nécessaire de pouvoir activer et désactiver aisément chacune de ces phases, comme l'optimiseur de code [BM06]. Il faut aussi pouvoir gérer les différentes options de compilation pour activer chacune de ces phases (par exemple, pour choisir le générateur de code adapté au langage hôte).

De plus, ce compilateur est un outil de recherche, et il est nécessaire d'avoir la possibilité d'ajouter ou supprimer de nouvelles phases et de nouvelles options au compilateur sans requérir de changement profonds dans l'architecture du compilateur.

C'est ainsi qu'avec Pierre Étienne Moreau, Julien Guyon et Gregory Andrien, nous avons conçu une plateforme dans laquelle chaque phase de compilation est un greffon. Ces greffons sont découverts dynamiquement, à partir d'un fichier de configuration listant les différents greffons que le compilateur doit utiliser. De plus, ces greffons lorsqu'ils sont activés, doivent déclarer les différentes options qu'ils rendent disponibles à l'utilisateur (par exemple, le greffon *optimiseur* déclare les différents niveaux d'optimisation possibles). La plateforme vérifie alors la cohérence de ces informations, et permet l'utilisation de ces options dans la ligne de commande d'appel au compilateur.

La gestion de l'analyse de la ligne de commande, de la gestion des options en ligne de commande de l'outil ainsi que du lancement des différents greffons dans l'ordre spécifié est déléguée à la plateforme, tandis que les constituants du compilateur sont des *greffons*

[2] `http://www.junit.org/`
[3] `http://www.extremeprogramming.org/`

développés indépendamment — mais manipulant un AST commun — qui coopèrent. Il est ainsi possible d'intégrer de nouveaux greffons au système sans nécessiter de changements dans les parties existantes du compilateur. Ceci permet d'intégrer et de tester de nouvelles fonctionnalités pour le langage ou l'environnement en limitant l'impact sur l'existant. Le vérificateur de filtrage présenté au chapitre 3 est ainsi intégré comme un greffon, qui est indépendant du reste du compilateur.

Cette plateforme est indépendante du compilateur Tom lui-même, est peut être réutilisée. Le compilateur Gom est ainsi lui aussi basé sur cette plateforme, et constitué de greffons indépendants.

7.3.3. Environnement intégré de programmation

Comme nous l'avons présenté au début du chapitre 2, le langage Tom n'était pas destiné lors de sa création à être utilisé par un développeur humain, mais constituait plutôt un langage de « bas niveau » pouvant servir de langage intermédiaire pour un compilateur le langage à base de règles, comme Elan.

C'est alors qu'il est apparu que ce langage pouvait être utile aussi pour un développeur humain. L'environnement de développement était cependant spartiate et minimaliste. L'introduction de la plateforme à greffons à tout d'abord permis d'obtenir un compilateur utilisable en ligne de commande qui offre des options cohérentes avec l'existant, tout en diminuant l'effort de maintenance.

Une attention toute particulière a été portée à la facilité d'utilisation de ces outils, en particulier pour le problème de la détection et de la correction des erreurs de programmation. Ainsi, les erreurs et problèmes potentiels détectés par le compilateur Tom sont présentés à l'utilisateur avec le numéro de ligne correspondant à l'erreur. Un des problèmes principaux de l'utilisation d'un préprocesseur est que les erreurs non détectées par le préprocesseur, mais qui le sont par le compilateur travaillant sur le code produit sont généralement reportées sur une ligne ne correspondant pas à la position réelle de l'erreur dans le programme original, avant le passage du préprocesseur. Cet écueil est contourné par le compilateur Tom en usant d'un générateur de code s'assurant que les éléments de code hôte dans un programme Tom sont générés sur la même ligne que celle qu'ils occupaient dans le programme source. Ainsi, les messages d'erreur du compilateur Java ou C peuvent être utilisés pour corriger le programme. Le compilateur effectue un important travail de vérification des parties Tom des programmes, afin de toujours générer du code ne posant pas de problèmes à la compilation dans le langage hôte : les seules erreurs que le compilateur Java doit avoir à remonter sont les erreurs réelles de l'utilisateur en programmation Java.

Nous avons ensuite fourni un ensemble d'outils pour intégrer la compilation des programmes Tom dans l'environnement de compilation de projets *Apache Ant*, ainsi que des greffons pour l'éditeur `vim`[4], fournissant ainsi un environnement de développement minimal, mais efficace.

Cependant, si l'objectif est d'apporter au développeur Java la possibilité d'utiliser des

[4] `http://www.vim.org`

méthodes issues de la spécification algébrique, cet environnement n'est pas suffisant. L'objectif est de faire en sorte que l'utilisation de Tom et des outils associés (Gom, stratégies, ...) ne demande pas trop de connaissances préalables, ni ne requiert d'effort pour être intégré dans un développement. L'intégration de nos outils dans un environnement de développement intégré (IDE) reconnu est alors nécessaire.

Nous avons alors développé un *plugin* pour la plateforme Eclipse[5] [ecl01]. Celui-ci intègre entièrement l'utilisation du langage Tom dans un IDE [GMR04] pour Java, fournissant coloration syntaxique, compilation des programmes Tom et signatures Gom, ainsi que les rapports d'erreur. La compilation des programmes Tom étant automatique, les messages d'erreurs des compilateur Tom et Java sont intégrés, et présentés à l'utilisateur comme un tout, ce qui permet d'abstraire la phase de compilation par le compilateur Tom puis Java, et de présenter à l'utilisateur une interface unifiée.

Il est alors possible d'utiliser Tom aussi simplement que Java dans un nouveau projet, afin d'implémenter des transformations de structures générées par Gom, de documents XML ou de structures arbitraires. Cela aide à promouvoir l'utilisation de Tom dans des communautés plus larges.

7.4. Synthèse

La participation au développement du compilateur Tom et des outils associés à été une activité importante tout au long de cette thèse. Cela passe par des aspects de développement permettant d'appuyer l'effort de recherche, mais aussi par la vie et la gestion d'un projet de développement open source.

Le compilateur Tom a beaucoup évolué, parallèlement au déroulement de cette thèse, et j'ai pu apporter aide et soutient lors des cette évolution. J'ai pu particulièrement m'impliquer dans la résolution de quelques problèmes d'efficacité, ainsi que dans l'établissement de pratiques de développement dont l'objectif est de permettre un développement plus sûr et plus facile du compilateur.

Ceci passe par des outils permettant de compiler et tester ce compilateur de manière efficace, afin de limiter le nombre d'erreurs introduites. Aussi, une meilleure intégration dans une chaîne de compilation ou un IDE comme Eclipse permettent de faciliter l'accès au langage.

[5] http ://www.eclipse.org/

Conclusion

> The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.
>
> *(Douglas Adams)*

L'objectif de cette thèse est de se donner des moyens d'écrire plus facilement et plus efficacement des compilateurs plus sûrs. Notre démarche n'était pas focalisée sur le fait de produire des preuves formelles de cette sécurité, mais aussi de donner des outils pratiques permettant de faciliter l'écriture de compilateurs ou transformateurs de documents corrects.

Compilation

Une première contribution a été de formaliser les relations qui existent entre la description abstraite d'une extension de langage — l'ajout de constructions de filtrage dans Java ou C — et les parties de programme générées par la compilation de ces constructions. On a ainsi présenté la notion d'*ancrage formel* qui relie les données algébriques abstraites à leur représentation concrète, puis utilisé cette notion afin de fournir un cadre à la certification de la compilation du filtrage.

Cette certification se distingue par plusieurs caractéristiques :

- elle certifie le travail effectué par le compilateur, et non le compilateur lui-même ;
- elle est indépendante du fonctionnement de ce compilateur ;
- elle est indépendante du langage hôte (Java, C ou ML) ;
- elle s'étend à la compilation de filtrage non-linéaire, et de constructions multi-filtres ;

Cette méthode de certification est intégrée au compilateur Tom, et utilisée pour valider la compilation de programmes, dont le compilateur Tom lui-même.

La méthode de certification présentée traite de la compilation du filtrage algébrique de Tom avec ses particularités propre, en particulier l'indépendance vis-à-vis de la structure de données, mais peut être appliquée aux constructions de filtrage telles que celles présentes dans les langages fonctionnels, comme ML ou des langages à base de règles. Elle ne permet toutefois pas en l'état de traiter le cas du filtrage associatif avec élément neutre (encore appelé « filtrage de liste »). Cette certification impose tout d'abord de comprendre les liens entre les structure de données concrête et la structure algébrique

modulo associativité et élément neutre, avant de pouvoir traiter du filtrage proprement dit.

Le travail présenté dans le chapitre 5 constitue une première étape vers ce but, en présentant comment une structure de termes variadiques (comme celle qui est générée par le compilateur Gom) peut être utilisée pour représenter des représentants canoniques pour des structures de termes possédant des opérateurs associatifs avec élément neutre. Il est aussi détaillé comment il est possible de contruire des fonctions de normalisation permettant d'obtenir des représentants canoniques pour les classes d'équivalence modulo associativité et élément neutre. Ces fonctions de normalisation sont alors certifiées correctes en utilisant l'assistant à la preuve Coq.

Il reste alors à savoir utiliser les résultats obtenus sur les structures de données pour traiter d'une part de la correction des algorithmes de filtrage associatifs, lesquels doivent être exprimés dans un langage plus expressif que le langage `PIL` présenté au chapitre 3, et d'autre part de la correction de la compilation des motifs associatifs vers ces algorithmes de filtrage.

Langage

La certification de la compilation est un aspect important, car elle nous permet d'avoir confiance dans les transformations de documents écrites en utilisant ce filtrage. Un autre aspect permettant d'augmenter la confiance dans les transformations écrites est de rendre cette écriture plus aisée, en particulier pour des transformations complexes.

L'utilisation du filtrage et de la réécriture seuls n'est pas suffisant pour décrire élégamment des transformations de document complexes. Il est nécessaire de pouvoir tout d'abord contrôler la forme des données à manipuler, puis contrôler la manière d'enchaîner les transformations.

Données Pour cela, nous avons proposé un langage de descriptions de structures arborescentes qui permet non seulement d'obtenir une implémentation efficace de ces arbres, mais aussi de spécifier des *invariants* dans la structure des arbres dont la préservation est garantie.

L'intégration de ces invariants au sein même de la structure de données permet alors de simplifier les transformations, qui manipulent des *représentants canoniques*.

Filtrage L'étude des liens entre représentation concrète des structures de données variadiques manipulées par Tom et les structures de termes modulo associativité et élément neutre, présentée dans le chapitre 5 permet de comprendre comment le filtrage de liste permet d'écrire des algorithmes traitant de structures associatives.

Stratégies D'autre part, afin d'offrir un moyen élégant et efficace de décrire les procédures de transformations, nous avons présenté un langage de stratégies permettant dans un environnement de programmation Java de décrire des stratégies de parcours

et de transformation de termes. Ce langage offre la possibilité de manipuler explicitement les positions de réécriture, offrant ainsi un contrôle sur le non-déterminisme des transformations.

Ces éléments combinés constituent un environnement fiable et puissant pour le développement de transformations de documents. En utilisant d'une part un compilateur capable de produire un certificat montrant que le programme effectuant la transformation correspond bien à sa spécification, et d'autre part des mécanismes autorisant la spécification à haut niveau d'invariants de structure de données ainsi que de stratégies de transformation rend alors l'écriture de transformations « de confiance », et ce d'une manière qui s'intègre entièrement au langage Java. Cette intégration permet alors d'utiliser ces techniques et transformations au sein de projets plus complexes, sans introduire de contraintes sur le reste du projet.

Perspectives

Les perspectives de ce travail sont multiples, la confiance dans les outils de transformation de programmes ou documents a encore besoin d'être confortée.

Différents travaux peuvent être entrepris pour étendre les contributions de cette thèse, à plus ou moins longue échéance, visant à étendre le champ de la certification du filtrage, ou l'expressivité du langage de stratégies ou des structures de données.

Filtrage équationnel Certification de la compilation de filtrage équationnel, et surtout du filtrage « de liste » de Tom. Les résultats présentés au chapitre 5 permettent d'établir les liens entre structures variadiques et les structures algébriques associatives. Il faut alors établir la correspondance entre le filtrage de liste effectué par Tom sur ces structures et le filtrage associatif avec élément neutre, afin de pouvoir certifier les algorithmes sur des objets associatifs écrits avec le filtrage équationnel.

Certification des termes de stratégies L'utilisation d'un langage de stratégies permet de séparer règles et contrôle lors de l'implantation d'un système de calcul ou de transformation. Cependant, afin de prouver le bon comportement d'une transformation exprimée en utilisant des stratégies, il est nécessaire de prendre en compte la sémantique de ces stratégies. Nous voulons alors nous donner les moyens de certifier que l'implémentation d'un certain terme de stratégie à bien une sémantique conforme à la sémantique formelle de la stratégie. Le fait de constituer systématiquement les stratégies comme un assemblage de stratégies « élémentaires », et ce aussi au niveau de leur implémentation, permet de donner des éléments de réponse. Il faut alors certifier individuellement chaque brique de base du langage de stratégies comme ayant une sémantique correspondant à sa sémantique formelle, puis assurer que l'assemblage d'objets constituant la stratégie est bien similaire à la stratégie que l'on veut certifier. Il faut alors pouvoir raisonner sur les propriétés d'exécution d'un ensemble d'objets interagissant, en utilisant un modèle de cet assemblage de classes.

Stratégies interruptibles Une meilleurs intégration de la librairie de stratégies, en particulier rendre plus aisée l'utilisation du compilateur « juste à temps » de stratégies. Un autre point important est la conception de stratégies *pas-à-pas*, qui rendent le déboguage d'applications utilisant des stratégies plus facile.

Compilation par îlots On a permis la certification de la compilation de constructions de filtrage. Un point qui n'est pas traité concerne la validité du remplacement des parties de code « hébergé » au sein de programmes hôtes. On a montré en figure 2.1 que le compilateur Tom ne modifie pas la *forme* du programme : les parties du programme hôte qui ne concernent pas les constructions Tom ne sont pas touchées. Cependant, on peut se demander si les différentes constructions de filtrage — même si elles sont correctement compilées — sont insérées à la bonne position dans le programme.

Vers des aspects pour Gom La définition de *hooks* dans le langage Gom peut être vue comme la définition d'aspects modifiant le comportement des fonctions de construction des objets qui sont générés. Ceci peut être étendu en définissant de nouveaux points pour lesquels une modification du comportement est possible. Ainsi, on veut pouvoir insérer de nouvelles fonctions à différents niveaux dans la hiérarchie d'objets créés, comme des méthodes d'affichage formaté. Il est alors utile de permettre aux classes générées d'implémenter de nouvelles interfaces.

La possibilité de modifier le traitement des différents sous-termes (ou *slots*) d'un opérateur est utile. On peut ainsi vouloir donner des valeurs par défaut à certains sous-termes qui peuvent être laissés vides par l'utilisateur, ou calculer la valeur d'un sous-terme en fonction des autres.

Ceci conduit à développer un langage pour spécifier plus finement les points de greffe dans le code à générer pour la structure de données. On peut aussi considérer des points d'action dynamiques, n'étant déclenchés que dans des cas de figure particuliers.

Certifier des invariants Le système de *hooks* présenté est très souple : il est possible d'utiliser du code Java arbitraire. Cependant, une utilisation importante de ce mécanisme est la spécification d'invariants assurant que les termes manipulés sont des représentants canoniques pour certaines théories que l'on attribue aux opérateurs. Étendre le langage à la spécification algébrique de ces théories et à la génération automatique de fonctions de normalisation permettrait de rendre l'écriture de telles fonctions de normalisation plus faciles. D'autre part, il serait alors possible de vérifier la cohérence des différentes règles, afin de fournir un témoin de preuve validant la fonction de normalisation produite.

À plus long terme Les deux préoccupations de cette thèse sont d'une part la certification des processus de compilation, et d'autre part l'expressivité des langages qui permettent de construire ces compilateurs. On a décrit une technique permettant de certifier la compilation de constructions de filtrage considérée comme une boîte noire. D'autres fonctionnalités du langage ne se prêtent pas aisément au même traitement, et l'on peut vouloir les certifier par d'autres techniques.

La combinaison des différentes techniques permettant de donner des garanties sur la qualité de la compilation, telles que validation de traduction, *proof carrying code* ou certification des algorithmes de compilation eux-même est nécessaire si l'on veut se donner les moyens de construire un environnement dans lequel un compilateur peut être développé par améliorations successives, et en réutilisant autant que faire se peut les techniques, algorithmes et processus provenant d'autres expériences.

Le développement d'application est rarement un processus linéaire, et généralement se fait par agrégation de composants et améliorations provenant de différentes sources. Il est important alors de se donner les moyens d'adopter un mode de développement souple pour un compilateur certifié. Ainsi, il faut pouvoir combiner différents modules dans un compilateur ainsi que les certifications correspondantes, à la manière des différentes phases d'un compilateur, pour obtenir un tout certifié.

Du point de vue de l'expressivité du langage, il est important de comprendre comment simplifier les interactions entre les différents aspects de l'environnement, Gom, Tom et le langage de stratégies. Aussi, Gom permet de manipuler des représentants canoniques des termes modulo une théorie équationnelle arbitraire. Le filtrage effectué par Tom quant à lui ne considère que des termes algébriques ou modulo associativité. On voudrait pourtant pouvoir utiliser un filtrage modulo la théorie équationnelle utilisée sur les termes, ou pouvoir contrôler les connections entre filtrage modulo associativité et représentants canoniques pour la théorie associée aux termes.

D'autre part, certaines transformations de programmes, en particulier lors de l'optimisation de code, ne sont pas aisément exprimables en terme de transformations d'arbres ou AST, car elles font intervenir la notion de flot de contrôle, et s'expriment plutôt comme des transformations de graphes. Étendre d'une part le langage Gom pour représenter de tels graphes, puis le filtrage de Tom et surtout le langage de stratégie afin d'exprimer des traversées de graphes de manière efficace est un défi à relever. La certification de la compilation de ces transformations est alors essentielle, et peut s'appliquer à des champs d'applications comme la transformation de modèles logiciels ou biologiques.

Finalement, l'objectif de construire un environnement de développement de compilateurs « de confiance » peut être poursuivi en intégrant plus étroitement les notions de transformation et de stratégie de transformations au sein d'un langage fournissant plus de garanties pour ces transformations globales. En particulier, un tel langage pourrait permettre la certification non seulement de la compilation du filtrage, mais aussi des structures de données (et leurs invariants), les stratégies de transformation ainsi que la combinaison de ces trois aspects. Cette intégration doit être faite sans empêcher l'utilisation d'un langage généraliste « moins sûr », afin de préserver souplesse et facilité de collaboration des outils produits. C'est par l'obtention d'un langage assez sûr pour écrire des transformations dont la vérification formelle est complète, ainsi que l'écriture aisée de transformations requérant un niveau de confiance moins important que l'on peut parvenir à généraliser l'utilisation de techniques formelles.

De plus, pour faire en sorte qu'un langage comme la combinaison de Tom, Gom et

Java avec un langage de stratégies puisse être accepté et utilisé par une communauté de développeurs plus large, comprenant des intérêts industriels, il est nécessaire d'une part de diminuer le niveau d'expertise nécessaire pour l'utilisation efficace du langage et d'autre part de rendre plus accessibles les concepts de programmation sous-jacents. Ceci peut être obtenu par la définition d'un langage basé sur ces idées proposant un tout uniforme, en masquant les aspects composants réutilisables derrière une syntaxe et une sémantique unifiées. De plus, afin de disséminer les idées présentées dans cette thèse, il sera important de développer et expliciter des méthodes et méthodologies de modélisation et de développement basées sur la représentation d'arbres sous forme canoniques et leur transformation à l'aide de stratégies. C'est en expliquant et en détaillant ces méthodologies de développement du point de vue de l'ingénierie de développement, ainsi que les gains apportés par ces méthodes que l'on pourra diffuser les résultats de cette recherche pour la programmation en général.

A. Annexes

A.1. Code Coq

Cette annexe contient l'ensemble de la formalisation utilisant le système Coq des liens entre opérateurs associatifs avec élément neutre et opérateurs variadiques présentée dans le chapitre 5.

```
Ltac caseq x := generalize (refl_equal x); pattern x at -1; case x.

Inductive T : Set :=
  | E : T
  | C : nat -> T
  | F : T -> T -> T
.

Fixpoint peigne (t : T) : T :=
  match t with
  | E => E
  | C x => C x
  | F x y => peigne2 x (peigne y)
  end
with peigne2 (t1 : T) (t2 : T) {struct t1} : T :=
  match t1 with
  | E => F t1 t2
  | C x => F t1 t2
  | F x y => peigne2 x (peigne2 y t2)
  end
.

Fixpoint supprT (t : T) : T :=
  match t with
    | E => E
    | C x => t
    | F x y =>
       match supprT x with
       | E => supprT y
       | xx =>
          match supprT y with
```

```
          | E => xx
          | yy => F xx yy
          end
      end
  end
.

Inductive U : Set :=
  | Ev : U
  | Cv : nat -> U
  | Fv : Ulist -> U
with Ulist : Set :=
  | Cons : U -> Ulist -> Ulist
  | Nil : Ulist
.

Fixpoint UofT (t : T) : U :=
  match t with
    | E => Ev
    | C x => Cv x
    | F x y => Fv (Cons (UofT x) (UofT2 y))
  end
with UofT2 (t : T) : Ulist :=
  match t with
  | F x y => Cons (UofT x) (UofT2 y)
  | E => Cons Ev Nil
  | C x => Cons (Cv x) Nil
  end
.

Fixpoint aplat_list (us tl : Ulist) {struct us} : Ulist :=
  match us with
  | Nil => tl
  | Cons x us' => aplat2 x (aplat_list us' tl)
  end
with aplat2 (x : U) (us : Ulist) {struct x} : Ulist :=
  match x with
  | Ev => Cons x us
  | Cv _ => Cons x us
  | Fv us1 => aplat_list us1 us
  end
.

Definition aplat (u : U) : U :=
```

```
  match u with
  | Ev => Ev
  | Cv x => Cv x
  | Fv us => Fv (aplat_list us Nil)
  end
.

Definition supprCons u us :=
  match u with
  | Ev => us
  | _ => Cons u us
  end
.

Fixpoint supprU_list (us : Ulist) : U :=
  match us with
  | Nil => Ev
  | Cons x tl =>
     match (supprU_list tl) with
     | Ev => supprU x
     | Cv n =>
        match supprU x with
        | Ev => Cv n
        | y => Fv (Cons y (Cons (Cv n) Nil))
        end
     | Fv tl2 =>
        match supprU x with
        | Ev => Fv tl2
        | y => Fv (Cons y tl2)
        end
     end
  end
with supprU (u : U) : U :=
  match u with
  | Ev => Ev
  | Cv x => Cv x
  | Fv us => supprU_list us
  end
.

Lemma l1 : forall t l,
  aplat_list (UofT2 t) l = aplat2 (UofT t) l.
Proof.
induction t; simpl; auto.
```

```
Qed.

Lemma l4 : forall t1 t2,
   aplat2 (UofT t1) (UofT2 t2) = UofT2 (peigne2 t1 t2).
Proof.
induction t1; simpl; auto.
intros.
assert (forall b,
  aplat_list (UofT2 t1_2) (UofT2 b) = UofT2 (peigne2 t1_2 b)).
Focus 2.
rewrite H.
rewrite IHt1_1.
auto.
intro.
rewrite <- IHt1_2.
apply l1.
Qed.

Lemma l3 : forall t b,
   aplat_list (UofT2 t) (UofT2 b) = UofT2 (peigne2 t b).
Proof.
induction t; simpl; auto.
intro.
simpl.
rewrite IHt2.
apply l4.
Qed.

Lemma l2 : forall t a b,
  a = UofT2 b ->
  Fv (aplat2 (UofT t) a) = UofT (peigne2 t b).
Proof.
induction t; simpl; intros; try rewrite H; auto.
rewrite (IHt1 (aplat_list (UofT2 t2) (UofT2 b)) (peigne2 t2 b)).
auto.
apply l3.
Qed.

Lemma l6 : forall t,
     aplat2 (UofT t) Nil = UofT2 (peigne t).
Proof.
induction t; simpl; auto.
rewrite l1.
rewrite IHt2.
```

```
apply l4.
Qed.


Theorem comm1 : forall t : T, aplat (UofT t) = UofT (peigne t).
Proof .
induction t; simpl; auto.
rewrite l1.
apply l2.
apply l6.
Qed.

Lemma l7 : forall t,
  supprU_list (UofT2 t) = supprU (UofT t).
Proof.
destruct t; simpl; auto.
Qed.

Lemma l8 : forall t1 t2,
  UofT (F t1 t2) = Fv (UofT2 (F t1 t2))
.
Proof.
simpl.
auto.
Qed.

Lemma l11 : forall x,
  supprU (UofT (F x E)) = supprU (UofT x).
Proof.
intro.
simpl.
destruct (supprU (UofT x)); simpl; auto.
Qed.

Lemma l12 : forall x y,
  supprU (UofT x) = Ev ->
  supprU (UofT y) = Ev ->
  supprU (UofT (F x y)) = Ev
.
Proof.
intros.
simpl.
rewrite H.
rewrite l7.
```

```
rewrite H0.
auto.
Qed.

Lemma l10 : forall x,
  supprT x = E -> supprU (UofT x) = Ev.
Proof.
induction x.
simpl; intro; auto.
simpl; intro; congruence.
intro.
simpl in H.
destruct x1.
simpl.
rewrite l7.
auto.
destruct x2.
simpl in H.
rewrite IHx1.
auto.
simpl; auto.
simpl in H.
congruence.
destruct (supprT (F x2_1 x2_2)).
simpl in H.
rewrite IHx2.
auto.
simpl; auto.
destruct (supprT E); congruence.
destruct (supprT E); congruence.
destruct (supprT (C n)).
simpl in IHx1.
absurd (Cv n = Ev); auto.
congruence.
destruct (supprT x2); congruence.
destruct (supprT x2); congruence.
apply l12.
apply IHx1.
destruct (supprT (F x1_1 x1_2)); auto.
destruct (supprT x2); congruence.
destruct (supprT x2); congruence.
destruct (supprT (F x1_1 x1_2)); try congruence.
apply IHx2; auto.
destruct (supprT x2); congruence.
```

```
destruct (supprT x2); congruence.
Qed.

Lemma l14 : forall x y,
  supprT x = E ->
  supprT y = E ->
  supprT (F x y) = E
.
Proof.
intros.
simpl.
rewrite H.
rewrite H0.
destruct x; auto.
Qed.

Lemma l13 : forall x,
  supprU (UofT x) = Ev -> supprT x = E.
Proof.
induction x.
simpl.
congruence.
simpl.
congruence.
intro.
simpl in H.
destruct (supprU (UofT x1)).
rewrite l7 in H.
apply l14.
auto.
auto.
simpl in H.
destruct (supprU (UofT x2)); auto; congruence.
destruct (supprU_list (UofT2 x2)); congruence.
destruct (supprU_list (UofT2 x2)); congruence.
Qed.

Lemma l9 : forall t,
    supprU_list (UofT2 t) = Ev -> supprT t = E.
Proof.
intros.
apply l13.
rewrite <- l7; auto.
Qed.
```

```
Lemma l18 : forall x,
  supprU (UofT (F E x)) = supprU (UofT x).
simpl.
intros; rewrite l7.
destruct (supprU (UofT x)); auto.
Qed.

Lemma l17 : forall x,
    supprT x = E -> supprU (UofT x) = Ev.
Proof.
induction x.
simpl.
auto.
simpl.
congruence.
intro.
simpl in H.
destruct x1.
rewrite l18.
auto.
destruct (supprT x2); simpl in H; try congruence.
apply l12.
apply IHx1.
destruct (supprT (F x1_1 x1_2)); simpl; try congruence.
destruct (supprT x2); congruence.
destruct (supprT x2); congruence.
apply IHx2.
destruct (supprT (F x1_1 x1_2)).
auto.
destruct (supprT x2); congruence.
destruct (supprT x2); congruence.
Qed.

Lemma l16 : forall t,
  supprT t = E -> supprU_list (UofT2 t) = Ev.
Proof.
intros t; rewrite l7; apply l17.
Qed.

Lemma l22 : forall t1 t2,
  supprT t1 = E -> supprT (F t1 t2) = supprT t2.
Proof.
intros.
```

```
destruct t1.
simpl; auto.
simpl in H; congruence.
destruct t2; simpl; simpl in H; rewrite H; auto.
Qed.

Lemma l24 : forall t1 t2,
  F t1 t2 = t1 -> False.
Proof.
induction t1.
congruence.
congruence.
intros.
apply IHt1_1 with t1_2.
congruence.
Qed.

Lemma l23 : forall t1 t2,
  supprT (F t1 t2) = supprT t1 -> supprT t2 = E
.
Proof.
simpl.
intros.
destruct (supprT t1).
auto.
destruct (supprT t2); congruence.
destruct (supprT t2); try congruence.
elimtype False.
generalize H.
apply l24.
elimtype False.
generalize H.
apply l24.
Qed.

Lemma l25 : forall t n,
  supprU (UofT t) = Cv n -> supprT t = C n.
Proof.
induction t; try (simpl; congruence).
intros.
simpl in H.
rewrite l7 in H.
caseq (supprU (UofT t2)); intros.
rewrite H0 in H.
```

```
simpl.
rewrite (IHt1 n); auto.
rewrite l13; auto.
rewrite H0 in H.
caseq (supprU (UofT t1)); intros.
rewrite H1 in H.
simpl.
rewrite (IHt2 n); auto; try congruence.
rewrite l13; auto.
rewrite H1 in H.
congruence.
rewrite H1 in H.
congruence.
rewrite H0 in H.
destruct (supprU (UofT t1)); congruence.
Qed.

Lemma l27 : forall t1 t2 n,
  supprT (F t1 t2) = C n ->
  supprT t1 = C n /\ supprT t2 = E \/
  supprT t2 = C n /\ supprT t1 = E.
Proof.
intros.
simpl in H.
destruct (supprT t1).
right; auto.
destruct (supprT t2).
left; auto.
congruence.
congruence.
destruct (supprT t2); congruence.
Qed.

Lemma l28 : forall t1 t2,
  supprT t2 = E -> supprT (F t1 t2) = supprT t1.
Proof.
intros.
simpl.
rewrite H.
destruct (supprT t1); congruence.
Qed.

Lemma l29 : forall t n,
  supprT t = C n -> supprU (UofT t) = Cv n.
```

```
Proof.
induction t; try (simpl; intros; congruence).
intros.
simpl.
elim (l27 _ _ _ H); intros.
elim H0; intros.
rewrite (IHt1 n).
rewrite l16.
simpl; auto.
auto.
auto.
elim H0; intros.
rewrite l17.
rewrite l7.
rewrite (IHt2 n); auto.
auto.
Qed.

Lemma l32 : forall t u,
  UofT t = Fv u -> UofT2 t = u.
Proof.
intro t.
caseq t; simpl; congruence.
Qed.

Lemma l34 : forall t1 t2 t3 t4,
  supprT (F t1 t2) = F t3 t4 ->
     supprT t1 = F t3 t4 /\ supprT t2 = E
  \/ supprT t1 = E /\ supprT t2 = F t3 t4
  \/ supprT t1 = t3 /\ supprT t2 = t4
.
Proof.
intros.
simpl in H.
destruct (supprT t1).
right; left; auto.
destruct (supprT t2).
congruence.
right; right; split; congruence.
right; right; split; congruence.
destruct (supprT t2).
left; auto.
right; right; split; congruence.
right; right; split; congruence.
```

```
Qed.

Lemma l35 : forall t1 t2,
  t1 = F t1 t2 -> False.
Proof.
induction t1.
congruence.
congruence.
intros.
apply (IHt1_1 t1_2).
congruence.
Qed.

Lemma l36 : forall t1 t2,
  t1 = F t2 t1 -> False.
Proof.
induction t1.
congruence.
congruence.
intros.
apply (IHt1_2 t1_1).
congruence.
Qed.

Lemma l37 : forall t1 t2,
  supprT t1 = F E t2 -> False.
Proof.
induction t1.
intro t2.
simpl.
congruence.
simpl.
congruence.
simpl.
intros.
destruct (supprT t1_1);
  destruct (supprT t1_2);
  try congruence.
apply (IHt1_2 t2); auto.
apply (IHt1_1 t2); auto.
Qed.

Lemma l38 : forall t1 t2,
  supprT t1 = F t2 E -> False.
```

```
Proof.
induction t1.
intro t2.
simpl.
congruence.
simpl.
congruence.
simpl.
intros.
destruct (supprT t1_1);
  destruct (supprT t1_2);
  try congruence.
apply (IHt1_2 t2); auto.
apply (IHt1_1 t2); auto.
Qed.

Lemma l39 : forall x : T,
  match x with E => E | _ => x end = x.
Proof.
intro.
destruct x; congruence.
Qed.

Lemma l30 : forall t1 t2 t3,
  supprT t1 = F t2 t3 ->
  supprU (UofT t1) = Fv (Cons (UofT t2) (UofT2 t3)).
Proof.
induction t1.
intros.
simpl in H.
congruence.
intros.
simpl in H.
congruence.
intros.
elim (l34 _ _ _ _ H); intros.
elim H0; intros.
rewrite <- IHt1_1; auto.
simpl.
rewrite l16; auto.
elim H0; intros.
elim H1; intros.
rewrite <- IHt1_2; auto.
simpl.
```

```
rewrite l17; auto.
rewrite l7.
destruct (supprU (UofT t1_2)); congruence.
elim H1; intros.
destruct t2.
elimtype False.
apply (l37 _ _ H).
simpl.
rewrite (l29 t1_1 n); auto.
destruct t3.
simpl in H.
rewrite H2 in H.
rewrite H3 in H.
congruence.
rewrite l7.
rewrite (l29 t1_2 n0).
simpl.
auto.
auto.
rewrite l7.
rewrite (IHt1_2 t3_1 t3_2).
simpl.
auto.
auto.
simpl.
rewrite (IHt1_1 t2_1 t2_2).
rewrite l7.
destruct t3.
elimtype False; apply (l38 _ _ H).
rewrite (l29 _ _ H3).
simpl.
auto.
rewrite (IHt1_2 _ _ H3).
simpl.
auto.
auto.
Qed.
```

Theorem `comm2 : forall t : T,`
`supprU (UofT t) = UofT (supprT t).`

```
Proof.
induction t; simpl; auto.
rewrite IHt1.
destruct t1.
```

```
simpl.
rewrite l7.
auto.
simpl.
rewrite IHt2.
destruct (UofT (supprT t2)); congruence.
simpl.
rewrite l7.
rewrite IHt2.
destruct (supprT t2); try (simpl; auto).
rewrite l7.
rewrite IHt2.
destruct (supprT t2).
rewrite l39.
simpl; auto.
simpl.
destruct (supprT t1_1); destruct (supprT t1_2); simpl; congruence.
simpl.
destruct (supprT t1_1); destruct (supprT t1_2); simpl; congruence.
Qed.

Inductive equa : T -> T -> Prop :=
  | refl : forall t : T, equa t t
  | sym : forall t1 t2 : T,
      equa t1 t2 -> equa t2 t1
  | trans : forall t2 t1 t3 : T,
      equa t1 t2 -> equa t2 t3 -> equa t1 t3
  | subst : forall t1 t2 t3 t4,
      equa t1 t3 -> equa t2 t4 -> equa (F t1 t2) (F t3 t4)
  | assoc : forall t1 t2 t3 : T,
      equa (F (F t1 t2) t3) (F t1 (F t2 t3))
  | neutre_l : forall t : T, equa (F E t) t
  | neutre_r : forall t : T, equa (F t E) t
.

Hint Constructors equa.

Lemma l100 : forall t1 t2,
  equa (F t1 t2) (peigne2 t1 t2).
Proof.
induction t1; auto.
simpl.
intro.
apply (trans (F t1_1 (F t1_2 t2))).
```

```
auto.
apply (trans (F t1_1 (peigne2 t1_2 t2))).
auto.
auto.
Qed.

Theorem correct_peigne : forall t : T, equa t (peigne t).
Proof.
induction t; auto.
simpl.
apply (trans (F t1 (peigne t2))).
auto.
apply l100.
Qed.

Theorem correct_supprT : forall t : T, equa t (supprT t).
Proof.
induction t; auto.
simpl.
destruct (supprT t1).
apply (trans (F E t2)).
auto.
apply (trans t2); auto.
destruct (supprT t2).
apply (trans (F t1 E)).
auto.
apply (trans t1); auto.
auto.
auto.
destruct (supprT t2).
apply (trans (F t1 E)).
auto.
apply (trans t1); auto.
auto.
auto.
Qed.

Inductive fe_tree : T -> Prop :=
  | fe_e : fe_tree E
  | fe_f : forall t1 t2 : T,
      fe_tree t1 -> fe_tree t2 -> fe_tree (F t1 t2)
.

Hint Constructors fe_tree.
```

```
Definition fe_pred := fun t t0 =>
  fe_tree t -> fe_tree t0
.

Lemma l102 : forall n,
  fe_tree (C n) -> False
.
Proof.
intros.
inversion H.
Qed.

Lemma l101 : forall t1 t2,
  equa t1 t2 -> (fe_tree t1 <-> fe_tree t2).
Proof.
intros.
apply (equa_ind (fun t t0 => fe_tree t <-> fe_tree t0)); auto.
intros; split; auto.
intros; elim H1; intros; split; auto.
intros; elim H1; elim H3; intros; split; auto.
intros; elim H1; elim H3; intros; split;
  intros; apply fe_f; inversion H8; auto.
intros; split; intros.
inversion H0; inversion H3; apply fe_f; auto; apply fe_f; auto.
inversion H0; inversion H4; apply fe_f; auto; apply fe_f; auto.
intros; split; intros.
inversion H0; auto.
apply fe_f; auto; apply fe_e; auto.
intros; split; intros.
inversion H0; auto.
apply fe_f; auto; apply fe_e; auto.
Qed.

Lemma l104 : forall t1 t2,
  equa t1 t2 -> fe_tree t1 -> fe_tree t2.
Proof.
intros.
elim (l101 t1 t2); intros; auto.
Qed.

Lemma l105 : forall t1 t2,
  equa t1 t2 -> fe_tree t2 -> fe_tree t1.
Proof.
```

```
intros.
elim (l101 t1 t2); intros; auto.
Qed.

Lemma l103 : forall n,
  equa E (C n) -> False.
Proof.
intros.
assert (fe_tree (C n)).
elim (l101 E (C n)); intros.
apply H0.
apply fe_e.
auto.
apply (l102 n).
auto.
Qed.

Lemma l107 : forall t1 t2,
  fe_tree t1 -> fe_tree t2 -> fe_tree (peigne2 t1 t2).
Proof.
induction t1.
simpl; intros; auto.
simpl; intros; auto.
intros; simpl.
inversion H.
apply IHt1_1; auto.
Qed.

Lemma l108 : forall t,
  fe_tree t -> fe_tree (peigne t).
Proof.
induction t.
simpl; auto.
simpl; auto.
simpl.
intro; inversion H.
apply l107; auto.
Qed.

Lemma l106 : forall t,
  fe_tree t -> supprT t = E.
Proof.
induction t.
simpl; auto.
```

```
simpl.
intro; elimtype False; apply (l102 n); auto.
intro; inversion H.
simpl.
rewrite IHt1; auto.
Qed.

Lemma l110 : forall t a b c,
  supprT (match t with E => a | C _ => b | F _ _ => c end)
  = match t with E => supprT a
  | C _ => supprT b | F _ _ => supprT c end
.
Proof.
destruct t; simpl; auto.
Qed.

Inductive subterm : T -> T -> Prop :=
  | left : forall t1 t2 t3 : T, t1 = t2 -> subterm t1 (F t2 t3)
  | right : forall t1 t2 t3 : T, t1 = t2 -> subterm t1 (F t3 t2)
  | recleft : forall t1 t2 t3 : T,
      subterm t1 t2 -> subterm t1 (F t2 t3)
  | recright : forall t1 t2 t3 : T,
      subterm t1 t2 -> subterm t1 (F t3 t2)
.

Hint Constructors subterm.

Lemma l117 : forall t1 t2 t3,
  subterm (F t1 t2) t3 -> subterm t1 t3.
Proof.
induction t3.
intro.
inversion H; auto.
intro.
inversion H; auto.
intro.
inversion H; auto; rewrite <- H2; auto.
Qed.

Lemma l118 : forall t1 t2 t3,
  subterm (F t1 t2) t3 -> subterm t2 t3.
Proof.
induction t3.
intro.
```

```
inversion H; auto.
intro.
inversion H; auto.
intro.
inversion H; auto; rewrite <- H2; auto.
Qed.

Lemma l116 : forall t,
  subterm t (supprT t) -> False.
Proof.
intros.
induction t.
simpl in H; inversion H.
simpl in H; inversion H.
simpl in H.
caseq (supprT t1); intros; caseq (supprT t2); intros;
  rewrite H0 in H; rewrite H1 in H; simpl in H; try congruence.
inversion H.
inversion H.
apply IHt2.
rewrite H1.
apply l118 with t1; auto.
inversion H; auto.
inversion H; auto.
inversion H4; auto.
inversion H4; auto.
rewrite <- H1 in H.
apply IHt2.
inversion H; auto.
inversion H4; auto.
inversion H4; auto.
inversion H4; auto.
inversion H4; auto.
inversion H4; auto.
apply IHt2.
rewrite H1.
inversion H; auto.
congruence.
rewrite <- H4.
auto.
inversion H4; auto.
apply l118 with t1; auto.
apply IHt1.
rewrite <- H0 in H.
```

```
apply l117 with t2; auto.
rewrite <- H0 in H.
inversion H; auto.
apply IHt1.
rewrite <- H4; auto.
congruence.
apply IHt1.
apply l117 with t2; auto.
inversion H4; auto.
inversion H.
apply IHt1.
rewrite H0.
rewrite <- H4.
auto.
apply IHt2.
rewrite H1.
rewrite <- H4.
auto.
apply IHt1.
rewrite H0.
apply l117 with t2; auto.
apply IHt2.
rewrite H1.
apply l118 with t1; auto.
Qed.

Lemma l111 : forall t1 t2,
  supprT (F t1 t2) = F t1 t2 -> supprT t1 = E -> False.
Proof.
simpl.
intros.
destruct (supprT t1).
apply l116 with t2.
rewrite H; auto.
congruence.
congruence.
Qed.

Lemma l112 : forall t1 t2,
  supprT (F t1 t2) = F t1 t2 -> supprT t2 = E -> False.
Proof.
simpl.
intros.
destruct (supprT t2).
```

```
caseq (supprT t1); intros; rewrite H1 in H; try congruence.
rewrite H in H1.
apply l116 with t1.
rewrite H1.
auto.
congruence.
congruence.
Qed.

Lemma l113 : forall t1 t2,
  supprT (F t1 t2) = F t1 t2 -> supprT t1 = t1.
Proof.
simpl.
intros.
caseq (supprT t2); intros; rewrite H0 in H.
caseq (supprT t1); intros; rewrite H1 in H; try congruence.
rewrite H in H1.
elimtype False.
apply l116 with t1.
rewrite H1; auto.
destruct (supprT t1); try congruence.
destruct (supprT t1); try congruence.
rewrite H in H0.
elimtype False.
apply l116 with t2.
rewrite H0; auto.
Qed.

Lemma l114 : forall t1 t2,
  supprT (F t1 t2) = F t1 t2 -> supprT t2 = t2.
Proof.
simpl.
intros.
caseq (supprT t2); intros; rewrite H0 in H.
caseq (supprT t1); intros; rewrite H1 in H; try congruence.
rewrite H in H1.
elimtype False.
apply l116 with t1.
rewrite H1; auto.
destruct (supprT t1); try congruence.
destruct (supprT t1); try congruence.
rewrite H in H0.
elimtype False.
apply l116 with t2.
```

```
rewrite H0; auto.
Qed.

Lemma l120 : forall t1 t2,
  t1 <> E ->
  supprT t1 = t1 ->
  t2 <> E ->
  supprT t2 = t2 ->
  supprT (F t1 t2) = F t1 t2
.
Proof.
simpl.
intros.
destruct (supprT t1); destruct (supprT t2);
    simpl; try congruence.
Qed.

Lemma l119 : forall t,
  supprT (supprT t) = supprT t.
Proof.
induction t; simpl; auto.
rewrite l110.
rewrite l110.
rewrite IHt1.
rewrite IHt2.
destruct (supprT t1); destruct (supprT t2); auto;
  try (rewrite (l120 (supprT t1) (supprT t2)); auto).
apply l120; try congruence.
apply l120; try congruence.
apply l120; try congruence.
Qed.

Lemma l109 : forall t1 t2,
  supprT (peigne2 t1 (supprT t2)) = supprT (peigne2 t1 t2).
Proof.
induction t1.
simpl; auto.
induction t2.
simpl; auto.
simpl; auto.
simpl.
rewrite l110.
rewrite IHt2_2.
rewrite l110.
```

```
rewrite IHt2_1.
destruct (supprT t2_1); destruct (supprT t2_2); simpl; auto.
rewrite (l113 t1 t2); auto.
rewrite (l114 t1 t2); auto.
destruct t1; destruct t2; simpl; try (simpl in IHt2_2; congruence).
elimtype False.
apply (l37 _ _ IHt2_2).
elimtype False.
apply (l38 _ _ IHt2_2).
rewrite (l113 t1 t2); auto.
rewrite (l114 t1 t2); auto.
destruct t1; destruct t2; simpl; try (simpl in IHt2_1; congruence).
elimtype False; apply (l37 _ _ IHt2_1).
elimtype False; apply (l38 _ _ IHt2_1).
rewrite (l113 t1 t2); auto.
rewrite (l114 t1 t2); auto.
rewrite (l113 t3 t4); auto.
rewrite (l114 t3 t4); auto.
destruct t1; destruct t2; destruct t3; destruct t4; simpl;
  try (simpl in IHt2_2; congruence); try (simpl in IHt2_1; congruence);
  try (elimtype False; apply (l37 _ _ IHt2_1));
  try (elimtype False; apply (l38 _ _ IHt2_1));
  try (elimtype False; apply (l37 _ _ IHt2_2));
  try (elimtype False; apply (l38 _ _ IHt2_2));
idtac.
simpl.
intro; rewrite l119; auto.
intro; simpl.
rewrite <- IHt1_1.
rewrite IHt1_2.
rewrite IHt1_1.
auto.
Qed.

Lemma l122 : forall t1 t2 t3,
  peigne2 t1 (peigne2 t2 t3) = peigne2 (F t1 t2) t3.
Proof.
destruct t1; simpl; auto.
Qed.

Lemma l123 : forall t1 t2,
  supprT (peigne2 (F t1 E) t2) = supprT (peigne2 t1 t2).
Proof.
induction t1.
```

```
simpl; auto.
simpl; auto.
simpl.
intro.
rewrite <- l109.
rewrite <- (l109 t1_2 (F E t2)).
simpl.
rewrite (l109 t1_2 t2).
rewrite l109.
auto.
Qed.
```

Lemma `l121 : forall t1 t2,`
`  supprT (peigne2 (supprT t1) t2) = supprT (peigne2 t1 t2).`

```
Proof.
induction t1.
simpl; auto.
simpl; auto.
intro.
simpl.
rewrite <- IHt1_1.
rewrite <- (l109 (supprT t1_1) (peigne2 t1_2 t2)).
rewrite <- IHt1_2.
rewrite l109.
rewrite l122.
destruct (supprT t1_1).
simpl; auto.
destruct (supprT t1_2).
simpl; auto.
simpl; auto.
auto.
destruct (supprT t1_2); auto.
simpl.
rewrite <- (l109 t1 (peigne2 t3 (F E t2))).
rewrite <- (l109 t3).
simpl.
rewrite (l109 t3).
rewrite l109.
auto.
Qed.
```

Lemma `l125 : forall t1 t2 t3,`
`  peigne2 t1 (peigne2 t2 t3) = peigne2 (peigne2 t1 t2) t3.`

```
Proof.
```

```
induction t1.
simpl; auto.
simpl; auto.
simpl.
intros.
rewrite IHt1_2.
rewrite IHt1_1.
auto.
Qed.
```

Lemma `l124: forall t1 t2,`
```
  peigne2 t1 t2 = peigne2 (peigne t1) t2.
Proof.
induction t1; simpl; auto.
intro.
rewrite IHt1_2.
apply l125.
Qed.
```

Theorem `completT :`
```
  forall t1 t2 : T,
    equa t1 t2 -> supprT (peigne t1) = supprT (peigne t2).
Proof.
intros.
induction H; auto.
congruence.
simpl.
rewrite <- l109.
rewrite IHequa2.
rewrite l109.
rewrite l124.
rewrite <- l121.
rewrite IHequa1.
rewrite l121.
rewrite <- l124.
auto.
simpl; auto.
induction t; simpl; auto.
rewrite <- l109.
rewrite IHt2.
rewrite l109.
auto.
Qed.
```

```
Theorem completU : forall t1 t2 : T,
  equa t1 t2 -> supprU (aplat (UofT t1)) = supprU (aplat (UofT t2)).
Proof.
intros.
rewrite comm1.
rewrite comm1.
rewrite comm2.
rewrite comm2.
rewrite (completT t1 t2 H); auto.
Qed.

Theorem correct_aplat :
  forall t1 : T, exists t2 : T,
    aplat (UofT t1) = UofT t2 /\ equa t1 t2.
Proof.
intros.
rewrite comm1.
exists (peigne t1).
split; auto.
apply correct_peigne.
Qed.

Theorem correct_supprU :
  forall t1 : T, exists t2 : T,
    supprU (UofT t1) = UofT t2 /\ equa t1 t2.
Proof.
intros.
rewrite comm2.
exists (supprT t1).
split; auto.
apply correct_supprT.
Qed.

Theorem inj2 : forall t1 t2,
  (UofT t1 = UofT t2 -> t1 = t2) /\ (UofT2 t1 = UofT2 t2 -> t1 = t2).
Proof.
induction t1; induction t2.
  simpl; auto.
  simpl; split; intro; discriminate.
  simpl; split.
    intro; discriminate.
    intro.
    injection H.
    intros H1 H2.
```

```
    destruct t2_2; simpl in H1; discriminate.
  intuition; discriminate.
  simpl; intuition; congruence.
  simpl; split; intro H; try (discriminate).
    injection H.
    intro H1.
    destruct t2_2; simpl in H1; discriminate.
  simpl; split; intro H; try (discriminate).
    injection H.
    intro H1.
    destruct t1_2; simpl in H1; discriminate.
  simpl; split; intro H; try (discriminate).
    injection H.
    intro H1.
    destruct t1_2; simpl in H1; discriminate.
split.
  simpl; intro H.
  injection H.
  clear H.
  intros H1 H2.
  generalize (IHt1_1 t2_1).
  intros (H3, H4).
  generalize (H3 H2).
  intro Heq1.
  clear H3 H4.
  generalize (IHt1_2 t2_2).
  intros (H3, H4).
  generalize (H4 H1).
  intro Heq2.
  rewrite Heq1; rewrite Heq2; reflexivity.

  simpl; intro H.
  injection H.
  clear H.
  intros H1 H2.
  generalize (IHt1_1 t2_1).
  intros (H3, H4).
  generalize (H3 H2).
  intro Heq1.
  clear H3 H4.
  generalize (IHt1_2 t2_2).
  intros (H3, H4).
  generalize (H4 H1).
  intro Heq2.
```

```
  rewrite Heq1; rewrite Heq2; reflexivity.
Qed.

(* Il faut rajouter l'injectivite de UofT, sinon les theoremes de
correction sur U ne sont pas tres interessants *)
Theorem injection : forall t1 t2,
  UofT t1 = UofT t2 -> t1 = t2.
Proof.
intros t1 t2.
exact (proj1 (inj2 t1 t2)).
Qed.

Lemma compat_eq : forall t1 t2,
  t1 = t2 -> equa t1 t2.
Proof.
intros.
induction t1; rewrite H; auto.
Qed.

(* Compatibilite de UofT avec equa : *)
Theorem compatible : forall t1 t2,
  UofT t1 = UofT t2 -> equa t1 t2
.
intros.
apply compat_eq.
apply injection.
exact H.
Qed.
```

Bibliographie

[All78] John Allen – *Anatomy of LISP*, McGraw-Hill, Inc., New York, NY, USA, 1978. 67

[BBK+06] Émilie Balland, Paul Brauner, Radu Kopetz, Pierre-Étienne Moreau et Antoine Reilles – « The Tom manual », 2006, `http://tom.loria.fr/soft/release-2.4/manual-2.4/index.html`. 17, 75

[BDL06] Sandrine Blazy, Zaynah Dargaye et Xavier Leroy – « Formal verification of a C compiler front-end », *Formal Methods 2006*, Lecture Notes in Computer Science, Springer-Verlag, 2006. 33, 34

[BHKO02a] Mark van den Brand, Jan Heering, Paul Klint et Peter Olivier – « Compiling language definitions : the ASF+SDF compiler », *ACM Trans. Program. Lang. Syst.* **24** (2002), no. 4, p. 334–368. 15

[BHKO02b] Mark van den Brand, Jan Heering, Paul Klint et Pieter Olivier – « Compiling language definitions : The ASF+SDF compiler », *ACM Transactions on Programming Languages and Systems* **24** (2002), no. 4, p. 334–368. 14, 74

[BJKO00] Mark van den Brand, Hayco de Jong, Paul Klint et Pieter Olivier – « Efficient annotated terms », *Software, Practice and Experience* **30** (2000), no. 3, p. 259–291. 20, 29, 66

[BKK+98] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Étienne Moreau et Christophe Ringeissen – « An overview of ELAN », *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)* (Pont-à-Mousson, France) (C. Kirchner et H. Kirchner, éds.), vol. 15, Electronic Notes in Theoretical Computer Science, Septembre 1998. 116

[BKM06] Émilie Balland, Claude Kirchner et Pierre-Étienne Moreau – « Formal islands », *Proceedings of the 11th International Conference on Algebraic Methodology and Software Technology (AMAST'06)* (Kuressaare, Estonia), 2006. 35, 36

[BKO99] Mark van den Brand, Paul Klint et Pieter Olivier – « Compilation and memory management for asf+sdf », *Proceedings of the 8th International Conference on Compiler Construction (CC'99)* (London, UK), Springer-Verlag, 1999, p. 198–213. 83

[BKV03] Mark van den Brand, Paul Klint et Jurgen Vinju – « Term rewriting with traversal functions », *ACM Trans. Softw. Eng. Methodol.* **12** (2003), no. 2, p. 152–190. 31, 119

[BLC02] Éric Bruneton, Romain Lenglet et Thierry Coupaye – « ASM : a code manipulation tool to implement adaptable systems », *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)*, Novembre 2002. 144

[BM06] Émilie Balland et Pierre-Étienne Moreau – « Optimizing pattern matching compilation by program transformation », *3rd Workshop on Software Evolution through Transformations (SeTra'06)* (J.-M. Favre, R. Heckel et T. Mens, éds.), Electronic Communications of EASST, 2006, To appear. 156

[BMV05] Mark van den Brand, Pierre-Étienne Moreau et Jurgen Vinju – « A generator of efficient strongly typed abstract syntax trees in java », *IEE Proceedings - Software Engineering* **152** (2005), no. 2, p. 70–78. 29, 68

[Bor95] Peter Borovanský – « Implementation of higher-order unification based on calculus of explicit substitutions », *Proceedings of the 22nd Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM'95)* (London, UK) (M. Bartošek, J. Staudek et J. Wiedermann, éds.), Lecture Notes in Computer Science, vol. 1012, Springer-Verlag, 1995, p. 363–368. 15

[Bor98] — , « Le contrôle de la réécriture : étude et implantation d'un formalisme de stratégies », Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, Octobre 1998. 15, 116, 117, 123

[BSV97] Mark van den Brand, Alex Sellink et Chris Verhoef – « Generation of components for software renovation factories from context-free grammars », *Proceedings of the 4th Working Conference on Reverse Engineering* (I. Baxter, A. Quilici et C. Verhoef, éds.), IEEE Computer Society Press Los Alamitos California, 1997. 15

[BSV98] — , « Control flow normalization for COBOL/CICS legacy system », *Proceedings : 2nd Euromicro Conference on Software Maintenance and Reengineering* (P. Nesi et F. Lehner, éds.), IEEE Computer Society Press, 1998, p. 11–20. 15

[BV96] Mark van den Brand et Eelco Visser – « Generation of formatters for context-free languages », *ACM Trans. Softw. Eng. Methodol* **5** (1996), no. 1, p. 1–41. 15

[BVVV05] Martin Bravenboer, Rob Vermaas, Jurgen Vinju et Eelco Visser – « Generalized type-based disambiguation of meta programs with concrete object syntax », *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering (GPCE'05)* (Tallinn, Estonia) (R. Glück et M. Lowry, éds.), Lecture Notes in Computer Science, vol. 3676, Springer-Verlag, Septembre 2005, p. 157–172. 154

[BWH06] Frédéric Blanqui, Pierre Weis et Thérèse Hardin – « An extension of ML

with automatic preservation of data invariants », Juillet 2006, Submitted. 87

[BY92] Robert S. Boyer et Yuan Yu – « Automated correctness proofs of machine code programs for a commercial microprocessor », *Proceedings of the 11th International Conference on Automated Deduction* (D. Kapur, éd.), Springer-Verlag, 1992, p. 416–430. 35

[Cas98] Carlos Castro – « Une approche déductive de la résolution de problèmes de satisfaction de contraintes », Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, France, 1998. 15

[CC77] Patrick Cousot et Radhia Cousot – « Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints », *Conference Records of the 4th ACM Symposium on Principles of Programming Languages (POPL'77)* (Los Angeles, CA, USA), Janvier 1977, p. 238–252. 49

[Cir01] Horatiu Cirstea – « Specifying Authentication Protocols Using Rewriting and Strategies », *3rd International Symposium on Practical Aspects of Declarative Languages* (Las Vegas, NV, USA), Lecture Notes in Computer Science, vol. 1990, Mars 2001, p. 138–153. 15

[CJ03] Hubert Comon et Jean-Pierre Jouannaud – « Les termes en logique et en programmation », 2003, Master lectures at Univ. Paris Sud. 75

[CK97] Horatiu Cirstea et Claude Kirchner – « Theorem Proving Using Computational Systems : The Case of the B Predicate Prover », *Workshop CCL'97* (Schloß Dagstuhl, Germany), Septembre 1997. 15

[CKL01] Horatiu Cirstea, Claude Kirchner et Luigi Liquori – « Matching Power », *Proceedings of the 12th Conference on Rewriting Techniques and Applications (RTA'01)* (Utrecht, The Netherlands) (A. Middeldorp, éd.), Lecture Notes in Computer Science, vol. 2051, Springer-Verlag, Mai 2001, p. 77–92. viii

[CM00] Manuel Clavel et Jos'e Meseguer – « Reflection and strategies in rewriting logic », *Electronic Notes in Theoretical Computer Science* (J. Meseguer, éd.), vol. 4, Elsevier Science Publishers, 2000. 14

[CMR04] Horatiu Cirstea, Pierre-Étienne Moreau et Antoine Reilles – « Rule based programming in Java for protocol verification », *Proceedings of the 5th International Workshop on Rewriting Logic and its Applications* (N. Marti-Oliet, éd.), vol. 117, Electronic Notes in Theoretical Computer Science, Avril 2004, p. 209–227. 81

[Dau89] Max Dauchet – « Simulation of Turing machines by a left-linear rewrite rule », *Proceedings 3rd Conference on Rewriting Techniques and Applications* (N. Dershowitz, éd.), Lecture Notes in Computer Science, vol. 355, Springer-Verlag, Avril 1989, p. 109–120. 13

[DF01] Olivier Dubuisson et Philippe Fouquart – *ASN.1 : communication between heterogeneous systems*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. 72

[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/`. 53

[ecl01] « Eclipse platform technical overview » – Tech. report, Object Technology International, Inc., 2001. 158

[GHJV93] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides – « Design patterns : Abstraction and reuse of object-oriented design », *Lecture Notes in Computer Science* **707** (1993), p. 406–431. 66

[GM01] Jean-Louis Giavitto et Olivier Michel – « MGS : a rule-based programming language for complex objects and collections », *Electronic Notes in Theoretical Computer Science* (M. van den Brand et R. Verma, éds.), vol. 59, Elsevier Science Publishers, 2001. 134

[GMR04] Julien Guyon, Pierre-Étienne Moreau et Antoine Reilles – « An integrated development environment for pattern matching programming », *Proceedings of the 2nd eclipse Technology eXchange workshop (eTX'2004)* (B. Barry et O. de Moor, éds.), Electronic Notes in Theoretical Computer Science, Avril 2004. 158

[GN04] Sumit Gulwani et George C. Necula – « Global value numbering using random interpretation », *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'04)*, ACM Press, 2004, p. 342–352. 49

[Gog78] Joseph Goguen – « Abstract errors for abstract data types », p. 491–526, North-Holland, 1978. 14

[Gou94] Jean Goubault – « HimML : Standard ML with fast sets and maps », 1994. 67

[Gug02] Alessio Guglielmi – « A system of interaction and structure », Tech. Report WV-02-10, TU Dresden, 2002, To app. in ACM Transactions on Computational Logic. 88

[GWM+93] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi et Jean-Pierre Jouannaud – « Introducing OBJ », Applications of Algebraic Specification using OBJ (J. Goguen, éd.), Cambridge, 1993. 14

[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** (1992), no. 1, p. 29–51. 71

[JO04] Hayco de Jong et Pieter Olivier – « Generation of abstract programming interfaces from syntax definitions », *Journal of Logic and Algebraic Programming* **59** (2004), no. 1-2, p. 35–61. 68, 86

[Kah87] Gilles KAHN – « Natural semantics », *4th Annual Symposium on Theoretical Aspects of Computer Sciences (STACS'87)* (London, UK), Springer-Verlag, 1987, p. 22–39. 45

[Kah04] Ozan KAHRAMANOĞULLARI – « Implementing system BV of the calculus of structures in maude », *Proceedings of the ESSLLI-2004 Student Session* (Université Henri Poincaré - Nancy I) (L. ALONSO I ALEMANY et P. ÉGRÉ, éds.), 2004, 16th European Summer School in Logic, Language and Information, p. 117–127. 90, 97

[Kah06] Ozan KAHRAMANOĞULLARI – « Reducing the non-determinism in the calculus of structures », *Proceedings of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, Lecture Notes in Computer Science, Springer-Verlag, 2006, Available at `http://www.doc.ic.ac.uk/~ozank/Papers/reducingNondet.pdf`. 97

[KKV95] Claude KIRCHNER, Hélène KIRCHNER et Marian VITTEK – « Designing Constraint Logic Programming Languages using Computational Systems », Principles and Practice of Constraint Programming. The Newport Papers. (P. VAN HENTENRYCK et V. SARASWAT, éds.), MIT press, 1995, p. 131–158. 12

[KLM+97] Gregor KICZALES, John LAMPING, Anurag MENHDHEKAR, Chris MAEDA, Cristina LOPES, Jean-Marc LOINGTIER et John IRWIN – « Aspect-oriented programming », Proceedings European Conference on Object-Oriented Programming (M. AKŞIT et S. MATSUOKA, éds.), vol. 1241, Springer-Verlag, Berlin, Heidelberg, and New York, 1997, p. 220–242. 121

[KLS96] Claude KIRCHNER, Christopher LYNCH et Christelle SCHARFF – « A fine-grained concurrent completion procedure », *Proceedings of the 7th Conference on Rewriting Techniques and Applications (RTA'96)* (H. GANZINGER, éd.), Lecture Notes in Computer Science, vol. 1103, Springer-Verlag, Septembre 1996, p. 3–17. 15

[KM95] Hélène KIRCHNER et Pierre-Étienne MOREAU – « Prototyping completion with constraints using computational systems », *Proceedings 6th Conference on Rewriting Techniques and Applications* (J. HSIANG, éd.), Lecture Notes in Computer Science, vol. 914, Springer-Verlag, 1995, p. 438–443. 15

[KMR05a] Ozan KAHRAMANOĞULLARI, Pierre-Étienne MOREAU et Antoine REILLES – « Implementing deep inference in TOM », *Structures and Deduction* (P. BRUSCOLI, F. LAMARCHE et C. STEWART, éds.), Technische Universität Dresden, Juillet 2005, ISSN 1430-211X, p. 158–172. x, 4, 88, 94, 97

[KMR05b] Claude KIRCHNER, Pierre-Étienne MOREAU et Antoine REILLES – « Formal validation of pattern matching code », *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming* (P. BARAHONE et A. FELTY, éds.), ACM Press, Juillet 2005, p. 187–197. ix, 3

[KV00] Tobias KUIPERS et Joost VISSER – « Object-oriented tree traversal with JJForester », Tech. report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 2000. 72

[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/`. 87

[Ler06] Xavier LEROY – « Formal certification of a compiler back-end, or : programming a compiler with a proof assistant », *33rd symposium Principles of Programming Languages*, ACM Press, 2006, p. 42–54. 33, 34

[LFM01] Fabrice LE FESSANT et Luc MARANGET – « Optimizing pattern matching », *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, ACM Press, 2001, p. 26–37. 40

[LJVWF02] David LACEY, Neil D. JONES, Eric VAN WYK et Carl Christian FREDERIKSEN – « Proving correctness of compiler optimizations by temporal logic », *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, ACM Press, 2002, p. 283–294. 33

[Low95] Gavin LOWE – « An attack on the Needham-Schroeder public key authentication protocol », *Information Processing Letters* **56** (1995), p. 131–133. 81

[Mar96] Claude MARCHÉ – « Normalized rewriting : an alternative to rewriting modulo a set of equations », *Journal of Symbolic Computation* **21** (1996), no. 3, p. 253–288. 71

[MK98] Pierre-Étienne MOREAU et Hélène KIRCHNER – « A compiler for rewrite programs in associative-commutative theories », *Principles of Declarative Programming*, Lecture Notes in Computer Science, no. 1490, Springer-Verlag, Septembre 1998, p. 230–249. 15

[Mor73] F. Lockwood MORRIS – « Advice on structuring compilers and proving them correct », *Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ACM Press, 1973, p. 144–152. 33

[Mor99] Pierre-Étienne MOREAU – « Compilation de règles de réécriture et de stratégies non-déterministes », Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, France, 1999. 116

[MP67] John MCCARTHY et James PAINTER – « Correctness of a compiler for arithmetic expressions », Proceedings Symposium in Applied Mathematics, Vol. 19 (J. T. SCHWARTZ, éd.), AMS, 1967, p. 33–41. 33

[MRV03] Pierre-Étienne MOREAU, Christophe RINGEISSEN et Marian VITTEK – « A Pattern Matching Compiler for Multiple Target Languages », *12th Conference on Compiler Construction* (G. HEDIN, éd.), Lecture Notes in Computer Science, vol. 2622, Springer-Verlag, Mai 2003, p. 61–76. viii, 2, 17

[MZ04] Pierre-Etienne Moreau et Olivier Zendra – « Gc^2 : A generational conservative garbage collector for the aterm library », *The Journal of Logic and Algebraic Programming (JLAP)* **59** (2004), no. 1-2, p. 5–34. 29

[Nec97] George C. Necula – « Proof-carrying code », *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, 1997, p. 106–119. 35

[Ngu02] Quang-Huy Nguyen – « Calcul de réécriture et automatisation du raisonnement dans les assistants de preuve », Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, Octobre 2002. 139

[NL98] George C. Necula et Peter Lee – « The design and implementation of a certifying compiler », *SIGPLAN Not.* **33** (1998), no. 5, p. 333–344. 35

[NS78] Roger Needham et Michael Schroeder – « Using encryption for authentication in large networks of computers », *Communications of the ACM* **21** (1978), no. 12, p. 993–999. 81

[ORW95] Dino P. Oliva, John D. Ramsdell et Mitchell Wand – « The VLISP verified PreScheme compiler », *Lisp Symb. Comput.* **8** (1995), no. 1-2, p. 111–182. 33

[OW97] Martin Odersky et Philip Wadler – « Pizza into java : translating theory into practice », *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'97)* (New York, NY, USA), ACM Press, 1997, p. 146–159. viii, 2, 72

[PQ95] Terence Parr et Russell Quong – « Antlr : a predicated-ll(k) parser generator », *Softw. Pract. Exper.* **25** (1995), no. 7, p. 789–810. 115

[PSS98] Amir Pnueli, Michael Siegel et Eli Singerman – « Translation validation », *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, Springer-Verlag, 1998, p. 151–166. 35

[Rei06] Antoine Reilles – « Canonical abstract syntax trees », *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science, 2006, to appear. x, 4

[Rin97] Christophe Ringeissen – « Prototyping Combination of Unification Algorithms with the ELAN Rule-Based Programming Language », *Proceedings 8th Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science, vol. 1232, Springer-Verlag, 1997, p. 323–326. 15

[Riv04] Xavier Rival – « Symbolic transfer function-based approaches to certified compilation », *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'04)*, ACM Press, 2004, p. 1–13. 49

[RM99] Martin C. Rinard et Darko Marinov – « Credible compilation with pointers », *Proceedings of the FLoC Workshop on Run-Time Result Verification* (Trento, Italy), Juillet 1999. 35

[Str02] Martin Strecker – « Formal verification of a java compiler in Isabelle », *Proceedings of the 18th International Conference on Automated Deduction*, Springer-Verlag, 2002, p. 63–77. 33

[VB98] Eelco Visser et Zine-el-Abidine Benaissa – « A core language for rewriting », *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)* (Pont-à-Mousson (France)) (C. Kirchner et H. Kirchner, éds.), vol. 15, Electronic Notes in Theoretical Computer Science, Septembre 1998. 123

[VBT98] Eelco Visser, Zine-el-Abidine Benaissa et Andrew Tolmach – « Building program optimizers with rewriting strategies », *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, ACM Press, Septembre 1998, p. 13–26. 15, 118, 123, 135

[Vis01] Joost Visser – « Visitor combination and traversal control », *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'01)* (New York, NY, USA), ACM Press, 2001, p. 270–282. x, 5, 72, 120, 123

[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 I, Octobre 1994. 15, 116

[Vit96] — , « A compiler for nondeterministic term rewriting systems », *Proceedings of the 7th International Conference on Rewriting Techniques and Applications (RTA'96)* (London, UK) (H. Ganzinger, éd.), Lecture Notes in Computer Science, vol. 1103, Springer-Verlag, Septembre 1996, p. 154–167. 15

[Wad87] Philip Wadler – « Views : a way for pattern matching to cohabit with data abstraction », *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ACM Press, 1987, p. 307–313. 39

[WAK97] Daniel C. Wang, Andrew W. Appel et Jeff L. Korn – « The zephyr abstract syntax description language », *USENIX Workshop on Domain-Specific Languages*, 1997. 72

[Wan95] Mitchell Wand – « Compiler correctness for parallel languages », *Proceedings of the seventh international conference on Functional programming languages and computer architecture (FPCA'95)* (New York, NY, USA), ACM Press, 1995, p. 120–134. 33

[WAS03] Dinghao Wu, Andrew W. Appel et Aaron Stump – « Foundational proof checkers with small witnesses », *Proceedings of the 5th ACM SIGPLAN international Conference on Principles and Practice of Declarative Programming*, ACM Press, 2003, p. 264–274. 35

[ZHS04] David Zook, Shan Shan Huang et Yannis Smaragdakis – « Generating aspectj programs with meta-aspectj. », *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE'04)*, 2004, p. 1–18. 154

Table des figures

Index

Résumé

La plupart des processus informatiques mettent en jeu la notion de transformation, en particulier la compilation. La réécriture est un formalisme qui permet de décrire et de raisonner sur ces transformations. Nous nous intéressons dans cette thèse à fournir des outils et des méthodes, utilisant ce formalisme, et permettant d'accroître la confiance que l'on peut placer dans ces processus.

La base de cette étude est le langage Tom, qui intègre filtrage et réécriture dans des langages généralistes comme Java ou C. Nous développons dans un premier temps un cadre permettant de valider la compilation de constructions de filtrage, produisant une preuve formelle de la validité de la compilation, ainsi qu'un témoin de cette preuve, à chaque exécution du compilateur. Ensuite, afin de permettre l'écriture sûre de transformations complexes, nous proposons un outil permettant de générer une structure de données efficace intégrant des invariants algébriques en Java, ainsi qu'un langage de stratégies permettant de contrôler l'application des transformations. Nous étudions formellement les propriétés de fonctions de normalisation pour la théorie associative avec élément neutre, et les correspondants variadiques. Nous détaillons ensuite le langage de stratégie proposé. Les stratégies permettent de composer des transformations complexes en combinant transformations élémentaires et combinateurs génériques de manière algébrique. La réflexivité de ces constructions stratégiques autorise leur modification pour intégrer traçage et analyse des transformations.

Ces résultats constituent donc une avancée vers la constitution de méthodes génériques sûres pour le développement de transformations de confiance.

Mots clefs : Compilation, certification, confiance, filtrage algébrique, termes, structures de données, stratégies de réécriture, langages de programmation

Abstract

Most computer processes involve the notion of transformation, in particular the compilation processes. Rewriting is a formalism enabling description and reasoning about those transformations. We interest in this thesis in providing tools and methods, based on this formalism, and giving the opportunity to increase the confidence we can place into those processes.

The basis of this study is the Tom language, which integrates matching and rewriting into general-purpose languages such as Java and C. We develop first a framework used to validate the compilation of matching constructs, building a formal proof of the validity of the compilation process along with a witness of this proof, for each run of the compiler. Then, in order to allow one to write safely complex transformations, we propose a tool that generates an efficient data structure integrating algebraic invariants in Java, as well as a strategy language that enables to control the application of transformations. We formally study the properties of normalisation functions for the associative with neutral element theory, and their variadic counterparts. We then detail the proposed strategy language. With strategies, complex transformations are composed by combining elementary transformations and generic combinators in an algebraic manner. The reflexivity of those strategic constructions permits their modification to integrate tracing and analysis of those transformations.

Those results can be seen as a first step towards the constitution of generic and safe methods for the development of trustworthy transformations.

Keywords: Compilation, certification, confidence, algebaic matching, terms, data structures, rewriting strategies, programming language

INSTITUT NATIONAL
POLYTECHNIQUE
DE LORRAINE

AUTORISATION DE SOUTENANCE DE THESE
DU DOCTORAT DE L'INSTITUT NATIONAL
POLYTECHNIQUE DE LORRAINE

o0o

VU LES RAPPORTS ETABLIS PAR :
Monsieur Pierre COINTE, Professeur, École des Mines de Nantes, Nantes
Monsieur Paul KLINT, Professeur, Université d'Amsterdam, Hollande

Le Président de l'Institut National Polytechnique de Lorraine, autorise :

Monsieur REILLES Antoine

NANCY BRABOIS
2, AVENUE DE LA
FORET-DE-HAYE
BOITE POSTALE 3
F - 54501
VANDŒUVRE CEDEX

à soutenir devant un jury de l'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE, une thèse intitulée :

"Réécriture et compilation de confiance"

en vue de l'obtention du titre de :

DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

Spécialité : « **Informatique** »

Fait à Vandoeuvre, le 14 novembre 2006
Le Président de l'I.N.P.L.,
L. SCHUFFENECKER

INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE - NANCY

TEL. 33/03.83.59.59.59
FAX. 33/03.83.59.59.55