Conception d'un langage dédié à l'analyse et la transformation de programmes

Emilie Balland

Nancy-Université
Université Henri Poincaré

Département de formation doctorale en informatique

Université Henri Poincaré

École doctorale IAEM Lorraine

Conception d'un langage dédié à l'analyse et la transformation de programmes

THÈSE

présentée et soutenue publiquement le 11 Mars 2009

pour l'obtention du

Doctorat de l'Université Henri Poincaré

(spécialité informatique)

par

Emilie Balland

Composition du jury

Président :	Bernard Girau	Professeur, Université Henri Poincaré, Nancy, France
Rapporteurs :	Olivier Danvy	Professeur associé, Aarhus, Danemark
	Stéphane Ducasse	Directeur de recherche, INRIA, Lille, France
Examinateurs :	Rachid Echahed	Chargé de recherche, CNRS, Grenoble, France
	Claude Kirchner	Directeur de recherche, INRIA, Nancy, France
	Pierre-Étienne Moreau	Chargé de recherche, INRIA, Nancy, France
	Joost Visser	Directeur du service R&D de SIG, Amsterdam, Pays-Bas

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

Loria

Mis en page avec LaTeX

À Paul,

Remerciements

Je tiens tout d'abord à remercier l'ensemble des enseignants de l'Université Henri Poincaré qui m'ont fait découvrir l'informatique en tant que science et tout particulièrement Martine Gautier et Karol Proch qui m'ont communiqué le goût pour les langages de programmation. Je souhaite aussi remercier Dominique Méry qui, en m'accueillant en stage, m'a permis de découvrir le monde de la recherche et m'a donné envie de faire un doctorat.

Je remercie mes directeurs de thèse Pierre-Etienne Moreau et Claude Kirchner, qui m'ont accompagnée pendant ces trois années et ont toujours su trouver les mots pour m'encourager. Par leur disponibilité et leur patience, ils m'ont offert des conditions de travail exemplaires et je leur en suis profondément reconnaissante. Merci en particulier à Pierre-Etienne pour son enthousiasme communicatif.

Je voudrais remercier les personnes qui ont accepté d'être membres de mon jury. Stéphane Ducasse et Olivier Danvy ont accepté la tâche d'être rapporteurs. Je tiens à les remercier pour leurs commentaires et pour l'intérêt qu'ils ont manifesté pour ce travail. Je suis très touchée que Rachid Echahed ait accepté d'être président de ce jury car durant cette thèse, nos échanges ont profondément contribué à améliorer mon travail sur les termes-graphes. Je remercie enfin Joost Visser et Bernard Giraud qui ont accepté de jouer le rôle d'examinateurs.

Merci à mes collègues et amis de l'équipe Protheo/Pareo pour tous les agréables moments que nous avons partagés. C'est avec beaucoup de nostalgie que j'ai vu chacun terminer son doctorat et aujourd'hui, je veux profiter de ces quelques mots pour leur souhaiter à chacun de réussir dans ce qui leur tient à cœur. Je tiens aussi à remercier mes amis du projet Tom. En particulier, merci à Antoine Reilles avec qui j'ai beaucoup travaillé et qui a toujours été de très bon conseil. Ses qualités scientifiques et ses compétences en programmation m'ont toujours impressionnée et il a toujours su les partager avec beaucoup de simplicité. Merci aussi à Radu Kopetz et Guillaume Burel avec qui j'ai eu la chance de partager mon bureau et qui ont toujours été très attentionnés et d'une bonne humeur constante. Je tiens à remercier aussi mes collègues du projet RAVAJ et notamment Yohan Boichut et Thomas Genet pour leurs qualités scientifiques et humaines.

Un grand merci à Yannick Parmentier, Sébastien Hinderer, Eric Kow et Dmitry Sustretov pour tous les bons moments passés ensemble en France et dans les quatre coins de l'Europe. Enfin, je tiens à remercier ma famille et mes amis. Notamment merci à Jacques sans qui je ne serais pas là et dont le soutien sans borne a toujours été d'un très grand réconfort. Merci enfin à Paul pour son amour et la patience avec laquelle il a relu chaque ligne de cette thèse.

Table des matières

Extended Abstract

Developing static analysers is a tedious task that requires in particular to intensively manipulate tree and graph structures. Even if mainstream languages like Java or C have a lot of specific libraries to deal with such structures, as there is no built-in language statement to work with trees and graphs, the resulting code is unnecessarily complex and thus prone to errors.

In the term rewriting community, the notion of rewrite rule *is an abstraction that has been intensively used to model, study, and analyze different parts of complex systems. Given a set of rules, the control of rule application can be described in a distinct expression called a* strategy*. By separating rules and control, it is easier to reason about properties such as termination or confluence. All these notions have been constituting the basis of numerous programming languages such as Maude [CEL99], ELAN [BKK+98] or Stratego [VBT98]. The experience showed that these primitives offer a good level of abstraction to describe program analyses and transformations. Unfortunately, developing a whole static analyser only with these constructions is not realistic. A lot of analyses require to manipulate big collections and it is more natural to represent them by mutable data-structures than algebraic terms. For example, in the case of points-to analysis, an efficient way to represent the resulting sets is to use Binary Decision Diagrams.*

The goal of this thesis was to propose specific language statements adapted to static analysers prototyping. The main idea was to take inspiration from notions introduced in term rewriting to offer primitives for manipulating tree structures. But instead of developing a new standalone language, the chosen approach was to integrate all the notions of pattern-matching, rewriting and strategies as new language constructs in Java. The Tom [BBK+07] language provides a higher level language piggybacked on top of Java for describing tree traversals and transformations. The main advantage of this embedded *approach is to use these new language statements only when necessary and to take advantage of external Java libraries when it is more suitable.*

Formal islands

Motivated by the proliferation and usefulness of embedded domain specific languages as well as by the demand for enriching mainstream languages with high level capabilities, we introduce the *Formal Islands* framework. The embedded domain specific language and the host generalist language are respectively called *island* and *ocean*. The parts of

the code written in the island language are translated into the ocean language using a source to source transformation called *dissolution*.

This framework presented in [BKM06] enables us to define the combination of these two languages in terms of syntax and semantics. In particular, it makes explicit the interface between the two languages and enables one to certify the development of dedicated embedded languages. An example of island language is Tom, which adds rewriting features to generalist languages. The theoretical foundations provided by the *Formal Island* framework helped us to understand how properties of Tom such as termination can be preserved by compilation.

Strategies for static analysis

In term rewriting, a strategy is a way to control the application of a set of rules on a term. The strategy specifies which rules are applied and at which positions. We argue that strategy languages are also well-adapted for collecting information inside a complex tree structure. Thus the combination of pattern-matching and strategies can greatly facilitate the development of static analysers. In the domain of program analysis, the strategy language as proposed by rewriting engines suffers from some limitations. This thesis defines a new strategy language smoothly integrated into Java, flexible enough to express complex traversals and to collect information using Java collections. This language is influenced by the strategy languages offered by ELAN and Stratego. One of its main characteristics is the data-structure independence. Any Java data-structure can be traversed thanks to the Tom anchor. The language design allows complex definitions of traversals. In particular, we introduce an original strategy combinator that enables back and forth traversals of trees. This has lead to the full specification of the Java identifier lookup using strategies. Moreover, special strategy combinators enable graph traversals.

Graph manipulation

Graphs are omnipresent in program analysis. The implementation of static analysers require the representation of control-flow and data-flow graphs for instance. As Tom can only manipulate tree structures, we proposed an extension to deal with graph structures as shown in [BB07, BM08].

The main idea is to use paths to represent cycles and shared parts. Paths are defined as a sequence of integers and denote a relation from a source position to a target one. The main originality of this approach is to make the notion of path first-class, i.e. paths become *part* of the terms and not just meta-information about them. The first contribution is the introduction of referenced terms in order to ground an extension of term rewriting. In referenced terms, paths are used to express references and thus to provide a natural way to add pointers in classical terms. Based on the formalization of paths and on the notion of rewrite relation for referenced terms, the second contribution is the simulation of term-graph rewriting by referenced term rewriting. Since this simulation

is completely based on standard first-order terms, another main interest of the approach is to provide a safe and efficient way to represent and transform term-graphs in a purely term rewriting based language. This leads to an original and clean way of representing and transforming graphs in a rewrite-based environment, in particular allowing the use of strategies. All these formalisms have been completely integrated in the Tom language.

Applications for Java analysis

Bytecode analysis. There exist several tools for manipulating Java Bytecode, among them Soot, BCEL and ASM. Although they are quite powerful, a deep knowledge of the API may be needed to use them effectively. In this thesis, we propose a new abstraction level, based on terms and rewriting, to make the definition of high-level transformations and analyses easier. Using the notion of algebraic view, we have extended the ASM library such that a Bytecode program can be seen as a tree [BMR07b]. This gives us the possibility to directly express transformation rules, and thus to reduce the gap between the user's wishes and the language expressiveness. We propose an application of Bytecode rewriting for secure class loading by redirecting method invocations to new targets based on a security policy. In particular, we enforce the use of a safe API for file accesses. Another application of Bytecode analysis is the compilation of strategies at runtime [BMR07a] directly on Bytecode programs.

Source analysis. Refactoring is the process of improving the design of existing code by performing behaviour-preserving program transformations. It can be done manually but most of the refactorings require tedious, error-prone manipulation of the code. Such transformations are complex and even the most sophisticated IDEs contain bugs in their refactoring features. For example, their implementation depends on the identifier lookup analysis which is very hard to specify because of the large number of visibility rules. We propose to specify the Java name analysis using strategies. Given its declarative nature, the code is more legible and thus the level of confidence increases.

Roadmap

After a brief presentation of basic notions in Chapter 1 and of the Tom language in Chapter 2, Chapter 3 presents the foundation of a general framework for embedded domain specific languages called *Formal islands*. The Chapter 4 defines an extensible strategy language integrated into Java which makes possible the definition of generic traversals over trees but also over cyclic graph data structures. In Chapter 5, a complete formalisation of term-graph data structures and term-graph matching based on first-order terms is proposed and we show how this formalism can be integrated in the Tom language. To finish, Chapter 6 presents several application prototypes of theses language constructs to Java analysis.

Introduction

Développer des analyseurs statiques est une tâche difficile qui requiert en particulier une manipulation intensive de structures d'arbres et de graphes. Même si la plupart des langages généralistes tels que Java ou C++ disposent de nombreuses bibliothèques dédiées à la manipulation de telles structures, l'absence d'instruction dédiée à ces tâches dans le langage rend le code complexe et donc difficile à maintenir.

Dans la communauté de réécriture, la notion de règle *est une abstraction qui a été beaucoup utilisée pour modéliser et analyser des systèmes complexes. Étant donné un ensemble de règles, le contrôle de l'application des règles peut être décrit séparément dans une expression appelée* stratégie. *En séparant les règles du contrôle de leur application, il est beaucoup plus facile de raisonner sur le système. De plus, tous ces concepts ont donné lieu à de nombreux langages de programmation tels que Maude [CEL99], ELAN [BKK+98] ou encore Stratego [VBT98].*

L'expérience a montré que ces notions offraient un bon niveau d'abstraction pour décrire des analyses et des transformations de programmes. Cependant, développer un analyseur statique complet uniquement à partir de ces primitives n'est pas très réaliste. De nombreuses analyses nécessitent de manipuler de grands ensembles et il est donc plus naturel d'utiliser des structures de données mutables plutôt que des termes algébriques. Par exemple, dans le cas des analyses de pointeurs, une manière efficace de représenter l'ensemble des résultats est d'utiliser des diagrammes de décision binaires.

Le finalité de cette thèse est donc de proposer des constructions de langage dédiées au protypage d'outils d'analyse et de transformation de programmes en se fondant sur les notions introduites dans le domaine de la réécriture de termes (le filtrage, la réécriture et les stratégies). Mais au lieu de développer un langage dédié, notre approche est d'embarquer tous ces concepts dans les langages généralistes tels que Java sous la forme d'un langage dédié embarqué *décrit ici formellement sous le concept plus général de* langage îlot. *Le principal intérêt de cette approche est de pouvoir profiter du langage hôte lorsque cela s'avère plus adapté.*

Les travaux de cette thèse se fondent sur le langage Tom [BBK+07] qui propose d'embarquer des constructions de réécriture dans des langages généralistes comme Java. L'ensemble des propositions de cette thèse a donc été intégré au langage Tom sous la forme de nouvelles constructions syntaxiques ou d'améliorations de constructions existantes.

Un cadre formel pour les langages dédiés embarqués

Un langage dédié est un langage de programmation spécifique à un domaine d'application et donc offrant les bonnes abstractions de ce domaine à l'utilisateur. Les principales caractéristiques d'un langage dédié sont un nombre limité de constructions ainsi que des notations et des abstractions appropriées à un domaine. Ils s'opposent donc en ce sens aux langages dits généralistes tels que Java ou C. Le code écrit dans un langage dédié est souvent beaucoup plus déclaratif et permet une meilleure maintenance du code.

La prolifération des langages dédiés et des plateformes multi-langages telles que `.NET` pose de nouvelles questions sur la sémantique d'applications développées à l'aide de plusieurs langages. En particulier, les interfaces entre langages sont parfois tellement complexes qu'il est difficile de se convaincre par exemple que les propriétés structurelles des données partagées sont préservées. Cette nouvelle problématique a donc donné lieu à de nouveaux formalismes sémantiques [Tah99, GWDD06, MF07] permettant de décrire l'intéropérabilité entre plusieurs langages.

Durant cette thèse, nous avons formalisé un cadre général permettant de raisonner sur les langages tels que Tom, c'est à dire des langages dédiés embarqués dans des langages généralistes. En effet, comme les constructions des deux langages s'entrelacent, les interactions avec le langage hôte sont beaucoup plus directes que dans la situation d'un langage isolé communiquant avec d'autres langages au travers d'interfaces. Ils sont donc en général plus difficiles à formaliser. Le cadre que nous proposons [BKM06] permet de définir la combinaison des syntaxes et des sémantiques des deux langages au travers d'ancrages entre les syntaxes et les valeurs des deux langages et ainsi déterminer quelles propriétés sont suffisantes pour préserver la sémantique du langage embarqué. Ce formalisme a ainsi permis de certifier le compilateur Tom [KMR05b].

Définition d'analyse par stratégies

Le but de cette thèse est de proposer des constructions de langage adaptées à la fois à la définition de transformations et d'analyses de programmes. La plupart de ces traitements sont généralement réalisés sur l'arbre abstrait et donc un langage dédié à l'analyse doit fournir un moyen déclaratif d'exprimer un parcours sur ce type de structures. En réécriture, le concept de stratégie [BKV03a, KKK08] a été introduit pour contrôler l'application d'un système de règles afin d'assurer sa terminaison et sa confluence. Cette notion est très similaire à la notion de stratégie d'évaluation dans le lambda-calcul [Bar84]. La plupart des langages à base de règles proposent des constructions déclaratives permettant de décrire une stratégie, comme par exemple dans le langage ELAN [KKV93, KM01], Stratego [VBT98] ou plus récemment dans Maude [MOMV05].

En s'inspirant de ces langages, nous avons défini des constructions de langage intégrées dans Java qui permettent de spécifier de manière déclarative une stratégie de transformation d'une structure Java. Par rapport aux autres langages de stratégie, trois particularités de ce langage le rendent mieux adapté à l'expression d'analyses de programmes :

– l'indépendance du langage par rapport aux structures de données,
– l'environnement d'exécution des stratégies permettant de connaître le contexte d'évaluation mais aussi de définir des parcours plus complexes comme ceux dans les graphes,
– l'interaction avec le langage Java permettant de tirer avantage au mieux des deux paradigmes.

En plus de la conception du langage, une des contributions de cette thèse a été de formaliser sa sémantique opérationnelle. De plus, nous avons proposé un encodage des opérateurs de logique temporelle arborescente [CES86] permettant de vérifier des formules temporelles sur des arbres abstraits directement par stratégies. L'intérêt pratique est de pouvoir exprimer de manière concise des analyses et de pouvoir raisonner sur les stratégies correspondantes. David Lacey a montré dans sa thèse [LM01] que ce type de formules logiques couplées à la réécriture permet de définir des optimisations de programmes complexes de manière concise et déclarative.

Simulation de la réécriture de termes-graphes

Plusieurs théories formalisent la transformation de graphes. Une de ces théories est née de la généralisation de la réécriture de termes aux termes-graphes (termes pouvant contenir des sous-termes partagés et des cycles). La première définition a été introduite en 1987 par Barendregt [BEG$^+$87] et ne gérait pas les cycles. En effet, ce formalisme avait été d'abord introduit pour réaliser des implémentations plus efficaces des langages fonctionnels en partageant les calculs [HP91]. Le concept des termes-graphes a ensuite été étendu aux cycles [AK96, Plu99] afin de représenter des termes infinis réguliers comme les termes rationnels [CD97]. La structure de terme-graphe cyclique s'avère assez expressive pour pouvoir représenter de manière satisfaisante des structures de flot comme le flot de données ou le flot de contrôle, très utiles pour la définition d'analyses de programmes. C'est pour cette raison que nous avons proposé un formalisme permettant d'intégrer de manière non intrusive la réécriture de termes-graphes dans un langage à base de règles comme le langage Tom.

Afin que cette extension soit la moins intrusive possible, notre approche a consisté à représenter les termes-graphes par des termes. Une première contribution a donc été d'utiliser une notion de chemins pour représenter les cycles et les sous-termes partagés. Un deuxième apport a été la définition d'un algorithme original permettant d'exprimer le pas de réécriture de termes-graphes comme défini dans [AK96] par des opérations de redirection de pointeurs. On obtient ainsi une simulation de la réécriture de termes-graphes dans un langage manipulant des termes. L'ensemble de ces travaux, présenté dans [BB07, BM08], a été implémenté et intégré dans le langage Tom, profitant ainsi de l'environnement de programmation Java mais aussi des autres constructions du langage Tom comme la bibliothèque de stratégies.

Analyse de programmes Java

Afin de valider l'ensemble des constructions de langage proposées dans cette thèse, nous avons réalisé plusieurs prototypes d'applications à l'analyse de programmes Java.

Analyse de code source. Le *refactoring* est un procédé qui consiste à améliorer le code d'une application par transformation de programmes. Ce type de transformation doit préserver la sémantique du code. Or même de simples transformations telles que le renommage de variable s'avèrent très difficiles à implémenter car une transformation naïve peut entraîner le masquage d'autres variables. La plupart des implémentations proposées dans les environnements de programmation comme par exemple Eclipse contiennent des *bugs*. Dans le cas du renommage de variable, toute la difficulté est concentrée dans la correction de l'implémentation de la résolution de noms (c'est à dire trouver la déclaration associée à un nom). Comme cela a été démontré dans [SEM08], cette tâche s'avère assez difficile à cause du nombre important de règles de visibilité. Nous proposons dans cette thèse une implémentation de cette analyse en Tom fondée sur les stratégies. La déclarativité du langage offre un code beaucoup plus lisible et facilement extensible.

Analyse de Bytecode Java. Afin de réaliser des analyses de Bytecode Java, nous avons développé une bibliothèque permettant de manipuler un programme Bytecode comme un terme Tom. Ainsi il est possible de filtrer des programmes écrits en Bytecode, de les transformer puis de regénérer la classe Java. Le principal avantage de cette approche est de permettre aux utilisateurs de décrire leurs transformations sous forme de règles et ainsi d'améliorer la confiance en leurs programmes. Une application de ces travaux a été la réalisation en Tom d'un chargeur de classes paramétré par une politique de sécurité présenté dans [BMR07b]. Cette application est en mesure par analyse/transformation de Bytecode d'assurer l'application de la politique de sécurité sans modification de la machine virtuelle Java. Une autre application de l'analyse de Bytecode en Tom a été la compilation à la volée des stratégies [BMR07a].

Plan de la thèse

Après une brève présentation des concepts utilisés tout au long de la thèse dans le chapitre 1 et du langage Tom dans le chapitre 2, les chapitres suivants présentent les principales contributions de cette thèse :

- le chapitre 3 présente le formalisme des îlots formels permettant de modéliser des langages embarqués tels que Tom. Ce formalisme se fonde sur le concept d'ancrages qui permet d'exprimer comment les environnements d'exécution des deux langages peuvent communiquer et quelles propriétés structurelles des valeurs doivent être préservées.
- le chapitre 4 concerne la formalisation et l'implémentation du langage de stratégies de Tom dans Java. Ce langage permet d'exprimer des traversées génériques de structures arborescentes mais aussi de structures de données cycliques telles que les

termes-graphes. Ainsi une stratégie décrit de manière déclarative le contrôle d'un ensemble de transformations ou d'analyses complexes de programmes. Les transformations sont réalisées par réécriture et les analyses par filtrage et en collectant de l'information lors du parcours de la structure. Ce langage a été conçu de manière à favoriser l'interaction avec le langage Java, ce qui offre une grande souplesse de programmation.

– le chapitre 5 propose un encodage de la réécriture de termes-graphes dans un langage à base de termes du premier ordre. Cet encodage se fonde sur des structures de termes étendus aux pointeurs. Le pas de réécriture des termes-graphes est simulé par des opérations de redirection de pointeurs. Cette extension implémentée dans Tom permet de définir et de manipuler directement des graphes de flot de programmes.
– le chapitre 6 présente la validation de ces constructions dédiées au travers de plusieurs prototypes d'application à l'analyse et la transformation du langage Java.

1 Notions préliminaires

Nous présentons dans ce premier chapitre quelques notions nécessaires à la lecture du manuscrit. Pour une introduction plus détaillée, le lecteur peut se référer à [BN98].

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 considère parfois des termes obtenus en répartissant les variables et les opérateurs dans des classes appelées *sortes*. On parle alors de signature multi-sortée.

Définition 2 (Signature multi-sortée)**.** *Une signature multi-sortée est un couple $(\mathcal{S}, \mathcal{F})$ où $\mathcal{S}$ est un ensemble de sortes et $\mathcal{F}$ est un ensemble d'opérateurs sortés défini par $\mathcal{F} = \bigcup_{S_1,\ldots,S_n,S \in \mathcal{S}} \mathcal{F}_{S_1,\ldots,S_n,S}$. Le rang d'un symbole de fonction $f \in \mathcal{F}_{S_1,\ldots,S_n,S}$ noté $rank(f)$ est défini par le tuple $(S_1, \ldots, S_n, S)$ et on le note souvent $f : S_1 \times \cdots \times S_n \to S$.*

Pour simplifier, le reste de ces définitions se fonde sur des signatures sans sorte mais peut s'étendre facilement au cas multi-sorté (comme défini dans [BKV03a]). À partir d'une signature, on peut définir les termes construits sur ces opérateurs et un ensemble infini dénombrable de variables $\mathcal{X}$.

Définition 3 (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 :*

- *$\mathcal{X} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$: 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 $\mathcal{T}(\mathcal{F}, \mathcal{X})$.*

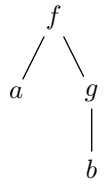
Pour tout terme t de la forme $f(t_1, \ldots, t_n)$, le symbole de tête de t, noté $symb(t)$ est par définition l'opérateur f. Par abus de notation, le terme $a \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et le symbole de fonction $a \in \mathcal{F}$ d'arité 0 sont confondus.

Définition 4 (Variables)**.** *L'ensemble $Var(t)$ des variables d'un terme t est défini par récursion sur la structure du terme :*

- $Var(a) = \emptyset$ *où* $a \in \mathcal{F}_0$,
- $Var(x) = \{x\}$ *où* $x \in \mathcal{X}$,
- $Var(f(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} Var(t_i)$ *où* $f \in \mathcal{F}$ *et* $n = ar(f)$.

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

Les termes peuvent être représentés sous une forme arborescente. Par exemple, le terme $f(a, g(b))$ peut se représenter de la manière suivante :



La notion de position permet d'identifier de manière unique chaque nœud d'un arbre.

Définition 5 (Position)**.** *Une position dans un terme t est représentée par un mot ω d'entiers naturels positifs décrivant le chemin de la racine du terme jusqu'à un nœud du terme. La position vide (correspondant au mot vide) est notée ϵ et la concaténation de deux positions ω_1 et ω_2 est notée $\omega_1 \cdot \omega_2$.*

Le sous-terme à la position ω (noté $t_{|\omega}$) est défini tel que

- $t_{|\epsilon} = t$,
- *si* $t = f(t_1, \ldots, t_n)$ *alors* $\forall i \in [1, n], t_{|i \cdot \omega} = t_{i|\omega}$.

Le remplacement à la position ω du sous-terme $t_{|\omega}$ par t' est noté $t[t']_\omega$. L'ensemble des positions de t se note $\mathcal{P}os(t)$.

Définition 6 (Préfixe)**.** *Nous notons $\sqsubseteq$ la relation classique de préfixe entre deux mots sur les positions ($\omega_1 \sqsubseteq \omega_2$ si il existe une position p tel que $\omega_1 \cdot p = \omega_2$).*

Nous notons $<_{\mathcal{P}}$ l'ordre lexicographique sur les positions. Par exemple $\epsilon <_{\mathcal{P}} 1$ et $1 \cdot 2 <_{\mathcal{P}} 1 \cdot 3$.

1.2 Filtrage

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

Définition 7 (Substitution)**.** *Une substitution σ est une fonction partielle de $\mathcal{X}$ vers $\mathcal{T}(\mathcal{F})$, notée $\sigma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$ lorsque son domaine est fini. 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, défini par récurrence sur la structure des termes :*

- $\sigma(x) = \begin{cases} \sigma(x) \text{ si } x \in \mathcal{D}om\,\sigma \\ x \text{ sinon} \end{cases}$
- $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$ *où* $f \in \mathcal{F}_n$.

On note $\mathcal{R}an(\sigma)$ le codomaine de la substitution σ.

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 8 (Filtrage). *Étant donnés un terme $p \in \mathcal{T}(\mathcal{F},\mathcal{X})$ appelé motif 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 $\sigma(p) = t$.*

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

1.3 Théorie équationnelle

Dans une théorie équationnelle, on appelle équation une paire de termes $l, r \in \mathcal{T}(\mathcal{F},\mathcal{X})$ (notée $l = r$) telle que $Var(r) \subseteq Var(l)$. Une équation est parfois appelée égalité, axiome équationnel ou encore axiome égalitaire.

Définition 9 (relation de réduction). *Étant donné un ensemble d'équations E, la relation binaire $\rightarrow_E \subseteq \mathcal{T}(\mathcal{F}) \times \mathcal{T}(\mathcal{F})$, appelée relation de réduction, est définie telle que $t \rightarrow_E t'$ si et seulement si il existe une équation $(l = r) \in E$, une position $\omega \in \mathcal{P}os(t)$ et une substitution σ telles que $t_{|\omega} = \sigma(l)$ et $t' = t[\sigma(r)]_\omega$.*

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

Soient deux relations binaires $\rightarrow_1$ et $\rightarrow_2$, on note $\rightarrow_1 \circ \rightarrow_2$ la composition de ces relations.

La fermeture réflexive, symétrique et transitive de $\rightarrow_E$ (notée $=_E$) est une théorie équationnelle dite théorie équationnelle engendrée par E ou plus brièvement, la théorie équationnelle E.

Dans cette thèse, les théories équationnelles que nous considérons sont l'associativité et la commutativité.

Définition 11 (Opérateur associatif). *Un opérateur binaire f est associatif si $\forall x, y, z \in \mathcal{T}(\mathcal{F},\mathcal{X}),\ f(f(x,y),z) = f(x,f(y,z))$.*

Définition 12 (Opérateur commutatif). *Un opérateur binaire f est commutatif si $\forall x, y \in \mathcal{T}(\mathcal{F},\mathcal{X}),\ f(x,y) = f(y,x)$.*

Définition 13 (Neutre). *Soit un opérateur binaire f, la constante $e \in \mathcal{T}(\mathcal{F})$ est neutre pour f si $f(e,x) = x$ et $f(x,e) = x$.*

On notera A et AU les théories équationnelles engendrées par l'équation d'associativité et respectivement par les équations d'associativité et neutralité. La commutativité sera dénotée par C.

Dans les langages de programmation fondés sur la réécriture, la théorie associative est souvent associée aux opérateurs variadiques (dont l'arité n'est pas fixée). Par exemple, l'opérateur variadique *list* est simulé par deux opérateurs d'arité fixe : l'opérateur *nil* d'arité nulle et l'opérateur binaire *cons*. Ainsi le terme $list(a,b,c)$ est équivalent au terme $list(list(a,b),c)$ et tous deux peuvent être représentés par $cons(a,cons(b,cons(c,nil)))$.

1.4 Systèmes de réécriture

En réécriture, on considère des égalités orientées appelées règles.

Définition 14 (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 appelé membre gauche de la règle, et r membre droit.*

Définition 15 (Système de réécriture)**.** *Un système de réécriture sur les termes est un ensemble de règles de réécriture (l, r) 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 16 (Réécriture)**.** *Un terme clos $t \in \mathcal{T}(\mathcal{F})$ 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 $t' = t[\sigma(r)]_\omega$.*

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

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 17 (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 $t \xrightarrow{*} t'$.*

1.5 Confluence et terminaison

Définition 18 (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 19 (Confluence)**.** *Soit une relation binaire $\to$ sur un ensemble T.*
- *$\to$ vérifie la propriété de Church-Rosser si et seulement si*

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

- *$\to$ 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*

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

1.6 Réécriture modulo une théorie équationnelle

Il existe des égalités que l'on ne peut pas orienter sans obtenir un système de règles non terminant. Un exemple typique est l'axiome de commutativité. Pour cela, une solution est de proposer une nouvelle relation de réécriture qui travaille modulo une théorie équationnelle.

Définition 20 (Réécriture modulo une théorie équationnelle)**.** *Le terme t se réécrit en t' par un système de réécriture R modulo une théorie équationnelle E s'il existe une règle $l \to r \in R$, une position $\omega \in \mathcal{P}os(t)$ et une substitution σ tels que $t_{|\omega} =_E \sigma(l)$ et $t' = t[\sigma(r)]_\omega$.*

1.7 Réécriture sous stratégies

Les concepts de stratégies et de réécriture sous stratégies ont été introduit afin de contrôler l'application d'un système et ainsi assurer sa confluence et sa terminaison. Cette notion est très similaire à la notion de stratégie d'évaluation dans le lambda-calcul [Bar84]. La concept de stratégie abstraite a été introduit dans [KKK08] et s'inspire de la définition de [BKV03a] fondée sur les systèmes abstraits de réduction [BKV03a, Klo90, KOR07] (notés ARS). En général, un ARS est présenté comme un ensemble et une relation binaire sur cet ensemble [BKV03a]. Ici, nous avons choisi les définitions de [KKK08] qui représentent un ARS sous forme d'un graphe. Le principal avantage de cette définition est de pouvoir raisonner plus facilement sur les différentes manières de réécrire un objet en un autre (correspondant à la notion de chemins dans un graphe).

Définition 21 (Système abstrait de réduction)**.** *Un système abstrait de réduction (ARS) est un graphe orienté étiqueté $(\mathcal{O}, \mathcal{S})$. Les nœuds $\mathcal{O}$ sont appelés objets, les arêtes orientées $\mathcal{S}$ sont appelés pas.*

Étant donné un ensemble de termes $\mathcal{T}(\mathcal{F}, \mathcal{X})$, l'ARS correspondant à un système de réécriture R est le graphe $(\mathcal{T}(\mathcal{F}, \mathcal{X}), \mathcal{S})$ où les arêtes sont étiquetées par le nom d'une règle de R et correspondent à des pas de réécriture de R (il existe une arête étiquetée $r_i \in R$ entre t_1 et t_2 si et seulement si t_1 se réécrit en t_2 par la règle r_i).

Définition 22 (Dérivation)**.** *Soit un ARS $\mathcal{A}$:*

1. *un pas de réduction est une arête étiquetée ϕ complétée de sa source a et de sa destination b. On note un pas de réduction $a \to^{\phi}_{\mathcal{A}} b$, ou simplement $a \to^{\phi} b$ lorsqu'il n'y a pas d'ambiguïté.*
2. *une $\mathcal{A}$-dérivation est un chemin π dans le graphe $\mathcal{A}$.*
3. *lorsque cette dérivation est finie, π peut s'écrire $a_0 \to^{\phi_0} a_1 \to^{\phi_1} a_2 \ldots \to^{\phi_{n-1}} a_n$ et on dit que a_0 se réduit en a_n par la dérivation $\pi = \phi_0\phi_1 \ldots \phi_{n-1}$; noté aussi $a_0 \to^{\pi} a_n$. La source de π est le singleton $\{a_0\}$ noté $dom(\pi)$. La destination de π est le singleton $\{a_n\}$ et est noté par $\pi[a_0]$.*
4. *une dérivation est vide si elle n'est formée d'aucun pas de réduction. La dérivation vide de source a est notée id_a.*

Définition 23 (Stratégie abstraite)**.** *Étant donné un ARS $\mathcal{A}$:*

1. *une stratégie abstraite ζ est un sous-ensemble de toutes les dérivations de $\mathcal{A}$.*
2. *appliquer la stratégie ζ sur un objet a, noté par $\zeta[a]$, est l'ensemble de tous les objets atteignables depuis a en utilisant une dérivation dans ζ :*
   $\zeta[a] = \{\pi[a] \mid \pi \in \zeta\}$.
   Lorsqu'aucune dérivation dans ζ n'a pour source a, on dit que l'application de la stratégie a échoué.
3. *la stratégie qui contient toutes les dérivations vides est $Id = \{id_a \mid a \in \mathcal{O}\}$.*

La formalisation des systèmes de réduction abstraits et des stratégies abstraites permet de définir de manière naturelle des propriétés telles que la terminaison (toutes les dérivations sont de taille finie) et la confluence (toutes les dérivations terminent par un même objet). Pour des définitions plus précises, le lecteur peut se référer à [KKK08]. Cette définition de stratégie est plus générale que celle proposée dans [BKV03a] qui considère une stratégie comme un sous-ensemble de l'ARS du système de règles ayant le même ensemble de formes normales.

Une stratégie peut être décrite de manière déclarative par un *langage de stratégies*. La plupart des langages à base de règles ont proposé des langages de stratégies comme par exemple ELAN [KKV93, KM01], Stratego [VBT98] et Tom [BBK+07, BBK+08]. L'ensemble de ces langages de stratégies sera présenté plus en détail dans le chapitre 4.

2 Contexte et motivations

2.1 Problématique de cette thèse

En informatique, le concept d'analyse statique de programmes recouvre un ensemble de méthodes permettant d'extraire de l'information concernant le comportement d'un programme sans l'exécuter. Cette notion est donc distincte de celle d'analyse dynamique (comme par exemple le *debugging*) qui est réalisée durant l'exécution du programme.

L'analyse statique de programmes a donné lieu à de nombreux outils qui traitent des problèmes variés allant de la vérification de propriétés de sûreté à l'extraction de métriques du code (un tour d'horizon des méthodes et des outils pour le langage Java est présenté dans la section 6.1). Ces analyseurs se fondent sur des bibliothèques permettant de manipuler des arbres ou des graphes représentant le programme à analyser. Les analyses (et les transformations afférentes) de ces structures sont en général écrites directement dans des langages généralistes comme Java ou C #. Or la plupart d'entre elles peuvent s'exprimer de manière déclarative et il est regrettable que le code obtenu ne reflète plus la formalisation sous-jacente. D'une part, il est plus difficile de se convaincre de leur correction et d'autre part, ces analyses sont difficilement extensibles ce qui pose problème lorsque les langages évoluent. Depuis quelques années, plusieurs projets proposent donc des langages dédiés à l'analyse et à la transformation de programmes.

2.1.1 Limites des langages dédiés existants

Un langage dédié est un langage de programmation spécifique à un domaine d'application et donc offrant les bonnes abstractions de ce domaine à l'utilisateur. Les principales caractéristiques d'un langage dédié sont un nombre limité de constructions ainsi que des notations et des abstractions appropriées à un domaine. Ce sont donc en général des langages plus déclaratifs que les langages généralistes tels que Java ou C #. Une introduction sur la conception des langages dédiés est présentée dans la section 3.1 et la sous-section 6.1.7 propose un tour d'horizon des langages dédiés à l'analyse statique et à la transformation de programmes.

La conception de ce type de langages soulève plusieurs questions comme par exemple :

- comment représenter les programmes ? (ex : arbres de syntaxe abstraite, arbres de syntaxe concrète, graphes de flot, bases de données, ...)
- comment extraire de l'information concernant le programme ? (ex : fonctions, grammaires attribuées, requêtes sur une base, ...)
- quel langage de transformation ? (ex : règles de réécriture, fonctions, ...)
- comment contrôler ces transformations ? (ex : fonctions d'ordre supérieur, annotations de priorité, langage de stratégies, ...)

– le langage à traiter est-il fixé ou programmable ?

Quelle que soit l'approche choisie, la plupart de ces langages souffrent de deux principales limites. La première limite porte sur l'intégration. Comme c'est le cas pour beaucoup de langages dédiés, leur intégration dans un projet existant est souvent difficile. Ces langages fournissent parfois leur propre environnement de développement ce qui nécessite alors que le développement des analyses et des transformations soit réalisé de manière isolée et programmé entièrement avec le langage dédié. Actuellement, certains langages dédiés tentent de palier à cette limite en développant des *plugins* pour IDE et en générant du code binaire pour les machines virtuelles comme `.NET` ou `JVM`. L'utilisation de plateformes multi-langages permet la communication entre le code dédié et le reste du projet qui peut alors être écrit en utilisant un langage généraliste. Dans cette thèse, nous proposons une autre approche en embarquant les constructions dédiées directement dans le langage généraliste, ce qui permet de plus fortes interactions. La deuxième limite des langages présentés dans la sous-section 6.1.7 est qu'ils sont dédiés soit à la transformation, soit à l'analyse. Or dans de nombreuses situations comme par exemple l'optimisation de programmes ou le *refactoring* de code, il est nécessaire de pouvoir à la fois collecter de l'information et réaliser des transformations complexes. Dans cette thèse, nous avons voulu essayer de remédier à ce problème en proposant des constructions de langage adaptées à ces deux tâches. Le langage proposé s'inspire de la réécriture de termes et de sa généralisation aux structures de graphes (plus précisément aux termes-graphes dont le concept sera introduit dans le chapitre 5).

2.1.2 Point de départ : le langage Tom

Au lieu de développer un langage dédié à partir de zéro, les travaux de cette thèse s'appuient sur le langage Tom [BBK+07] et se présentent sous forme d'extensions et d'améliorations de ce langage. Le langage Tom développé dans l'équipe depuis 2001 et présenté dans la section 3.4 est dédié à la transformation de structures arborescentes et permet d'embarquer des constructions de filtrage et de réécriture dans des langages généralistes comme Java ou C. En s'intégrant au langage Tom, le langage dédié proposé dépasse certaines des limites évoquées précédemment. En effet, Tom s'intègre facilement dans un projet existant et permet de réutiliser les outils de développement (compilation vers un langage de haut-niveau et non directement vers le langage de la machine virtuelle afin de pouvoir réutiliser les outils de développement fondés sur le code source). Cette capacité d'intégration du langage Tom est formalisée dans le chapitre 3 qui traite plus généralement des langages îlots (constructions du langage dédié embarquées dans un langage généraliste) et constitue une différence majeure entre les extensions de Tom dédiées à l'analyse de programmes introduites par cette thèse et les langages présentés dans la sous-section 6.1.7. En effet, un langage îlot profite de toute l'expressivité du langage hôte (en particulier toutes les bibliothèques et outils de programmation) tout en proposant des primitives spécifiques à son domaine d'application.

Durant le stage de Master qui a précédé cette thèse, nous avons développé un optimiseur pour le langage intermédiaire du compilateur Tom. Ces travaux ont permis de révéler plusieurs limites du langage Tom en tant que langage dédié à l'analyse et la trans-

formation de programmes. Ce langage est à l'origine dédié à la transformation d'arbres et il était donc difficile d'exprimer certaines analyses comme le calcul du nombre d'occurrences d'une variable dans une expression. En effet, certaines optimisations comme l'*inlining* nécessitent parfois de faire ce calcul sur le graphe de flot de contrôle afin de déterminer le nombre d'utilisations de la variable à l'exécution. Or le langage Tom ne permettait pas de manipuler des structures de graphes. De plus, nous nous servions des stratégies comme moyen de collecter l'information nécessaire à ce type d'analyses. Le code obtenu était alors trop complexe car le langage de stratégies était conçu à la base seulement pour contrôler l'application des règles et non pour collecter des informations dans un arbre. Il était alors parfois plus raisonnable d'exprimer certaines parties de l'analyse directement en Java plutôt qu'en Tom. Les extensions proposées dans cette thèse tentent de remédier à ces problèmes.

2.1.3 Caractéristiques du langage dédié proposé

Cette thèse propose donc des extensions de Tom adaptées à la fois à la transformation et à l'analyse de programmes en se fondant toujours sur les concepts de filtrage, de réécriture et de stratégies. Si l'on considère Tom comme un langage dédié à l'analyse et à la transformation de programmes, le langage Tom ainsi amélioré se caractérise par :

- les représentations de programmes traitées (arbres de syntaxe abstraite mais aussi graphes de flot grâce à l'extension pour les termes-graphes),
- la collecte d'informations par stratégies paramétrables par des collections Java et ayant accès à leur environnement d'exécution,
- un langage de transformations à base de règles de réécriture,
- le contrôle de l'application des règles également réalisé par stratégies,
- son indépendance par rapport au langage à analyser (les structures manipulées sont des termes algébriques).

2.2 Présentation du langage Tom

Le langage Tom a été conçu pour intégrer les concepts et techniques de réécriture dans les langages de programmation généralistes et largement utilisés, comme Java ou C. L'idée majeure est de pouvoir exprimer de manière déclarative des transformations d'arbres dans un langage généraliste. Un autre objectif de ce projet est d'offrir de nouvelles constructions qui soient le moins intrusives possible afin d'être utilisées dans des projets existants. 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]. Puis le compilateur de Tom a très rapidement été *bootstrappé* afin de permettre l'utilisation du filtrage dans le compilateur de Tom lui-même. Le langage a pu ainsi évoluer beaucoup plus rapidement et être utilisé dans le cadre académique comme un outil de protypage à base de réécriture mais aussi dans le milieu industriel pour écrire des transformations de structures de données arborescentes d'une manière plus déclarative. Nous décrirons dans ce chapitre le langage Tom de manière largement informelle. Les idées de base du langage sont décrites par les auteurs du tout premier compilateur

Tom [MRV03]. Le langage et ses bibliothèques ainsi que les outils associés sont décrits en détail dans le manuel de référence du langage [BBK+08]. Pour une présentation formelle du langage actuel, le lecteur peut se référer à [BBK+07].

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 type algébrique, ou de terme, mais uniquement de types de données composés comme les classes 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 1. Un exemple basique de programme Tom est 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 :

```
int plus(int t1, int t2) {
  %match(t1, t2) {
    x,zero() -> { return `x; }
    x,suc(y) -> { return `suc(plus(x,y)); }
  }
}

void run() {
    System.out.println("plus(1,2) = " + plus(1,2));
}
```

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 par filtrage, mais définit une fonction Java, qui peut alors être utilisée de manière classique par le code Java environnant.

La construction ‘ (appelée *backquote*) est utilisée pour exprimer la construction de termes algébriques, 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 rang (`Nat`,`Nat`,`Nat`), effectuant ainsi l'appel récursif.

Le code de l'exemple 1 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 n'analyse 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 (seuls les commentaires et les chaînes sont analysés).

L'originalité des constructions de filtrage syntaxique de Tom sont les membres droits qui au lieu d'être simplement des termes 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 1 utilise l'instruction Java `return` afin d'interrompre le flot de contrôle lorsqu'un filtre est trouvé.

2.2.2 Ancrages

On peut remarquer dans l'exemple 1 que la fonction Java `plus` prend en argument deux objets de type `int`, alors que la construction de filtrage est spécifiée sur le type de données algébrique `Nat` constitué des constructeurs `zero` et `suc`.

Cela illustre une spécificité importante 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 et comment ces objets concrets représentent les termes algébriques, par l'intermédiaire d'un ancrage.

Ces ancrages 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 de l'ancrage entre le type algébrique `Nat` et son implémentation par le type primitif `int`.

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 { int }
    equals(t1,t2) { t1 == t2 }
}
```

La sorte `Nat` est déclarée par la construction **%typeterm** comme implémentée par le type `int`. L'ancrage spécifie aussi que le test d'égalité de deux termes `Nat` peut être réalisé simplement par comparaison de valeurs. Les opérateurs de cette sorte sont alors définis en utilisant la construction **%op** qui permet de spécifier à la fois comment *construire* et *détruire* (dans le sens de décomposer) un terme dont le symbole de tête est cet opérateur. On définit donc les constructeurs `zero` et `suc` comme suit :

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

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

La première ligne de chaque construction **%op** définit la signature de l'opérateur algébrique, ainsi que les noms des éventuels sous-termes. La construction **is_fsym** permet de tester si un objet donné représente bien un terme dont le symbole de tête est l'opérateur, et la construction **get_slot** permet d'extraire les différents sous-termes en fonction de leur nom. Ces deux constructions constituent la partie destructive de l'ancrage. La construction `make` permet d'indiquer comment construire un terme de symbole de tête l'opérateur et dont les sous-termes sont passés en paramètres.

Les opérateurs variadiques sont définis de manière similaire en utilisant la construction **%oplist**. La première ligne spécifie le domaine et le co-domaine de l'opérateur, ainsi que son nom. Les opérations spécifiques aux listes doivent être définies, ce sont elles qui permettent de compiler le filtrage de liste. L'ancrage 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 { MyIntList }
    equals(l1,l2) { l1.equals(l2) }
  }
```

```
  %oplist List conc( Nat* ) {
    is_fsym(t)     { t instanceof MyIntList }
    make_empty() { new MyIntList() }
    make_insert(e,l) { new MyIntList(e,l) }
    get_head(l)    { l.get(0) }
    get_tail(l)    { l.sublist(1,l.size()) }
    is_empty(l)    { l.isEmpty() }
}
```

La définition de la définition **%oplist** pour la liste des entiers est implémentée par la classe `MyIntList` qui étend ArrayList<Integer> et propose des constructeurs pour construire une liste vide et une liste à partir de son élément de tête et d'une liste correspondant à la queue. Un ancrage ne doit pas nécessairement être complet. Il est parfois utile de ne spécifier que la partie destructive pour le filtrage. Par exemple, la bibliothèque du compilateur Tom fournit des ancrages partiels pour les implémentations standard des collections de la bibliothèque Java (paquetage `java.util`), qui peuvent être utilisées pour filtrer n'importe quelle collection Java.

Cette notion d'ancrage est inspirée des *views* de Philip Wadler [Wad87]. Ce concept a donné lieu au langage Pizza [OW97] qui est une extension du langage Java aux structures algébriques. En plus des types de données algébriques, ce langage offre les génériques, les clôtures ainsi que des constructions de filtrage. Les constructions de filtrage et de structures de données algébriques sont moins expressives que celles proposées par le langage Tom (il n'y a par exemple pas de filtrage modulo une théorie et les structures de données ne supportent pas le partage maximal, ni la gestion d'invariant). Des langages généralistes récents comme F # [Pic07] et Scala [OAC+06] proposent des primitives pour la définition de *views*. Ces constructions sont appelées respectivement *active patterns* dans F # [SNM07] et *extractors* dans Scala [Bur07]. Définir une vue permet de modifier le comportement du filtrage en proposant pour un même objet plusieurs manières de déconstruire sa valeur. Ce type de vue ne permet que de spécifier la partie destructive des ancrages. Comme les *extractors* de Scala, les ancrages proposés par Tom permettent en plus de construire les structures de données au travers des mêmes abstractions utilisées pour le filtrage.

2.2.3 Construction backquote

La construction ‘ permet de faire réaliser la construction d'un terme algébrique par Tom et se situe au niveau des expressions Java. Il est important de noter que les variables introduites par filtrage sont accessibles uniquement lorsqu'elles sont utilisées dans de telles constructions. 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. Soit l'instruction suivante :

```
int trois = ‘suc(suc(suc(zero())));
```

Cette instruction déclare une variable `trois` dont le type est `int`, qui est le type des représentants concrets des termes algébriques, comme ayant pour valeur le représentant du terme algébrique $suc(suc(suc(zero())))$ et comme valeur concrète l'entier 3.

Cela permet à l'utilisateur de ne pas avoir à connaître les détails de l'implémentation concrète du type Java mais de n'utiliser que sa *vue* algébrique. Les objets qui peuvent être vus comme des termes algébriques sont donc manipulés dans le programme de manière algébrique lors du filtrage et lors de la construction. Cela permet aussi de considérer des fonctions manipulant les termes algébriques à la manière d'opérateurs algébriques. Ainsi, dans la méthode Java `plus`, il est possible de rappeler la fonction `plus` à l'intérieur de l'expression `‘`. Notons que comme Tom n'analyse pas du tout le code hôte, l'algorithme de typage suppose que cette partie Java des expressions algébriques respecte les signatures des constructeurs algébriques. Durant cette thèse, nous avons commencé à prototyper une extension du typage de Java fondé sur les grammaires attribuées du langage JastAdd [EH07a] et permettant de typer complètement les expressions `‘`.

2.2.4 Filtrage associatif

Si le filtrage n'est pas unitaire (c'est-à-dire qu'il existe plusieurs substitutions solutions du problème de filtrage) alors l'action associée au motif sera exécutée pour chaque filtre. Le filtrage syntaxique est unitaire, et ne permet donc 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 variadique (c'est à dire que son arité n'est pas fixée). 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 syntaxique entre les variables de filtrage représentant des éléments, comme la variable `x` de l'exemple 1 et les variables représentant une sous-liste d'une liste existante, qui sont alors suivies de `*`.

Le filtrage de liste permet par exemple 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` avec l'opérateur de liste `conc`, on peut définir l'affichage de tous les éléments d'une liste par :

```
public void affiche(MyIntList l) {
  %match(l) {
    conc(X1*,x,X2*) -> {
      System.out.println("À␣l'indice␣"+‘X1.length()+"␣:␣"+‘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 `MyIntList`, ce qui permet d'utiliser la méthode `length()` qui retourne la longueur de la liste pour connaître l'indice de l'élément `x` dans la liste.

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. Le langage Tom ne spécifie pas l'ordre dans lequel les filtres sont énumérés, mais énumère les filtres un par un, sans se répéter. Cependant, concrètement, cette énumération est déterministe.

Les constructions de filtrage de Tom offrent aussi la possibilité d'exprimer des motifs non-linéaires. Ainsi, lorsqu'un même nom de variable apparaît plusieurs fois dans un motif, les sous-termes correspondant aux instances de ces variables doivent être égaux (au sens de l'ancrage). L'utilisation de motifs non-linéaires est en particulier beaucoup utilisée sur des listes. On peut par exemple définir ainsi une fonction éliminant les doublons dans une liste :

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

Enfin, les notations `_` et `_*` pour les sous-listes indiquent des variables non nommées. On ne peut alors pas utiliser sa valeur dans le membre droit de la règle. Dans le motif `conc(_*,zero(),_*)` les deux occurrences de `_*` désignent des variables non-nommées différentes, ce qui fait que le second motif itère sur les éléments de la liste. Ce motif permet donc de filtrer toutes les listes contenant au moins un élément `zero()`.

2.2.5 Contraintes de filtrage

En s'inspirant des règles de production de certains systèmes experts comme ILog ou JBoss, Radu Kopetz [Kop08] a proposé durant sa thèse plusieurs extensions des constructions de filtrage de Tom. Dans cette extension, la condition pour l'application d'une règle ne se limite plus à une simple contrainte de filtrage sur un terme, mais une combinaison de contraintes de filtrage et d'anti-filtrage. Le concept original d'anti-filtrage a été formalisé dans [KKM07, CRPE08]. Pour intégrer ce nouveau type de contraintes, la syntaxe de la construction `%match` a donc été généralisée.

Considérons la règle suivante : étant donnée une classe `Person`, ayant les champs `age` et `profession`, et un objet `pers` instance de cette classe, on veut exécuter la séquence

d'instructions notée `action` si l'âge de `pers` est plus grand que 60 et la profession est soit `Professor` soit `Researcher`. On peut par exemple écrire le programme Tom suivant :

```
%match(pers) {
  Person(age,profession) && age > 60 &&
  (Professor() << profession || Researcher() << profession) -> { action }
}
```

Avec l'ancienne syntaxe, cette règle se serait exprimée par des conditions Java et des instructions `%match` imbriquées. Avec cette nouvelle syntaxe, l'intention du code est plus claire car l'action de la règle est clairement séparée de ses conditions d'application, permettant de déduire plus facilement quand elle sera exécutée et facilitant les optimisations.

2.2.6 Notations

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.

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 d'assigner à une variable une valeur égale à ce sous-terme si le motif filtre. La notation `@` permet de définir de tels alias.

On peut par exemple 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(MyIntList l) {
  %match(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.

Notation crochets

Les signatures algébriques définies pour les opérateurs de Tom ont la particularité de donner un nom à chaque argument des opérateurs à la manière des structures du langage C. Il est alors possible d'utiliser ce nom dans un motif pour désigner un sous-terme particulier et ainsi 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].

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

La seconde écriture a certains avantages sur la première. Tout d'abord, le fait que l'on s'intéresse au *nom* dans ce motif est explicite ce qui rend la relecture plus facile. Ensuite, cette seconde écriture est plus robuste au changement : si la signature est étendue 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.

2.2.7 Architecture du compilateur Tom

Nous avons vu que le langage Tom est fondé sur la notion d'ancrage qui permet à l'utilisateur de spécifier comment manipuler n'importe quelle structure Java comme un terme Tom. Dans cette section, nous présentons un outil permettant à l'utilisateur (s'il ne souhaite pas utiliser de structures Java spécifiques) de définir directement une signature algébrique. Nous montrons ensuite comment cet outil intéragit avec le compilateur Tom et décrivons le schéma global de compilation d'une spécification Tom.

Générateur de structures de données

Tom propose un outil permettant de générer à partir d'une signature algébrique donnée un ensemble de classes Java implémentant cette signature, et fournissant l'ancrage pour cette signature. Cet outil, présenté en détail dans [Rei06a], fournit une implémentation 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 ainsi qu'un ancrage permettant à l'utilisateur d'utiliser cette structure dans des programmes Tom.

Les sortes `Nat` et `List` sont alors définies ainsi :

```
module Term
abstract syntax

Nat = zero()
    | suc(n:Nat)

List = conc(Nat*)
```

Dans cet exemple, `Term` est le nom du module, `Nat` est une sorte et `zero` et `suc` sont ses constructeurs. Un module peut aussi importer d'autres modules, en précisant leurs noms.

Les principales caractéristiques de ce générateur est d'offrir un typage fort au niveau du code généré, garantissant que les objets créés sont conformes à la signature multi-sortée, ainsi que du partage maximal [AG93], rendant ces structures très efficaces en temps (tests d'égalité en temps constant) et en espace. De plus, il est possible de modifier à l'aide de *hooks* les classes générées. On peut par exemple associer à certains opérateurs une théorie équationnelle telle que A, AC ou ACU. La théorie équationnelle peut aussi être spécifiée sous forme de règles de normalisation. L'implémentation assure que les termes construits sont toujours en forme normale. Enfin, il existe des *hooks* permettant d'ajouter des blocs de code Java aux classes générées permettant aux utilisateurs d'intégrer par exemple de nouveaux attributs qui ne seront pas visibles au niveau algébrique mais pourront être manipulés dans la partie Java de l'application.

Fonctionnement global du système

La figure 2.1 présente le fonctionnement global d'un développement en Tom. L'utilisateur peut soit manipuler ses propres structures de données Java et donc spécifier l'ancrage Tom correspondant ou alors définir une signature algébrique (notée dans la figure ADT pour *Algebraic Data Types*). Dans ce cas, le compilateur des signatures algébriques génère une hiérarchie de classes Java représentant la signature ainsi qu'un ensemble d'ancrages Tom pour ces classes. L'utilisateur peut alors directement utiliser cette signature dans un programme Tom sans avoir à spécifier les ancrages. Par ailleurs, nous avons vu qu'il est possible dans la signature algébrique de définir un ensemble de règles de normalisation. Ces règles de normalisation peuvent être spécifiées en utilisant

les constructions du langage Tom. Le compilateur de signatures algébriques appelle donc le compilateur Tom pour traduire ces règles en Java.

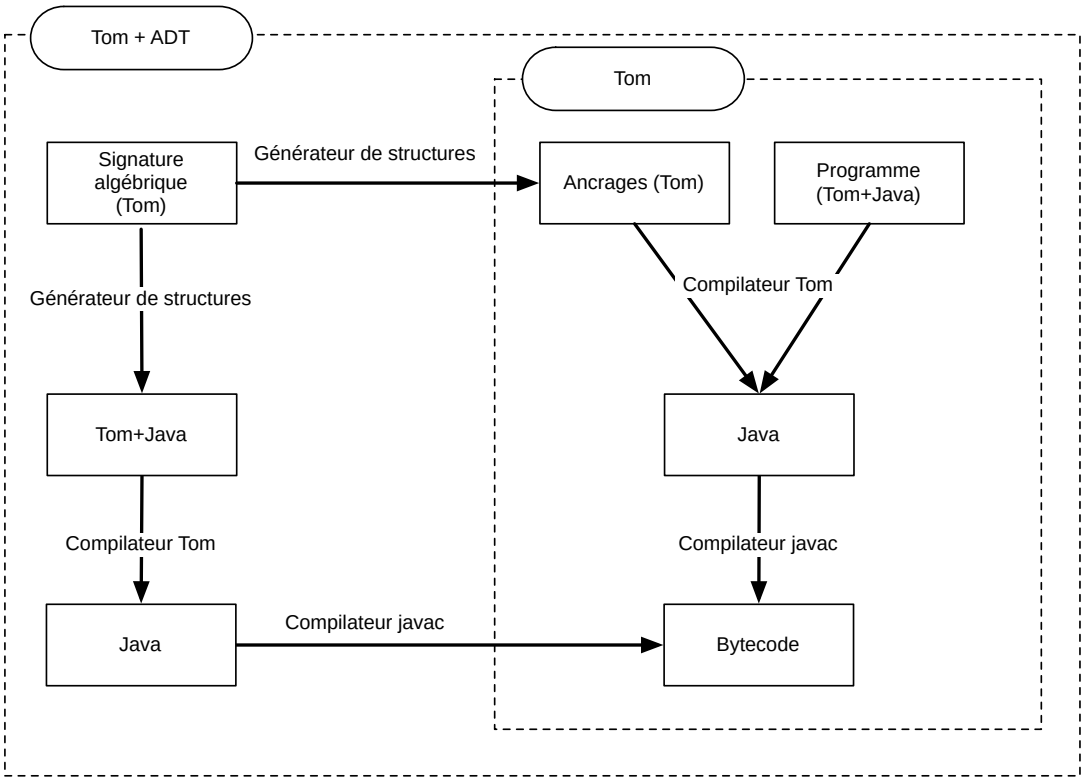


FIG. 2.1: Modes d'utilisation du langage Tom

Chaîne de compilation

La chaîne de compilation de Tom décrite dans la Figure 2.2 manipule l'arbre syntaxique abstrait et le transforme en une représentation correspondant au langage intermédiaire de Tom. Au cours de cette transformation, les instructions Tom sont compilées dans un langage intermédiaire (noté IL) et composé d'instructions basiques comme l'affectation, les conditionnelles et les boucles. Ce langage intermédiaire est ensuite traduit en langage hôte lors de l'étape de génération de code effectuée par le *backend*. Cette étape consiste simplement à traduire chaque instruction du code intermédiaire en une instruction du langage hôte. Par exemple, lorsque Tom est utilisé avec Java, l'instruction de séquence du langage intermédiaire est traduite par le " ;" de Java.

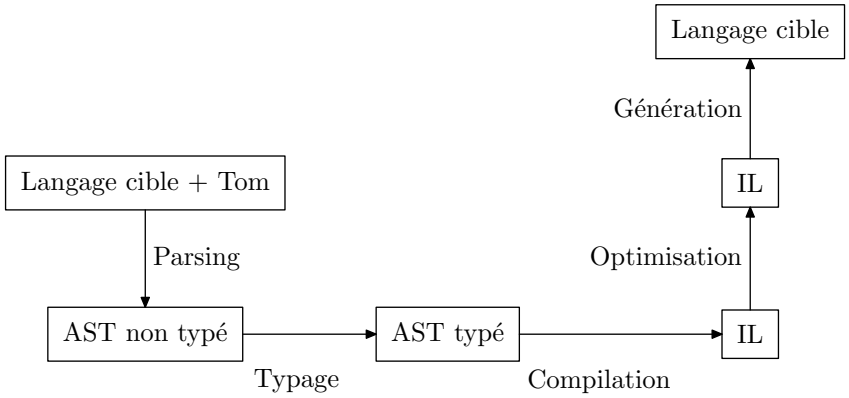


FIG. 2.2: Chaîne de compilation de Tom

Les différents arbres syntaxiques abstraits sont représentés sous forme de termes algébriques qui sont transformés par réécriture tout le long de la chaîne de compilation.

2.3 Apports de cette thèse au langage Tom

Le but de cette thèse était de proposer un langage dédié à l'analyse et la transformation de programmes en se fondant sur le langage Tom déjà existant. L'ensemble des propositions de cette thèse a donc pu être mis en pratique au travers d'extensions et d'améliorations de ce langage. Nous présentons ici brièvement trois apports.

2.3.1 Un langage de stratégies plus expressif

Au début de cette thèse, le langage de stratégies de Tom offrait la possibilité de contrôler l'application des règles. Durant cette thèse, nous avons proposé un nouveau langage plus expressif permettant entre autres de considérer la notion de position dans un terme comme un objet de premier ordre. Ce nouveau langage de stratégies est caractérisé par :

- la réification de l'environnement d'exécution permettant de connaître le contexte d'évaluation (en particulier la position dans le terme global) mais aussi de définir des parcours plus complexes comme ceux dans les graphes,
- une forte extensibilité due à son approche compositionnelle,

De plus, son implémentation apporte :

- la possibilité de paramétrer les stratégies par des objets du langage hôte,
- l'indépendance du langage par rapport aux structures de données grâce à son intéraction avec le langage Tom.

Ces caractéristiques permettent en plus du contrôle des règles de définir des analyses complexes sur des arbres ou des graphes en collectant de l'information lors du parcours de la structure. Ce langage et son implémentation seront présentés en détail dans le chapitre 4.

2.3.2 Réécriture de termes-graphes

Afin de faciliter l'implémentation d'analyses, une des contributions de cette thèse est de proposer une extension du langage Tom permettant de manipuler des structures de termes-graphes. La réécriture de termes-graphes dont le concept sera présenté en détail dans le chapitre 5 est une généralisation de la réécriture de termes autorisant le partage de sous-termes et les cycles. Ce type de structures de données permet de représenter et de transformer des graphes de flot.

Dans un premier temps, pour limiter l'impact de cette extension sur l'implémentation du langage Tom, nous avons choisi d'intégrer les termes-graphes au niveau du générateur de structures de données. L'idée principale est d'étendre automatiquement une signature algébrique de termes pour construire des termes-graphes sur cette même signature et décrire des systèmes de règles de termes-graphes respectant la sémantique de la réécriture de termes-graphes. En plus de cette extension au niveau du générateur de structures, nous avons proposé plusieurs extensions du langage de stratégies permettant de définir des parcours spécifiques aux graphes. L'ensemble de ces apports sera présenté dans le chapitre 5.

2.3.3 Manipulation de programmes Bytecode Java

Pour expérimenter nos extensions dans le cadre de l'analyse et la transformation de programmes, nous avons proposé une bibliothèque et une signature algébrique permettant de manipuler facilement des programmes Bytecode. Ainsi il est possible de filtrer des programmes écrits en Bytecode, de les transformer puis de regénérer le Bytecode après transformation. Un programme Bytecode est donc représenté par un terme algébrique Tom. Le principal avantage de cette approche est de permettre aux utilisateurs de décrire leurs transformations de manière déclarative (sous forme de règles et de stratégies) et ainsi d'améliorer la confiance en leurs programmes. Cette bibliothèque a été utilisée dans le cadre de plusieurs projets présentés dans le chapitre 6.

3 Un cadre théorique pour les langages îlots

Contexte En informatique, on distingue deux types de langages de programmation : les langages généralistes assez expressifs pour permettre d'implémenter n'importe quelle fonction calculable (autrement dit Turing-complets) et les langages dédiés à un domaine d'application (notés DSLs) dont la vocation est d'offrir les bonnes abstractions pour un domaine particulier. Dans cette thèse, nous nous sommes intéressés à un sous-ensemble de ces langages : les langages dédiés embarqués (notés EDSLs) qui se distinguent par le fait d'être embarqués dans un langage généraliste. Ils profitent ainsi de toute l'expressivité des langages généralistes (en particulier toutes leurs bibliothèques et outils de programmation) tout en proposant des primitives spécifiques à leur domaine d'application. Nous appelons ici ce type de langages des *langages îlots* ancrés dans un *langage océan* correspondant au langage généraliste.

Contributions Ce chapitre propose un cadre formel pour définir un langage îlot. Ce cadre définit la combinaison des syntaxes et sémantiques des deux langages permettant de déterminer quelles propriétés sont nécessaires pour préserver la sémantique du langage embarqué. La notion d'*îlot formel* est utilisée pour décrire des extensions correctement définies. Dans l'expression *îlot formel*, le mot *formel* se réfère aux notions de clarté et de précision de la définition du langage îlot. On peut ainsi plus facilement vérifier des propriétés spécifiques au domaine d'application de l'îlot. L'utilisation de ce type de formalisme permet donc de certifier le développement de cette catégorie de langages. Ce formalisme a été publié dans [BKM06].

Plan du chapitre Après une introduction sur les langages dédiés dans la section 3.1, nous présentons le formalisme de langages îlots dans la section 3.2. Au travers d'exemples réels de langages dédiés embarqués, nous montrons comment ce formalisme peut s'instancier. Puis dans la section 3.3, nous définissons les propriétés nécessaires à un langage îlot pour conserver ses propriétés structurelles après dissolution. Enfin, dans la section 3.4, nous instancions le formalisme des îlots formels par le langage Tom.

3.1 Les langages dédiés

Un langage dédié est un langage de programmation spécifique à un domaine d'application et donc offrant les bonnes abstractions de ce domaine à l'utilisateur. Dans ce sens, certaines bibliothèques dédiées à un domaine comme les *toolkits* graphiques répondent à la même problématique. Le principe des DSLs est d'aller encore plus loin en utilisant une syntaxe propre au domaine, offrant ainsi un code plus concis. On peut parfois aussi

vouloir se détacher complètement de la syntaxe d'un langage de programmation et permettre ainsi à des experts du domaine non informaticiens de *formaliser* leur expertise. Cela peut conduire par exemple à des langages comme les formulaires de *Visual Basic* ou les *business rules* d'ILOG. Il existe de nombreux travaux présentant des langages dédiés dans des domaines variés et la bibliographie annotée [DKV00] donne une vue d'ensemble des applications possibles de ce type de langages. Les principales caractéristiques d'un langage dédié sont donc une taille réduite ainsi que des notations et des abstractions appropriées à un domaine. Enfin, un code déclaratif est souvent privilégié à un code impératif.

Prenons l'exemple de la commande Unix *make*, qui permet de déterminer automatiquement quelles parties d'un programme doivent être recompilées et les commandes associées. Le langage des fichiers *Makefile* est simple et largement déclaratif. L'utilisateur n'a qu'à exprimer les dépendances de fichiers, tous les détails d'implémentation comme la date de dernière modification sont cachés et certaines règles de dépendance sont parfois implicites. *make* est donc un exemple probant de DSL offrant les bonnes abstractions liées à la construction d'un programme.

Faciliter la programmation et la maintenance du code Lorsque les abstractions et les notations sont correctement choisies, un programme DSL répondant à un problème du domaine est naturellement plus concis et plus lisible qu'un programme équivalent dans un langage généraliste. Cela permet de minimiser le temps de développement et de maintenance. Dans le cas de *make* par exemple, il est évident qu'écrire un script qui réaliserait la même tâche serait laborieux et sujet à de nombreuses erreurs.

Favoriser l'analyse de code Un autre avantage important de l'approche par DSL est la possibilité de vérifier plus facilement des propriétés du code. De part la spécificité de leur domaine d'application, leur sémantique est souvent assez restreinte pour permettre de réaliser des analyses liées au domaine. Par exemple, dans le cas de *make*, il est possible de détecter statiquement les dépendances cycliques et ainsi d'assurer la terminaison du programme.

3.1.1 Patrons de conception pour les langages dédiés

La conception d'un langage dédié est une tâche difficile et il existe de nombreuses approches principalement liées aux contraintes du domaine. Plusieurs classifications ont été proposées afin de mieux comprendre les motivations et les limites de chacune de ces conceptions. Nous présentons ici brièvement deux classifications :

- les patterns de Spinellis qui sont principalement liés à l'implémentation des DSLs,
- les patterns de Sloane qui sont classifiés suivant les phases de développement du langage.

Les patterns de Spinellis Dans la classification de [Spi01], Spinellis propose huit *design patterns* pour les DSLs. Comme dans le papier fondateur des *design patterns* [GHJV95], ils peuvent être de trois types différents :

- *structurels* qui s'intéressent aux intéractions entre le DSL et les autres langages (langages embarqués (*piggyback*), front-end),
- *comportementaux* qui s'intéressent à la structure globale d'un système faisant intervenir un ou plusieurs DSLs (pipeline),
- et *créationnels* qui décrivent la conception du DSL (traitement lexical, représentation de structures de données, transformation source à source, extension de langage, spécialisation de langage).

Dans cette présentation, les DSLs sont de réels langages de programmation caractérisés par une syntaxe et une sémantique propres. Leur mise en oeuvre correspond aux patterns créationnels. Cela peut aller d'un simple *traitement lexical* comme pour le préprocesseur C jusqu'à une *extension* (comme AspectJ [KHH+01]) ou à une *spécialisation* d'un langage généraliste (comme JavaLight [NO98]) entraînant de profondes modifications du compilateur ou de l'interpréteur. Le pattern créationnel *transformation source à source* propose une alternative plus légère : le programme DSL est traduit vers un programme équivalent dans un langage généraliste permettant ainsi de réutiliser tous les outils liés à ce langage sans avoir à les modifier. Le travail de Spinellis discute à travers des exemples historiques de l'intérêt de chacune de ces approches donnant ainsi une vue globale sur le développement logiciel impliquant des DSLs. En se limitant aux langages de programmation, ce travail ne traite pas des approches fondées directement sur un langage généraliste comme par exemple les *fluent interfaces* [Fow05].

Les patterns de Sloane Dans la classification de [MHS05], chacune des phases de développement d'un DSL est présentée : de la décision (*Pourquoi ?*) à l'implémentation (*Comment ?*). L'idée de cette approche est d'offrir des conseils pour les développeurs de DSLs et non d'offrir une taxinomie des DSLs comme celui de Spinellis.

Parmi les *design patterns* et *implementation patterns*, on retrouve l'ensemble des patterns proposés par Spinellis. Une comparaison est d'ailleurs faite entre les deux classifications. La classification proposée par Sloane, Heering et Mernik recouvre beaucoup plus de types de modes de conception de DSLs que celle de Spinellis (comme par exemple, l'implémentation embarquée proposée par la méta-programmation). De plus, les *decision patterns* qui ne sont pas du tout évoqués dans [Spi01] permettent aux futurs utilisateurs de les guider dans l'analyse de leurs besoins. Si certains besoins paraissent assez naturels (comme l'intégration de notations propres à un domaine ou encore l'automatisation de tâches), les auteurs proposent aussi le *parcours de structures de données complexes* comme un besoin pouvant entraîner la naissance d'un DSL. Ce besoin est en outre illustré par plusieurs langages dédiés au traitement du XML comme le langage S-XML. Dans cette thèse, le langage proposé pour prototyper des analyseurs statiques est fondé sur ce *decision pattern*. Tom offre en effet des primitives adaptées au parcours de structures arborescentes complexes comme les ASTs ou encore les graphes de flot de contrôle.

3.1.2 Cas particulier des langages îlots

Dans [Spi01], Spinellis propose le *piggyback pattern* qui correspond au *design* des langages embarqués ou îlots dans les langages généralistes. C'est ce type de *design* que

nous avons formalisé dans ce chapitre en adoptant la métaphore des îlots. L'intérêt de cette approche est de pouvoir profiter des capacités d'un langage généraliste comme par exemple de bibliothèques ou du support pour le *multithreading*. Dans ce cas, une implémentation possible est une *transformation source à source* préservant les parties écrites dans le langage généraliste. Dans [MHS05], les EDSLs correspondent dans la phase de conception au pattern *Language Exploitation* et dans la phase d'implémentation aux patterns *Embedding* et *Preprocessor*. On peut aussi se référer à [EFM00] qui propose une méthode de génération automatique de compilateurs optimisant pour les EDSLs à partir d'une spécification fonctionnelle des caractéristiques du langage embarqué. Une autre approche d'implémentation fondée sur une sémantique multi-niveaux et l'utilisation intensive des monades [Wad90] est proposée dans [SBP99].

Les exemples typiques d'EDSLs sont les langages Yacc et Lex puisque les actions associées aux règles de la grammaire (pour Yacc) et aux expressions régulières (pour Lex) sont directement écrites en C. De plus, le langage cible étant C, ces programmes sont compilés par transformation source à source en préservant le code des actions. Notons que la combinaison avec C est encore plus marquée pour Yacc puisqu'il est possible de référencer à l'aide de variables des parties de l'AST définies dans la partie gauche des règles puis d'y faire référence dans l'action C (le nom de la variable est alors préfixée par $).

Dans la section suivante, nous présentons un cadre formel pour les EDSLs afin de caractériser comment en terme de syntaxe et de sémantique, les liens entre l'EDSL et le langage généraliste peuvent se formaliser. En particulier, nous nous sommes intéressés aux propriétés nécessaires à la correction d'une telle combinaison. Les EDSLs répondant à ces critères sont appelés *îlots formels* dans le sens de bien définis. Si l'un des avantages des DSLs est de pouvoir vérifier facilement des propriétés liées au domaine, s'assurer de la conformité de leur implémentation est primordial.

3.2 Formalisation des langages îlots

Un programme écrit à l'aide d'un EDSL (comme une grammaire Yacc par exemple) est formé de parties écrites en langage généraliste (nous l'appellerons l'océan) et d'autres écrites en EDSL (les îlots). Ces deux langages, celui de l'océan (noté *ol*) et des îlots (noté *il*) ont chacun leur propre syntaxe et sémantique. Si l'on suppose que ces deux langages sont ainsi définis, comment exprimer formellement la combinaison des deux ? Nous appelons cette phase l'ancrage et comme pour un langage, il doit être défini en terme de syntaxe et de sémantique.

Le cycle de vie des îlots peut donc être défini en cinq parties :
- le *code existant* qui correspond à du code en langage généraliste,
- l'*ancrage* qui correspond à l'établissement de liens entre les deux langages,
- la *construction* qui correspond au développement du code EDSL,
- la *vérification* qui correspond à l'analyse de propriétés spécifiques au domaine sur ces îlots,
- la *dissolution* qui correspond à une transformation source à source de l'EDSL vers

le langage généraliste.

code existant ancrage construction vérification dissolution

FIG. 3.1: Cycle de vie des îlots de code

3.2.1 Préliminaires

Pour caractériser les langages océan et îlot, nous définissons la syntaxe d'un langage par une grammaire $\mathcal{G}$ et sa sémantique par une relation de réduction à grand pas à la Kahn [Kah87] que nous noterons *bs*.

Définition 24. *Une grammaire $\mathcal{G} = (A, N, T, R)$ est définie par :*
- *un ensemble fini N de symboles non terminaux,*
- *un ensemble fini T de symboles terminaux disjoint de N,*
- *un ensemble fini R de règles de production où chaque règle est de la forme $N \rightarrow (N \cup T)^*$,*
- *un symbole $A \in N$ représentant le symbole d'entrée ou axiome.*

Étant donné R un ensemble de règles de production, on définit $\mathsf{left}(R)$ comme l'ensemble des non-terminaux qui apparaissent en partie gauche d'une règle de R.

Définition 25. *Étant donnés une grammaire $\mathcal{G} = (A, N, T, R)$ et un ensemble de non terminaux $M \subseteq N$, on note $\mathcal{L}(\mathcal{G}, M)$ l'ensemble des séquences de terminaux (aussi appelées mots) engendrées à partir de M en appliquant les règles de production de R jusqu'à ce qu'aucun symbole non terminal n'apparaisse dans la séquence.*

Le langage d'une grammaire $\mathcal{G} = (A, N, T, R)$ noté $\mathcal{L}(\mathcal{G})$ est défini comme $\mathcal{L}(\mathcal{G}, \{A\})$, autrement dit l'ensemble des mots engendrés par l'axiome.

La suite de dérivations qui nous a amené à un mot du langage peut être représentée sous forme d'un arbre appelé *arbre de dérivation*. Si pour certains mots du langage $\mathcal{L}(\mathcal{G})$, il y a plusieurs arbres de dérivation possibles alors la grammaire est dite *ambiguë*. Nous considérerons par la suite uniquement des grammaires non ambigües.

Définition 26. *On note $\mathsf{getRoot}$ la fonction qui associe à chaque mot $m \in \mathcal{L}(\mathcal{G}, M)$ le non terminal qui l'engendre. Ce non terminal correspond à la racine de l'arbre de dérivation de m.*

Définition 27. *Soit $\mathcal{O}$ l'ensemble des objets manipulés par un langage, un environnement ϵ est une fonction de $\mathcal{X}$ vers $\mathcal{O}$ où $\mathcal{X}$ est l'ensemble des variables. L'ensemble des environnements est noté $\mathcal{E}nv$.*

Définition 28. *La relation de réduction d'une sémantique à grand pas est une relation dont le domaine est un sous-ensemble des couples $\langle \epsilon, i \rangle$ (formés d'un environnement ϵ et d'une séquence i de terminaux) et le codomaine est un sous-ensemble de $\mathcal{E}nv$.*

Une sémantique à grand pas est généralement définie par un ensemble de règles d'inférence que nous noterons $\mathcal{R}$.

3.2.2 Ancrage syntaxique

D'un point de vue syntaxique, l'ancrage d'un îlot consiste à placer un mot îlot à l'intérieur d'un mot océan. Au niveau des grammaires, la syntaxe de l'ancrage consiste à associer des non-terminaux *ol* à des non-terminaux *il*.

Si l'on note *oil* le langage correspondant à la combinaison de *ol* et *il*, la syntaxe de *oil* est fonction de :
- la grammaire de *ol* notée $\mathcal{G}_{ol}$,
- la grammaire de *il* notée $\mathcal{G}_{il}$,
- l'ancrage syntaxique noté anch.

Définition 29. *Étant données deux grammaires $\mathcal{G}_{ol}$ et $\mathcal{G}_{il}$ dont les terminaux sont disjoints et notées :*

$$\mathcal{G}_{ol} = (A_{ol}, N_{ol}, T_{ol}, R_{ol}) \text{ et } \mathcal{G}_{il} = (A_{il}, N_{il}, T_{il}, R_{il}),$$

on définit deux types d'ancrage syntaxique :
- *un* ancrage syntaxique simple *est une relation $\mathsf{anch}(\mathcal{G}_{ol}, \mathcal{G}_{il}) \subset N_{ol} \times N_{il}$ telle que $N_{ol} \cap N_{il}) = \emptyset$,*
- *un* ancrage syntaxique avec lacs *est une relation $\mathsf{anch}(\mathcal{G}_{ol}, \mathcal{G}_{il}) \subset N_{ol} \times N_{il}$ telle que $(N_{ol} \cap N_{il}) \neq \emptyset \wedge (N_{ol} \cap \mathsf{left}(R_{il})) = \emptyset$.*

La combinaison des deux grammaires est notée $\mathcal{G}_{oil}$ et est définie comme suit :

$$\mathcal{G}_{oil} = (A_{ol}, N_{ol} \cup N_{il}, T_{ol} \cup T_{il}, R_{ol} \cup R_{il} \cup \mathsf{anch}(\mathcal{G}_{ol}, \mathcal{G}_{il}))$$

Ilots isolés Pour les ancrages syntaxiques simples, la condition $T_{ol} \cap T_{il} = \emptyset \wedge N_{ol} \cap N_{il} = \emptyset$ assure qu'il n'y a pas de conflit entre les deux grammaires. En pratique, les instructions îlots sont généralement isolés par un token spécial qui n'est pas autorisé dans le langage océan.

Exemple 2. Considérons les deux grammaires $\mathcal{G}_{ol} = (A, \{A\}, \{a\}, \{(A ::= a), (A ::= Aa)\})$ et $\mathcal{G}_{il} = (B, \{B\}, \{b\}, \{(B ::= b)\})$. Le langage $\mathcal{L}(\mathcal{G}_{ol})$ est l'ensemble des séquences `a`, `aa`, `aaa`, ... Le langage $\mathcal{L}(\mathcal{G}_{il})$ contient uniquement `b`. En considérant l'*ancrage syntaxique simple* $\mathsf{anch}(\mathcal{G}_{ol}, \mathcal{G}_{il}) = \{(A ::= B)\}$ on peut définir le langage $\mathcal{L}(\mathcal{G}_{oil})$ qui est composé des mots `a`, `b`, `aa`, `ba` et plus généralement de toutes séquences formées de `a` ou `b` et terminant par `a`.

Ilots avec lacs Dans certains cas, il peut être intéressant d'embarquer du code océan dans le code îlot sans qu'il soit modifié pendant la phase de dissolution. C'est typiquement le cas des actions dans Yacc et Lex. En terme d'ancrage, cela signifie que la grammaire *il* peut contenir des non-terminaux *ol*. Cela correspond aux ancrages syntaxique avec lacs. Finalement la condition qui vérifie l'absence de conflit est assouplie en $T_{ol} \cap T_{il} = \emptyset \wedge N_{ol} \cap \mathsf{left}(R_{il}) = \emptyset$.

Exemple 3. Pour illustrer la notion d'ancrage avec lacs, nous allons considérer comme langage *ol* un langage simple permettant de manipuler des tableaux d'entiers. Le langage *il* considéré permettra de manipuler des listes d'entiers. La structure d'entiers sera partagée entre les deux langages, d'où la nécessité de lacs.

L'ensemble des objets manipulés par le langage *ol* est donc composé de l'ensemble des entiers $\mathbb{N}$ et de l'ensemble des tableaux. Un entier $i \in \mathcal{O}_{ol}$ est représenté par sa valeur et un tableau $t \in \mathcal{O}_{ol}$ en tant qu'objet du langage est représenté par la notation $[o_1, \ldots, o_n]$ où $o_1, \ldots, o_n$ sont des objets de type entier. L'ensemble $\mathcal{O}_{il}$ correspond à l'ensemble des listes d'entiers. Une liste $l \in \mathcal{O}_{il}$ est représentée par la notation $(o_1, \ldots, o_n)$ où $o_1, \ldots, o_n$ sont des objets *ol* de type entier. On considère que les deux langages partagent le même ensemble de variables $\mathcal{X}$. Pour simplifier, on utilise directement les ensembles $\mathbb{N}$ et $\mathcal{X}$ dans la définition des deux grammaires. Comme l'ensemble $\mathcal{X}$ est inclus à la fois dans les terminaux du langage *ol* et du langage *il*, on peut remarquer que $T_{ol} \cap T_{il} \neq \emptyset$. Cependant, comme la combinaison des deux grammaires n'est pas ambiguë, cela ne pose pas de problème de conflit par la suite.

Les grammaires des deux langages sont données Figure 3.2.

Dans *ol*, la construction `int t[n]` permet d'allouer un tableau `t` de taille `n` et de le remplir avec des `0`. Étant donné un tableau `t` et un entier `i`, l'instruction `x=t[i]` assigne à la variable `x` la valeur entière du `i`-ième élément de `t`. De manière symétrique, l'instruction `t[i]=e`, permet de modifier la valeur du `i`-ième élément de `t` avec la valeur de l'expression `e`.

Dans *il*, les structures de données sont des listes qui sont définies de manière classique par deux constructeurs *nil* et *cons*. Ce langage définit des îlots avec lacs puisque le non-terminal $\langle expr \rangle$ utilisé dans la définition de *cons* provient de la grammaire $\mathcal{G}_{ol}$.

Pour connecter les deux langages, nous définissons l'ancrage suivant :

$$\mathsf{anch} = \left\{ \begin{array}{c} (\langle instr \rangle ::= \langle instruction \rangle), \\ (\langle expr \rangle ::= \langle expression \rangle) \end{array} \right\}$$

En utilisant la grammaire définie dans la Figure 3.2, le programme `int t[2] ; t[1]=3 ; t[2]=7` contenant uniquement des instructions du langage océan est valide. On peut alors étendre cette séquence d'instructions avec des instructions du langage îlot comme par exemple `l←cons(t[1],cons(t[2],nil)) ; i=head(l)`. On notera que les expressions `t[1]` et `t[2]` sont des lacs dans l'îlot `l←cons(...)`.

3.2.3 Ancrage sémantique

Après avoir défini l'ancrage d'un point de vue syntaxique, voyons comment exprimer le lien sémantique entre les deux langages. Ce lien est d'autant plus important que les *objets* manipulés par les deux langages sont souvent partagés. L'évaluation d'un îlot entraîne la plupart du temps un enrichissement de l'environnement d'exécution du code océan. Même si ces objets ne sont pas du même type, il existe une relation entre les deux structures. Avant de définir la sémantique de la combinaison des deux langages, nous commençons par préciser ce lien.

Le langage océan

$$
\begin{array}{lcl}
\langle instr\rangle & ::= & \langle instr\rangle ; \langle instr\rangle \\
& | & int\ \langle vararray\rangle[\langle int\rangle] \\
& | & \langle vararray\rangle[\langle int\rangle] = \langle expr\rangle \\
& | & \langle varint\rangle = \langle expr\rangle \\
\langle expr\rangle & ::= & \langle int\rangle \\
& | & \langle varint\rangle \\
& | & \langle vararray\rangle[\langle int\rangle] \\
\langle vararray\rangle & ::= & x \in \mathcal{X} \\
\langle varint\rangle & ::= & x \in \mathcal{X} \\
\langle int\rangle & ::= & i \in \mathbb{N}
\end{array}
$$

Le langage îlot

$$
\begin{array}{lcl}
\langle instruction\rangle & ::= & \langle varlist\rangle \leftarrow \langle list\rangle \\
\langle list\rangle & ::= & nil \\
& | & cons(\langle expr\rangle, \langle list\rangle) \\
\langle varlist\rangle & ::= & x \in \mathcal{X} \\
\langle expression\rangle & ::= & head(\langle varlist\rangle)
\end{array}
$$

L'ancrage syntaxique

$$
\begin{array}{lcl}
\langle instr\rangle & ::= & \langle instruction\rangle \\
\langle expr\rangle & ::= & \langle expression\rangle
\end{array}
$$

FIG. 3.2: Syntaxe de la combinaison des deux langages *ol* et *il*

Représentation des structures de données Pour définir la sémantique du langage étendu, nous devons définir le lien entre les objets du langage îlot et leur représentation dans l'océan. Cet ancrage n'est pas défini en fonction des deux systèmes de types (certains langages îlots n'ont pas forcément de systèmes de types) mais d'une manière plus directe entre les objets des deux langages par des *fonctions de représentation et d'abstraction.* Cette notion a été introduite à l'origine dans le cadre de la théorie du raffinement de données [RE98, ASVO05], utilisée pour convertir un modèle abstrait de données (comme les listes) en une structure de données implémentables (comme les tableaux). Dans notre formalisme, les îlots sont considérés comme abstraits relativement à l'océan.

Définition 30. *Étant donné un ensemble d'objets du langage îlot noté $\mathcal{O}_{il}$ et un ensemble d'objets du langage océan noté $\mathcal{O}_{ol}$, une* fonction de représentation $\lceil\ \rceil$ *est une fonction totale injective de $\mathcal{O}_{il}$ vers $\mathcal{O}_{ol}$ et une* fonction d'abstraction $\lfloor\ \rfloor$ *est une fonction surjective (potentiellement partielle) de $\mathcal{O}_{ol}$ vers $\mathcal{O}_{il}$ telles que $\lfloor\ \rfloor.\lceil\ \rceil = Id_{\mathcal{O}_{il}}$.*

Exemple 4. Dans l'exemple 3, chaque liste du langage îlot peut être représentée par un tableau dans le langage océan contenant exactement les mêmes entiers dans le même ordre. Nous notons cette fonction de représentation map_1. La fonction map_1^{-1} peut tout simplement être choisie comme fonction d'abstraction. Le second type d'objets manipulés par le langage îlot sont les entiers et leur représentation est la même dans les deux langages.

Propriétés structurelles Dans la Définition 30, les fonctions de représentation et d'abstraction ont été introduites pour établir une correspondance entre les structures de données des îlots et leur représentation dans le langage océan. Pour l'instant, nous n'avons posé aucune contrainte sur la représentation de ces objets. Par exemple, ces fonctions

peuvent ne pas préserver les propriétés structurelles des objets. La fonction d'abstraction peut par exemple entraîner une perte d'information. Seul un ensemble de propriétés est préservé entre les deux mondes.

Pour assurer la préservation des propriétés structurelles, nous introduisons pour chaque langage un ensemble de prédicats dénotés respectivement $\mathcal{P}_{ol}$ et $\mathcal{P}_{il}$ définissant les propriétés structurelles des objets des deux mondes. Pour tout prédicat p, $ar(p)$ correspond au nombre d'arguments de p. À partir de ces deux ensembles, nous définissons l'*ancrage axiomatique*.

Définition 31. *Étant donné un ensemble de prédicats îlots $\mathcal{P}_{il}$ et un ensemble de prédicats $\mathcal{P}_{ol}$, un* ancrage axiomatique *(également appelé ancrage de prédicats) ϕ est une fonction injective de $\mathcal{P}_{il} \rightarrow \mathcal{P}_{ol}$ telle que $\forall p \in \mathcal{P}_{il}, ar(p) = ar(\phi(p))$.*

Cette fonction est étendue par morphisme aux formules du premier ordre, en utilisant la fonction de représentation :

$$\forall p \in \mathcal{P}_{il}, \forall t_1, \ldots, t_n \in \mathcal{O}_{il} \cup \mathcal{X}_{il}, \phi(p(t_1, \ldots, t_n)) = \phi(p)(\lceil t_1 \rceil, \ldots, \lceil t_n \rceil)$$

$$\begin{array}{l|l} \phi(\forall x\ P) = \forall x\ \phi(P), & \phi(\exists x\ P) = \exists x\ \phi(P), \\ \phi(P_1 \vee P_2) = \phi(P_1) \vee \phi(P_2), & \phi(P_1 \wedge P_2) = \phi(P_1) \wedge \phi(P_2), \\ \phi(\neg P) = \neg\phi(P), & \phi(P_1 \rightarrow P_2) = \phi(P_1) \rightarrow \phi(P_2). \end{array}$$

Ancrage formel

Définition 32. *Étant donné un ancrage axiomatique ϕ, une fonction de représentation $\lceil\ \rceil$ est* ϕ-formelle *si $\forall p \in \mathcal{P}_{il}, \forall o_1, \ldots, o_n \in \mathcal{O}_{il}$ où $n = ar(p)$*

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

Exemple 5. Considérons les relations d'égalité $=_{il}$ et $=_{ol}$ comme un exemple de prédicats respectivement définis sur les listes et les tableaux de l'exemple 3, et l'ancrage axiomatique $\phi_1 = \{(=_{il}, =_{ol})\}$. La fonction de représentation map_1, introduite dans l'exemple 4, est *ϕ_1-formelle* puisque deux listes sont égales avec $=_{il}$ (composées des mêmes entiers dans le même ordre) si et seulement si leurs représentations sont égales avec $=_{ol}$.

Considérons maintenant la fonction de représentation map_2 qui associe une liste à un tableau mais dont les éléments sont inversés. Soient deux prédicats eq_{head} pour les listes et eq_{elt} pour les tableaux définis tels que :

- $eq_{head}(l, l') \equiv (head(l) = head(l'))$,
- $eq_{elt}(t, t') \equiv (t[0] = t'[0])$

Étant donné l'ancrage axiomatique $\phi_2 = \{(eq_{head}, eq_{elt})\}$, la fonction de représentation map_2 n'est pas *ϕ_2-formelle* car nous pouvons construire deux listes $l_1 = (1, 2), l_2 = (1, 3)$ telles que $eq_{head}(l_1, l_2)$ est vrai mais pas $eq_{elt}(map_2(l_1), map_2(l_2))$, autrement dit $eq_{elt}([2, 1], [3, 1])$ est faux.

Combinaison des sémantiques Étant donnée une fonction de représentation $\lceil\ \rceil$ et une fonction d'abstraction $\lfloor\ \rfloor$, nous pouvons simuler le comportement des îlots dans l'environnement *ol*. Supposons que nous ayons deux sémantiques à grand pas (avec leurs relations de réduction respectives bs_{ol} et bs_{il} dans des ensembles d'environnements distincts $\mathcal{E}nv_{ol}$ et $\mathcal{E}nv_{il}$).

Pour définir l'évaluation d'un îlot dans un environnement $\epsilon_{ol} \in \mathcal{E}nv_{ol}$, nous avons besoin de traduire ϵ_{ol} dans un environnent $\epsilon_{il} \in \mathcal{E}nv_{il}$. Pour cela, il suffit d'étendre les fonctions de représentation et d'abstraction aux variables puis aux environnements.

Définition 33. *La fonction de représentation (resp. d'abstraction) est étendue aux variables et notée* $\lceil\ \rceil \in \mathcal{X}_{il} \to \mathcal{X}_{ol}$ *(resp.* $\lfloor\ \rfloor \in \mathcal{X}_{ol} \to \mathcal{X}_{il}$*). Elle vérifie encore* $\lfloor\ \rfloor.\lceil\ \rceil = Id_{\mathcal{X}_{il}}$.

Dans la plupart des cas, pour éviter toute confusion, le langage îlot et le langage océan partagent le même ensemble de variables $\mathcal{X}$ et les fonctions de représentation et d'abstraction sont réduites à l'identité sur les variables. Cela signifie en particulier que dans ce cas, toute variable du langage océan représentant un objet du langage îlot est visible dans tous les îlots contenus à l'intérieur de sa portée.

Exemple 6. Dans le cas de l'exemple 3, comme les deux langages partagent le même ensemble de variables $\mathcal{X}$, nous choisissons pour les deux fonctions $\lceil\ \rceil, \lfloor\ \rfloor \in \mathcal{X} \to \mathcal{X}$ la fonction $Id_{\mathcal{X}}$. La condition $\lfloor\ \rfloor.\lceil\ \rceil = Id_{\mathcal{X}_{il}}$ est donc trivialement vérifiée.

Exemple 7. Dans le cas du langage Tom, même si l'ensemble des noms de variables est inclus dans celui de Java, les fonctions de représentation et d'abstraction des variables ne sont pas réduites à l'identité. En effet, toute variable Tom `x` est traduite en une variable Java `tom_x`. Le compilateur de Tom suppose alors que les programmes respectent une convention d'hygiène *à la* Barendregt (autrement dit, une variable n'apparaît jamais à la fois libre et liée) dans le sens où le code Java environnant ne doit pas déclarer de variables préfixées par `tom_`. Ainsi tout programme îlot directement ancré dans Java (et non dans un lac comme en partie droite d'une instruction `%match`) est évalué dans un environnement vide. Seuls les îlots ancrés dans un lac peuvent accéder à des variables Tom. Par exemple, considérons le programme Tom suivant :

```
Term t = `f(a);
int x = 0;
%match(t) {
  f(x) -> {
    x++;
    return `g(x);
  }
}
```

Il est possible de déclarer une variable Java `x` puis dans un îlot de déclarer une variable Tom du même nom dans un motif. Dans la partie droite de la règle, le code correspond à un lac Java. La variable Java `x` est donc de nouveau visible (instruction `x++;`). Si dans ce lac, on définit un nouveau îlot (ici, `` `g(x) ``) alors la variable Java est cachée et `x`

est évaluée comme la variable Tom déclarée dans le motif. Ce choix de conception dans Tom n'est pas une bonne pratique car l'introduction des îlots ne préserve pas les règles de visibilité du langage hôte. Pour la prochaine version de Tom, il est donc envisagé de conserver les noms de variables Tom lors de la compilation.

Définition 34. *La fonction de représentation (resp. d'abstraction) est étendue aux environnements et notée* $\lceil\ \rceil \in \mathcal{E}nv_{il} \rightarrow \mathcal{E}nv_{ol}$ *(resp.* $\lfloor\ \rfloor \in \mathcal{E}nv_{ol} \rightarrow \mathcal{E}nv_{il}$*). Elle est définie telle que* $\forall \epsilon_{il} \in \mathcal{E}nv_{il}, \lceil \epsilon_{il} \rceil = \{(\lceil x \rceil, \lceil v \rceil) \mid (x, v) \in \epsilon_{il}\}$ *(resp.* $\forall \epsilon_{ol} \in \mathcal{E}nv_{ol}, \lfloor \epsilon_{ol} \rfloor = \{(\lfloor x \rfloor, \lfloor v \rfloor) \mid (x, v) \in \epsilon_{ol}, x \in \mathcal{D}om(\lfloor\ \rfloor), v \in \mathcal{D}om(\lfloor\ \rfloor)\}$*).*

Notons que pour les environnements, $\lfloor\ \rfloor$ est totale car dans le cas où aucun couple n'a de représentation au niveau îlot, la fonction retourne l'ensemble vide.

Nous pouvons maintenant simuler la réduction d'un îlot dans un environnement *ol* avec la relation de réduction de *il* en traduisant l'environnement *ol* avec $\lfloor\ \rfloor$. Les règles de sémantique de *ol* et *il* étendues avec deux règles de traduction donnent une sémantique au langage *oil*.

Définition 35. *Soient deux sémantiques* bs_{il} *et* bs_{ol} *respectivement définies par les ensembles de règles d'inférence* $\mathcal{R}_{il}$ *et* $\mathcal{R}_{ol}$*, la sémantique de oil est définie par l'ensemble des règles d'inférence* $\mathcal{R}_{oil} = \mathcal{R}_{ol} \cup \mathcal{R}'_{il} \cup \{r_1, r_2\}$ *où :*

- $\mathcal{R}'_{il} = \mathcal{R}_{il}$ *où* $\langle \epsilon, i \rangle \mapsto_{bs_{il}} \epsilon'$ *est remplacé par* $\langle \epsilon, \delta, i \rangle \mapsto_{bs_{il}} \langle \epsilon', \delta \rangle$ *(*$\delta \in \mathcal{E}nv_{ol}$ *et* $\epsilon, \epsilon' \in \mathcal{E}nv_{il}$*),*
- *les règles d'inférence* r_1 *et* r_2 *sont définies comme suit :*

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

avec $\gamma(\epsilon) = \epsilon - \lceil \lfloor \epsilon \rfloor \rceil$ *représente les éléments de l'environnement océan* ϵ *qui ne représentent pas d'objets îlot.*

Les règles d'inférence r_1 et r_2 relient les deux sémantiques : r_1 fait le pont de la sémantique *ol* à la sémantique *il* pour l'évaluation des îlots et r_2, inversement, relie *il* vers *ol* pour l'évaluation des lacs.

Dans la règle r_1, la fonction $\lfloor\ \rfloor$ calcule l'environnement *il* restreint aux couples ayant des représentations au niveau îlot, ainsi l'îlot est évalué dans la sémantique *il*. Nous obtenons ensuite, après évaluation, un nouvel environnement *il* qui peut être traduit en un environnement *ol* à l'aide de la fonction $\lceil\ \rceil$. Les objets qui ne peuvent pas être représentés dans l'îlot ($\gamma(\epsilon) = \epsilon - \lceil \lfloor \epsilon \rfloor \rceil$) sont donnés comme paramètre à la fonction d'évaluation de l'îlot en cas de lacs. C'est pour cela que l'environnement *ol* obtenu après évaluation d'un îlot correspond à l'union de la partie de ϵ qui n'est pas représentable dans l'îlot mais qui peut avoir été modifiée par les lacs (qui correspond à δ) et le résultat de l'évaluation des instructions *il* (qui correspond à $\lceil \epsilon' \rceil$). L'introduction de δ dans les règles de la sémantique *il* est donc nécessaire pour la réduction des lacs. En effet, il est ainsi possible de garder une trace de l'environnement *ol* lors de l'évaluation de l'îlot. Sans ce contexte, les lacs seraient complètement séparés du code océan environnant et

aucune référence aux variables du langage océan définis avant l'îlot ne pourraient être définie dans les lacs.

Remarquons aussi que dans r_1, $\lceil\epsilon'\rceil \cup \delta$ est une fonction seulement si $\texttt{dom}(\lceil\epsilon'\rceil) \cap \texttt{dom}(\delta) = \emptyset$. La fonctionnalité de cette relation signifie qu'une variable ne peut pas représenter à la fois un objet du langages îlot et océan. La règle r_2 requiert donc une condition similaire. Nous considérerons par la suite que les sémantiques vérifient ces conditions. En pratique, cela signifie que les îlots et les lacs introduisent des variables fraîches par rapport à l'environnement courant d'évaluation.

Dans la règle r_2, l'évaluation d'un lac est réalisée dans un environnement correspondant à l'union de la représentation de l'environnement îlot $\lceil\epsilon\rceil$ et du contexte δ afin de pouvoir utiliser dans le lac des objets du langage océan n'ayant pas d'abstraction dans le monde îlot. Une fois le lac évalué, l'environnement obtenu ϵ' est séparé entre les objets du langage océan représentant des objets îlots et le reste qui représentera le nouveau contexte. Dans la sémantique *il*, le couple obtenu est donc $\langle \lfloor\epsilon'\rfloor, \gamma(\epsilon')\rangle$.

Exemple 8. Dans l'exemple 3, on peut définir une sémantique pour chacun des langages. La sémantique du langage des tableaux est définie à l'aide de la relation de réduction $\mapsto_{itab}$ pour évaluer les instructions et d'une relation auxiliaire $\mapsto_{etab}$ pour évaluer les expressions. La sémantique du langage des listes est définie par les relations de réduction $\mapsto_{ilist}$ et $\mapsto_{elist}$. Au lieu de renvoyer un environnement, les relations de réduction auxilliaire $\mapsto_{etab}$ et $\mapsto_{elist}$ retournent une valeur (de type liste ou entier). Comme l'ancrage concerne à la fois les expressions et les instructions, il faut trois règles de connection :

- une règle pour la réduction des îlots de type instruction,
- une règle pour la réduction des îlots de type expression,
- une règle pour la réduction des lacs de type expression.

Dans cet exemple, il n'y a pas de lac de type instruction, il n'est donc pas nécessaire d'avoir une quatrième règle. La figure 3.3 présente l'ensemble des règles de sémantique de la combinaison de ces deux langages.

La figure 3.4 représente comment les sémantiques océan et îlot sont reliées par les fonctions de représentation et d'abstraction. En particulier, l'évaluation des îlots est réalisée lors de l'évaluation du programme océan environnant.

Pour conclure, la sémantique du langage *oil*, combinaison de *ol* et *il*, est fonction des deux sémantiques à grand pas bs_{ol} et bs_{il}, de la fonction de représentation $\lceil\ \rceil$ et de la fonction d'abstraction $\lfloor\ \rfloor$.

Nous allons maintenant utiliser ce formalisme d'îlots pour étendre les langages généralistes avec de nouvelles constructions. Au lieu d'étendre les compilateurs de ces langages pour prendre en compte ces nouvelles constructions, nous avons choisi de formaliser une approche *embarquée* comme dans [Spi01] où l'on *dissout* les îlots dans une phase de transformation source à source. Ainsi, le compilateur et tous les outils d'analyse du langage généraliste sont réutilisables.

Le langage océan

$$\frac{\langle \epsilon, i_1 \rangle \mapsto_{itab} \epsilon' \quad \langle \epsilon', i_2 \rangle \mapsto_{itab} \epsilon''}{\langle \epsilon, i_1 ; i_2 \rangle \mapsto_{itab} \epsilon''} \textit{ Séquence}$$

$$\frac{t \notin \mathcal{D}om(\epsilon)}{\langle \epsilon, int\ t[n] \rangle \mapsto_{itab} \epsilon[t \rightarrow [\underbrace{0, \ldots, 0}_{n}]]} \textit{ Déclaration tableau}$$

$$\frac{\langle \epsilon, e \rangle \mapsto_{etab} v \quad \epsilon(t) = [v_0, \ldots, v_n] \quad 0 \leq i \leq n}{\langle \epsilon, t[i] = e \rangle \mapsto_{itab} \epsilon[t \rightarrow [v_0, \ldots, v_{i-1}, v, v_{i+1}, \ldots, v_n]]} \textit{ Affectation tableau}$$

$$\frac{\langle \epsilon, e \rangle \mapsto_{etab} v}{\langle \epsilon, x = e \rangle \mapsto_{itab} \epsilon[x \rightarrow v]} \textit{ Affectation entier} \quad \frac{n \in \mathbb{N}}{\langle \epsilon, n \rangle \mapsto_{etab} n} \textit{ Lecture entier}$$

$$\frac{\epsilon(x) \in \mathbb{N}}{\langle \epsilon, x \rangle \mapsto_{etab} \epsilon(x)} \textit{ Lecture variable} \quad \frac{\epsilon(t) = [v_0, \ldots, v_n] \quad 0 \leq i \leq n}{\langle \epsilon, t[i] \rangle \mapsto_{etab} v_i} \textit{ Lecture tableau}$$

Le langage îlot

$$\frac{\langle \epsilon, \delta, e \rangle \mapsto_{elist} v}{\langle \epsilon, \delta, l \leftarrow e \rangle \mapsto_{ilist} \langle \epsilon[l \rightarrow v], \delta \rangle} \textit{ Déclaration et assignation liste}$$

$$\frac{\langle \epsilon, \delta, e \rangle \mapsto_{elist} v \quad \langle \epsilon, \delta, l \rangle \mapsto_{elist} () }{\langle \epsilon, \delta, cons(e, l) \rangle \mapsto_{elist} (v)} \textit{ cons (1)} \quad \frac{\langle \epsilon, \delta, e \rangle \mapsto_{elist} v \quad \langle \epsilon, \delta, l \rangle \mapsto_{elist} (v_1, \ldots, v_n)}{\langle \epsilon, \delta, cons(e, l) \rangle \mapsto_{elist} (v, v_1, \ldots, v_n)} \textit{ cons (2)}$$

$$\frac{}{\langle \epsilon, \delta, nil \rangle \mapsto_{elist} ()} \textit{ nil} \quad \frac{\epsilon(l) = (v_1, \ldots, v_n)}{\langle \epsilon, \delta, head(l) \rangle \mapsto_{elist} v_1} \textit{ head}$$

Les règles de combinaison

$$\frac{\langle \lfloor \epsilon \rfloor, \gamma(\epsilon), i \rangle \mapsto_{ilist} \langle \epsilon', \delta \rangle}{\langle \epsilon, i \rangle \mapsto_{itab} \lceil \epsilon' \rceil \cup \delta} \textit{ Îlot instruction} \quad \frac{\langle \lfloor \epsilon \rfloor, \gamma(\epsilon), e \rangle \mapsto_{elist} v}{\langle \epsilon, e \rangle \mapsto_{etab} \lceil v \rceil} \textit{ Îlot expression}$$

$$\frac{\langle \lceil \epsilon \rceil \cup \delta, e \rangle \mapsto_{etab} v}{\langle \epsilon, \delta, e \rangle \mapsto_{elist} \lfloor v \rfloor} \textit{ Lac expression}$$

FIG. 3.3: Combinaison des sémantiques des langages de tableaux et de listes

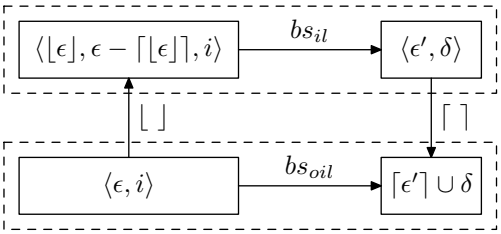


FIG. 3.4: Évaluation des îlots *il* dans la sémantique *ol*

3.2.4 Dissolution

La phase de *dissolution* correspond à une transformation source à source décrivant la traduction des îlots vers du code océan. À partir d'un programme *oil*, on obtient un programme *ol* en traduisant les parties *il* et en préservant les parties *ol*.

D'un point de vue syntaxique, cette phase consiste à traduire dans les mots *oil* tous les sous-mots *il* en *ol* afin d'obtenir un mot complètement engendré par la grammaire *ol*.

Définition 36. *Étant données deux grammaires $\mathcal{G}_{il}$ et $\mathcal{G}_{ol}$ et un ancrage syntaxique* anch*, la dissolution est définie comme une fonction totale* diss $: \mathcal{L}(\mathcal{G}_{oil}, \mathcal{D}om(\textsf{anch})) \rightarrow \mathcal{L}(\mathcal{G}_{ol}, \mathcal{R}an(\textsf{anch}))$.

Exemple 9. En reprenant un programme de l'exemple 3

```
int t[2];
t[0] = 3;
t[1] = 7;
l <- cons(t[0],cons(t[1],nil));
i = head(l)
```

On peut distinguer un îlot instruction `l←cons(t[0],cons(t[1],nil))`, un îlot expression `head(l)` et deux lacs expressions `t[0]` et `t[1]`. La dissolution de ces deux îlots pourrait résulter en le programme suivant :

```
int t[2];
t[0] = 3;
t[1] = 7;
int l[2];
l[0] = t[0];
l[1] = t[1];
y = l[0]
```

D'un point de vue sémantique, les sous-mots *ol* doivent s'évaluer de la même manière que les sous-mots *il* qu'ils remplacent. La figure 3.5 montre le lien entre l'évaluation

d'une instruction *il* dans la sémantique *il* (comme dans la figure 3.4) et l'évaluation de l'instruction *ol* correspondante (par dissolution) dans la sémantique *ol*.

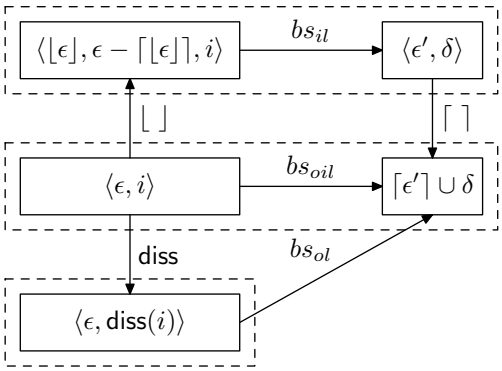


FIG. 3.5: Évaluation d'un îlot et de sa dissolution

Définition 37. *Étant données les fonctions de représentation et d'abstraction* $\lceil \ \rceil, \lfloor \ \rfloor$*, une fonction de dissolution* diss *est* bien-formée *si :*

- *le programme ol est syntaxiquement correct. Autrement dit,* diss *est définie telle que* $\forall i \in \mathcal{D}om(\mathsf{diss}), \mathsf{getRoot}(i) \in \mathsf{anch}(\mathsf{getRoot}(\mathsf{diss}(i)))$*,*
- *pour chaque mot* $i \in \mathcal{D}om(\mathsf{diss})$*, pour chaque environnement* $\epsilon \in \mathcal{E}nv_{il}$*, nous avons :* $\langle \epsilon, i \rangle \mapsto_{bs_{oil}} \epsilon' \Leftrightarrow \langle \lceil \epsilon \rceil, \mathsf{diss}(i) \rangle \mapsto_{bs_{ol}} \epsilon'$*,*

Exemple 10. La fonction de dissolution de l'exemple 3 peut être définie de la manière suivante :

$$
\begin{array}{lcl}
\mathsf{diss}(l \leftarrow nil) & \rightarrow & int\ l[0] \\
\mathsf{diss}(l \leftarrow cons(e_0, cons(\ldots, cons(e_n, nil)))) & \rightarrow & int\ l[n];\ l[0] = e_0;\ \ldots;\ l[n-1] = e_{n-1} \\
\mathsf{diss}(head(l)) & \rightarrow & l[0]
\end{array}
$$

Pour résumer, un langage îlot doit satisfaire les conditions suivantes :

Définition 38. *Étant donnés deux langages ol et il décrits par une grammaire, une sémantique à grand pas, un ensemble d'objets et un ensemble de prédicats sur ces objets, il est un langage îlot pour ol si il existe :*

- *un ancrage syntaxique* anch *(Définition 29),*
- *des fonctions* $\lceil \ \rceil$,$\lfloor \ \rfloor$ *(Définition 30),*
- *une fonction de dissolution* diss *(Définition 36).*

3.2.5 Exemples

Il existe dans la littérature de nombreux langages dédiés embarqués. Nous proposons d'instancier le formalisme d'îlots par quelques uns de ces langages illustrant des dépendances plus ou moins fortes avec le langage hôte : des systèmes de macros définis de manière purement syntaxique au langage SQLJ qui définit des fonctions de représentation et d'abstraction entre les types SQL et les types Java.

Les langages de macros Le but des langages de macros est de permettre d'automatiser l'utilisation de séquences d'instructions. Dans [BS02], les auteurs proposent une taxinomie des langages de macros. Ces langages sont formés de deux types de déclarations : la définition de la macro et son utilisation. Dans les langages compilés, l'évaluation des macros est réalisée par *expansion*. L'*expansion* de macros consiste à remplacer l'utilisation d'une macro par sa définition lors d'une phase précédant la compilation. Cette phase correspond exactement à la phase de dissolution dans le formalisme des îlots.

Les langages de macros peuvent être vus comme des cas très particuliers d'îlots. En effet, les fonctions de représentation et d'abstraction sont triviales étant donné qu'il n'y a pas de structure de données propre aux macros. Les macros permettent donc de définir des extensions de langage en terme de manipulation de données et non en terme de représentation de données. Ensuite, il n'y a pas d'ancrage syntaxique. L'identifiant d'une macro peut apparaître n'importe où dans le programme. Il n'y aucune contrainte particulière. Enfin, la fonction de dissolution d'une macro est la définition même de la macro et est locale à un programme. Nous appelons *îlots purement syntaxiques* ce type de langages îlots sans ancrage sémantique.

Linj [AML03] est une extension de Java permettant d'utiliser une syntaxe à la Lisp dans Java. L'exemple suivant permet par exemple de définir la fonction factorielle en Linj :

```
public class Test{
  /*linj
   defun fact (n)
     (if (= n 0)
      1
      (* n (fact (1- n)))))
   */
}
```

FIG. 3.6: Définition de la fonction factorielle en Linj

Dans le cas des îlots Linj pour Java, l'ancrage anch peut être défini comme suit :

$$\mathsf{anch} = \left\{ \ (\langle Declaration \rangle ::= \langle LispDeclaration \rangle) \ \right\}$$

$\langle LispDeclaration \rangle$ correspond aux définitions de fonctions comme dans l'exemple de `fact` mais aussi de classes et de méthodes dans le système objet de Common Lisp (CLOS) [KG89] ainsi que des macros. Le lecteur pourra se référer à [AML03] pour plus de détails sur l'expressivité de Linj.

Les fonctions de représentation et d'abstraction de Linj sont assez triviales puisque les îlots Linj peuvent accéder à n'importe quel type Java et aux méthodes associées. Nous sommes donc dans un cas particulier d'îlots où l'ensemble des objets est partagé

avec l'océan. Grâce à l'inférence de type de Linj, il n'est pas nécessaire à l'utilisateur de déclarer les types dans les îlots. Par exemple, dans la fonction `fact` précédente, le type de `n` a été inféré automatiquement.

La dissolution d'une déclaration de type `defun` correspond à une fonction statique Java portant le même nom. La figure 3.7 présente la fonction Java obtenue après dissolution du programme de l'exemple 3.6.

```
public static int fact(int n) {
   if (n == 0) {
     return 1;
   } else {
     return n * fact(n-1);
   }
}
```

FIG. 3.7: Dissolution du programme de l'exemple 3.6

Dans ce deuxième exemple, nous voyons comment le formalisme d'îlots peut aussi servir de base à des extensions de langage permettant la programmation multi-paradigmes. Une autre solution consiste à utiliser une machine virtuelle commune à plusieurs langages comme c'est le cas pour C # et F # avec `.NET` ou encore Scala et Java avec la JVM. Cependant, cette démarche de plate-forme générique permet plus de faire communiquer des programmes écrits dans différents langages que de réellement offrir de la programmation multi-paradigmes. L'avantage de l'approche par îlots est de mélanger les paradigmes directement au niveau syntaxique et donc de pouvoir profiter pleinement de tous les modèles de programmation dans un même programme.

SQLJ [EM98, ME00] est une extension de Java permettant de définir directement des requêtes SQL sans utiliser de bibliothèques telles que `jdbc`. Le principal avantage est de faciliter l'accès aux données d'une base sans avoir à connaître les détails de l'API `jdbc` et de permettre aussi de vérifier la syntaxe SQL, voire éventuellement d'optimiser le code au moment de la compilation.

Afin d'éviter tout conflit entre les deux grammaires, chaque îlot SQLJ commence par l'identifiant `#sql`, illégal en Java. L'exemple le plus simple d'îlot SQLJ consiste en une requête SQL précédé du mot-clé `#sql` et délimité par des accolades. Il est aussi possible d'affecter le résultat de la requête dans une variable Java comme illustré dans la figure 3.8. Cet îlot peut apparaître à la place de n'importe quelle instruction Java.

L'ancrage syntaxique anch peut donc être défini comme suit :

$$\mathsf{anch} = \left\{ \begin{array}{c} (\langle Declaration \rangle ::= \langle SqljDeclaration \rangle), \\ (\langle Statement \rangle ::= \langle SqljStatement \rangle) \end{array} \right\}$$

```
#sql iterator Blockbusters(String titre, int entrees)

public Blockbusters getBlockbusters(int seuil){
  Blockbusters resultat;
  #sql resultat =
    { SELECT TITRE,NB_ENTREES
      FROM FILMS
        WHERE NB_ENTREES >= :seuil
    }
  return resultat;
}
```

FIG. 3.8: Exemple d'îlot SQLJ

Type SQL	Type Java
INTEGER	int
VARCHAR	String
DECIMAL	java.math.BigDecimal
BIT	boolean
BINARY	byte[]
...	...

FIG. 3.9: Extrait de la table de conversion de types de SQLJ

Le non-terminal $\langle SqljDeclaration \rangle$ correspond aux déclarations permettant de créer des classes Java pour les itérateurs (correspondant aux ensembles de résultats dans `jdbc`) ou des déclarations de connexion à une base de données. Ils se situent au même niveau que les *déclarations* dans le corps d'une classe Java (non-terminal $\langle Declaration \rangle$) comme par exemple la déclaration d'un champ. Le non-terminal $\langle SqljStatement \rangle$ correspond aux requêtes SQL comme dans la méthode `getBlockbusters` de l'exemple 3.8. Les requêtes s'ancrent dans le code des méthodes Java (non-terminal $\langle Statement \rangle$).

Concernant les fonction de représentation et d'abstraction de SQLJ, elles sont spécifiées dans la documentation de SQLJ sous forme d'une table de conversions entre types SQL et types Java. Un extrait de cette table est présenté dans le figure 3.9.

Pour accéder aux résultats `r` d'une requête dans une variable Java `v`, le type `r` doit pouvoir se convertir dans le type de `v`. De la même manière, il existe une autre table de conversion correspondant à la fonction d'abstraction permettant de convertir un type Java vers un type SQL afin de pouvoir utiliser des objets Java à l'intérieur d'une requête. Dans l'exemple 3.8, la variable `seuil` doit être de type `int` afin de pouvoir être comparée avec l'objet SQL `NB_ENTREES` de type `INTEGER`.

Dans le langage SQLJ, le langage SQL n'est pas réellement le langage îlot pour Java car

les requêtes SQL ne sont pas traduites en du Java mais simplement encapsulées dans des appels à l'API `jdbc`. L'intérêt de ce *pre-processing* est d'offrir un meilleur typage ainsi que de la vérification du respect des schémas de la base. Les programmes écrits en SQLJ sont donc beaucoup plus robustes que ceux écrits directement en `jdbc`. Nous n'allons pas décrire formellement la fonction de dissolution mais en donner simplement l'intuition à travers la figure 3.10 qui donne le code obtenu après dissolution de la méthode `getBlockbusters` de l'exemple 3.8.

```
public Blockbusters getBlockbusters(int seuil){
 Blockbusters resultat;
 java.sql.PreparedStatement ps =
      recs.prepareStatement(
       "SELECT␣TITRE,␣NB_ENTREES␣"
       + "␣␣FROM␣FILMS␣"
       + "␣␣WHERE␣NB_ENTREES␣>=␣?␣");

  ps.setFloat(1, seuil);
  resutat = ps.executeQuery();
  return resultat;
}
```

où la variable `recs` correspond à un objet représentant la connexion à la base (instance de la classe `java.sql.Connection`).

FIG. 3.10: Exemple de dissolution pour SQLJ

3.3 Caractérisation des îlots formels

À partir des définitions de langages îlot et océan, nous proposons des conditions suffisantes pour assurer la préservation de la sémantique des îlots après dissolution. La principale originalité de ces conditions par rapport à la certification d'un compilateur est la notion de fonction de représentation et d'ancrage axiomatique. En effet, l'intérêt des langages îlots est de se restreindre à des types de données plus simples et même si la représentation concrète est beaucoup plus complexe, on ne s'intéresse qu'à préserver les propriétés structurelles du type abstrait.

Définition 39. *Étant donnés deux langages ol et il et un ancrage axiomatique ϕ (Définition 31) entre ces deux langages, il est un* îlot formel *pour ol si :*

1. *il est un langage îlot pour ol (Définition 38),*
2. *la fonction de représentation $\lceil\ \rceil$ est ϕ-formelle (Définition 32),*
3. *la fonction de dissolution* diss *est bien-formée (Définition 37).*

La première condition est purement syntaxique et simple à vérifier. La troisième condition est similaire à la correction d'un compilateur. La deuxième condition est plus spécifique au formalisme d'îlot. Cette définition d'îlot formel nous assure de la préservation des propriétés que l'on aurait pu vérifier sur le code îlot.

Supposons que l'on puisse définir certaines propriétés des îlots sous forme de triplets de Hoare [Hoa69]. Nous allons montrer que les conditions d'îlots formels sont suffisantes pour assurer que cette propriété est préservée après dissolution. Pour cela, nous étendons par morphisme l'environnement aux formules du premier ordre.

$$\begin{array}{l}\forall p \in \mathcal{P}_{il}, \forall t_1, \ldots, t_n, \epsilon(p(t_1, \ldots, t_n)) = \epsilon(p)(t'_1, \ldots, t'_n) \\ \quad \text{où } t'_i = \epsilon(t_i) \text{ si } t_i \in \mathcal{X}_{il} \text{ et } t_i \text{ sinon (i.e. } t_i \text{ est un objet)}, \\ \begin{array}{l|l} \epsilon(\forall x\ P) = \forall x\ \epsilon(P), & \epsilon(\exists x\ P) = \exists x\ \epsilon(P), \\ \epsilon(P_1 \vee P_2) = \epsilon(P_1) \vee \epsilon(P_2), & \epsilon(P_1 \wedge P_2) = \epsilon(P_1) \wedge \epsilon(P_2), \\ \epsilon(\neg P) = \neg\epsilon(P), & \epsilon(P_1 \rightarrow P_2) = \epsilon(P_1) \rightarrow \epsilon(P_2). \end{array}\end{array}$$

À partir de cette définition, nous notons $\epsilon \models pre$ pour $\epsilon(pre)$.

Théorème 1. *Étant donné un îlot formel il pour ol, soient pre, post deux formules du premier ordre construites à partir des prédicats $\mathcal{P}_{il}$, $\forall i \in \mathit{dom}(\mathit{diss}), \forall \epsilon \in \mathcal{E}nv_{il}$, nous avons :*

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

Démonstration. Par récurence sur la structure des formules *pre* et *post*, on prouve que pour tout environnement ϵ :

$$\begin{array}{l}\epsilon \models pre \Rightarrow \epsilon' \models post \text{ où } \langle\epsilon, i\rangle \mapsto_{bs_{il}} \epsilon' \\ \Leftrightarrow \\ \lceil\epsilon\rceil \models \phi(pre) \Rightarrow \epsilon'' \models \phi(post) \text{ où } \langle\lceil\epsilon\rceil, i\rangle \mapsto_{bs_{ol}} \epsilon''\end{array}$$

Étant donné un environnement ϵ, nous allons prouver dans un premier temps $\epsilon \models pre \Leftrightarrow \lceil\epsilon\rceil \models \phi(pre)$ par récurence sur la structure de *pre*. Le cas de base de cette récurence est $pre = p(t_1, \ldots, t_n)$. On a donc $\lceil\epsilon\rceil \models \phi(pre) \Leftrightarrow \lceil\epsilon\rceil(\phi(p)(\lceil t_1\rceil, \ldots, \lceil t_n\rceil))$. D'après la définition de la fonction de représentation $\lceil\ \rceil$ pour les environnements, on a $\lceil\epsilon\rceil(\phi(p)(\lceil t_1\rceil, \ldots, \lceil t_n\rceil)) \Leftrightarrow \lceil\phi(p)(\epsilon(t_1), \ldots, \epsilon(t_n))\rceil$. Comme la fonction de représentation $\lceil\ \rceil$ est ϕ-formelle, $\lceil\phi(p)(\epsilon(t_1), \ldots, \epsilon(t_n))\rceil \Leftrightarrow p(\epsilon(t_1), \ldots, \epsilon(t_n))$ Or $p(\epsilon(t_1), \ldots, \epsilon(t_n)) \Leftrightarrow \epsilon \models p(t_1, \ldots, t_n)$. Par transitivité de l'équivalence, nous pouvons conclure que $\epsilon \models pre \Leftrightarrow \lceil\epsilon\rceil \models \phi(pre)$ pour $pre = p(t_1, \ldots, t_n)$.

Continuons par récurence structurelle sur *pre*. Nous détaillons juste le cas $pre = P_1 \vee P_2$, les autres étant similaires. Par hypothèse de récurence, nous savons que cela est vérifié pour $pre = P_1$ et $pre = P_2$ et nous voulons le prouver pour $post = P_1 \vee P_2$. Par extension de l'environnement aux formules du premier ordre, si $pre = (P_1 \vee P_2)$ est valide, alors soit P_1 ou P_2 l'est et donc par hypothèse de récurence, $\phi(P_1)$ ou $\phi(P_2)$ sont vraies et alors aussi $\phi(P_1 \vee P_2)$. On a donc $\epsilon \models pre \Leftrightarrow \lceil\epsilon\rceil \models \phi(pre)$ (1).

De manière similaire, nous pouvons prouver que $\epsilon' \models post \Leftrightarrow \epsilon'' \models \phi(post)$ par récurence structurelle sur *post*. La preuve du cas de base est similaire à la preuve du cas de

base pour *pre* si ce n'est que $\epsilon'' \neq \lceil\epsilon'\rceil$. Mais comme la fonction de dissolution est bien-formée, nous savons que $\epsilon'' = \lceil\epsilon'\rceil \cup \delta$. Si $t_i \in \texttt{dom}(\epsilon')$, nous avons $t_i \in \texttt{dom}(\lceil\epsilon'\rceil)$. Comme ϵ'' est une fonction, $\epsilon''(t_i) = \lceil\epsilon'\rceil(t_i)$. Nous pouvons en déduire que $\epsilon''(t_i) = \lceil\epsilon'\rceil(t_i)$ car si $t_i \in dom(\epsilon')$ alors $t_i \in dom(\lceil\epsilon'\rceil)$ et comme ϵ'' est une fonction, $\epsilon''(t_i) = \lceil\epsilon'\rceil(t_i)$. Grâce à cette propriété, nous pouvons prouver le cas de base comme pour *pre*. Quant au cas de récurence sur *post*, il est très similaire à celui sur *pre*. Nous avons donc prouvé $\epsilon' \models post \Leftrightarrow \epsilon'' \models \phi(post)$ (2).

À partir des équivalences (1) et (2), on peut déduire directement la proposition. □

Un exemple d'îlot formel : MacroML MacroML [GST01] est une extension de ML permettant d'ajouter un système de macros respectant la contrainte de typage statique. À travers cet exemple concret d'extension, les auteurs montrent qu'il est possible de décrire formellement de telles extensions de manière plus abstraite que simplement d'un point de vue syntaxique. Leur approche est fondée sur l'évaluation multi-niveaux (*multi-stage computation* [Tah99]) permettant de décrire un système de types ou une sémantique opérationnelle avec différents étages indicés par des entiers. Par exemple, la sémantique opérationnelle de MacroML ou de MetaML est décrite par une famille de fonctions partielles d'évaluation indicées par un entier représentant le niveau d'évaluation. Dans ce formalisme, le concept d'ancrage se limite à la traduction d'une valeur en sa méta représentation (où l'évaluation est retardée) et inversement. Le formalisme de *multi-stage evaluation* est une généralisation de la sémantique d'un langage îlot. En effet, la sémantique de *oil* est un cas particulier de sémantique à deux niveaux où le premier niveau représente la sémantique de *ol* et le deuxième niveau la sémantique de *il* dans le cas où il n'y a pas de lac. En effet, tel qu'elle est définie dans [Tah99], une sémantique multi-niveaux ne permet pas de gérer le passage d'une partie de l'environnement aux lacs. Cette limite est due aux applications considérées pour ce type de semantiques (principalement les langages de macros et la méta-programmation). Il serait donc intéressant d'étendre ce formalisme de manière à ce qu'il s'adapte aux langages dédiés avec lacs, en s'inspirant par exemple des règles proposées dans ce chapitre.

Combinaison de systèmes de types Dans [MF07], les auteurs s'intéressent de manière plus générale à l'interopérabilité des langages comme par exemple dans les plates-formes `.NET` et `JVM`. Pour raisonner sur ce type de plates-formes, ils proposent de combiner les sémantiques opérationnelles des langages source à l'aide de nouvelles constructions frontières appelées *boundaries*. Ces opérateurs interviennent directement au niveau de la syntaxe et jouent le rôle de connecteurs entre les deux langages. Ils rendent donc complètement explicites l'ancrage des îlots. Enfin, les sémantiques des deux langages sont enrichies de nouvelles règles associées à ces opérateurs. Ces règles de réduction spécifient ainsi la nature du lien entre les deux langages à la manière de l'ancrage sémantique et de la dissolution dans le cadre des îlots. Pour illustrer leur formalisme, les auteurs proposent un sémantique pour deux langages : un langage typé à la ML et un langage non-typé dans la catégorie de Scheme. Ils proposent ensuite différents niveaux de *boundaries*. Le premier niveau appelé *lump embedding* correspond à un ancrage vide puisque les deux

langages ne partagent aucune structure de données. Le second niveau appelé *natural embedding* permet aux deux langages de manipuler les structures de données de l'autre. En particulier, la représentation des entiers est partagée. Afin d'assurer la correction du typage, de nouveaux constructeurs sont introduits dans le langage en tant que *gardes*. Ils permettent en particulier un traitement des erreurs du langage Scheme vers le langage ML.

Formalisation des interfaces entre langages Dans [GWDD06], Gybels et al. introduisent la *réflexion inter-langage*, un formalisme qui décrit comment les langages disposant de capacités réflexives peuvent communiquer une méta-représentation de leurs programmes à d'autres langages plus adaptés à la métaprogrammation que le langage d'origine. Ils instancient ensuite ce cadre au travers de deux exemples, l'un d'entre eux reliant les langage Smalltalk et SOUL (un dialecte de PROLOG). Cela rend possible la métaprogrammation de Smalltalk dans un langage logique, et permet par exemple d'exprimer élégamment l'ajout automatique d'accesseurs dans une classe. Les auteurs sont donc amenés à définir une notion d'ancrage pour décrire les communications entre langages. Dans le cas de l'exemple mettant en jeu Smalltalk et SOUL, cela revient non seulement à fournir une méta-représentation des structures Smalltalk sous forme de termes algébriques à SOUL, mais également à implanter un mécanisme d'unification dans Smalltalk au méta-niveau, mécanisme pouvant être invoqué par SOUL à la demande. Ce travail présente donc des similitudes avec les ancrages formels dans ce chapitre, et en particulier leur instanciation au cas de Tom. Cependant, ils se limitent à l'interaction de langages syntaxiquement et opérationellement disjoints (les programmes de chaque langage pouvant s'exécuter dans un processus propre), contrairement au présent formalisme.

Ces travaux montrent l'intérêt grandissant porté aux interactions de langages de programmation. Jusqu'à présent, les travaux sur ce type de systèmes étaient surtout techniques mais avec la prolifération de plates-formes multi-langages, de nouvelles questions se posent sur la formalisation et l'analyse de ce type de programmes.

3.4 Un îlot formel pour la réécriture : Tom

Le langage Tom [BM08] est une instance de ce concept d'îlots formels. Une présentation informelle du langage a été donnée dans le chapitre 2 Tom permet de définir des îlots algébriques dans n'importe quel langage de programmation généraliste. Une des originalités de Tom est d'avoir été défini indépendamment d'un langage hôte mais pour simplifier, nous allons instancier le formalisme des îlots formels sur Tom+Java. Les îlots proposés par Tom sont donc principalement la construction de termes (construction *backquote*), le filtrage de motifs (construction `%match`) et les stratégies (construction `%strategy`). Les construction `%typeterm` et `%op` permettent à l'utilisateur de spécifier l'ancrage.

Il est intéressant de noter les importantes imbrications entre les langage Tom et Java. Les programmes Tom contiennent des lacs comme par exemple l'action Java des règles qui peuvent eux-même contenir des îles (par exemple dans les construction ‘ et `%match`). Par

rapport au langage Linj qui ne contient pas de lacs, cela offre beaucoup une plus grande souplesse de programmation. En revanche, il est beaucoup plus difficile de raisonner sur ce type d'îlots.

3.4.1 Ancrage syntaxique des îlots Tom dans Java

Dans le cas de Tom, l'ancrage syntaxique anch est défini comme suit :

$$\mathsf{anch} = \left\{ \begin{array}{c} (\langle Declaration \rangle ::= \langle OpConstruct \rangle), \\ (\langle Declaration \rangle ::= \langle TypetermConstruct \rangle), \\ (\langle Declaration \rangle ::= \langle StrategyConstruct \rangle), \\ (\langle Statement \rangle ::= \langle MatchConstruct \rangle), \\ (\langle Expression \rangle ::= \langle BackQuoteConstruct \rangle) \end{array} \right\}$$

La définition des types et des opérateurs algébriques (instructions `%typeterm` et `%op`) ainsi que la définition de stratégies (construction `%strategy`) est réalisée au même niveau que les déclarations d'une classe (non-terminal $\langle Statement \rangle$). La construction `%match` est ancrée au même niveau que les instructions. Quant à la construction `, elle est ancrée au niveau des expressions.

3.4.2 Ancrage sémantique des termes algébriques

En Tom, la seule structure de données manipulée est celle de terme algébrique. Au lieu d'imposer l'ancrage sémantique par la sémantique du langage (par exemple, en utilisant une bibliothèque Java de termes), Tom permet à l'utilisateur de spécifier cet ancrage directement dans le programme. L'ancrage sémantique est donc un paramètre du programme Tom. On peut faire l'analogie avec les langages de macros qui permettent de définir la fonction de dissolution directement dans le code. L'utilisateur peut donc manipuler n'importe quelle structure de données Java comme un terme algébrique, après avoir spécifié l'ancrage au travers des constructions `%typeterm` et `%op`.

3.4.3 Dissolution par transformation source à source

Pour dissoudre les îlots de filtrage, le compilateur de Tom utilise l'ancrage défini par l'utilisateur pour décomposer la structure de données parcourue. Nous n'allons pas donner formellement la définition de la fonction de dissolution de Tom mais juste au travers d'un exemple expliquer comment l'ancrage sert à compiler le filtrage.

L'exemple suivant présente l'addition sur les entiers de Peano définie en Tom.

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

  %op Nat zero() {
```

```
    is_fsym(i) { i==0 }
    make() { 0 }
  }

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

 int plus(int t1, int t2) {
    %match(t1, t2) {
      x,zero()  -> { return `x; }
      x,suc(y) -> { return `suc(plus(x,y)); }
    }
  }

  void run() {
    System.out.println("plus(1,2) = " + plus(`suc(zero),`suc(suc(zero))));
  }

}
```

Voila le programme Java obtenu après dissolution de l'îlot `%match`.

```
  public Term plus(Term t1, Term t2) {
    if (is_fsym_zero(t2)) {
      return t1;
    } else if (is_fsym_suc(t2)) {
      return make_suc(plus(t1,subterm_suc(t2,1)));
    }
  }
```

Dans ce programme les fonctions `is_fsym_zero` et `is_fsym_suc` sont définies à partir des ancrages `%op` et permettent de déterminer si le symbole de tête de l'argument est `zero` ou `suc`. Les fonctions de constructions (comme `make_suc`) et les fonctions de décomposition qui permettent de récupérer les sous termes (comme `subterm_suc`) sont aussi définies à partir des ancrages `%op` (parties droites de `make` et `get_slot`).

Nous avons montré pour le moment comment le langage Tom instanciait le modèle d'îlot. Nous allons maintenant présenter brièvement en quoi les propriétés d'îlot formel sont vérifiées.

3.4.4 Propriétés d'îlots formels

Nous rappelons que les trois propriétés qu'un îlot formel doit vérifier sont :

1. être un langage îlot pour le langage hôte (Définition 39).

   Cette propriété est vérifiée pour Tom puisque nous avons présenté l'ancrage syntaxique, les fonctions de représentation et d'abstraction ainsi que la phase de dissolution.

2. avoir un ancrage axiomatique ϕ (Définition 31) tel que la fonction de représentation $\lceil\ \rceil$ est ϕ-formelle (Définition 32).

   Dans le cas de Tom, cette condition ne peut pas être vérifiée de manière générale car elle dépend des ancrages introduits par les utilisateurs et comme le compilateur de Tom ne réalise aucune analyse sur le code océan, il est impossible dans l'état actuel d'assurer que la fonction de représentation respecte l'ancrage axiomatique. En revanche, contrairement à beaucoup de langages îlots, l'ancrage axiomatique de Tom est clairement défini autour de deux prédicats :
   – l'égalité de deux termes $\texttt{eq}(t_1, t_2)$,
   – le test du symbole de tête d'un terme $\texttt{sym}(t, f)$.
   Ces deux prédicats sont définis de manière classique :

   $$\begin{array}{rcl} \texttt{eq}(t_1, t_2) & \triangleq & symb(t_1) = symb(t_2) \wedge \forall i \in [1..ar(symb(t_1))], \texttt{eq}(t_{1|i}, t_{2|i}) \\ \texttt{sym}(t, f) & \triangleq & symb(t) = f \end{array}$$

   Les prédicats Java, images des deux prédicats Tom par application de l'ancrage axiomatique, correspondent à des fonctions booléennes du langage hôte spécifiées dans les instructions d'ancrage `%op` et `%typeterm` et ne sont donc pas fixées à l'avance :

   $$\begin{array}{rcl} \phi(\texttt{eq}(t_1, t_2)) & \triangleq & \texttt{is\_equal}(\lceil t_1 \rceil, \lceil t_2 \rceil) \\ \phi(\texttt{sym}(t, f)) & \triangleq & \texttt{is\_fsym}(\lceil t \rceil) \end{array}$$

   Comme la fonction de représentation ainsi que l'ancrage axiomatique dépendent des fonctions définies dans les constructions d'ancrage `%typeterm` et `%op`, il est facile pour l'utilisateur de s'assurer que sa fonction de représentation respecte l'ancrage axiomatique. De plus, Tom offre un moyen de générer automatiquement des ancrages formels [KM08] pour certains types d'arborescence de classes.

3. avoir une fonction de dissolution diss bien-formée (Définition 37).
   En supposant que l'ancrage soit formel, vérifier que la fonction de dissolution est correcte revient à certifier le compilateur de Tom. L'approche choisie a été de considérer le compilateur comme une boîte noire et de prouver à chaque exécution la correction du code produit [KMR05b]. L'intérêt est de pouvoir préserver cette certification malgré les évolutions du compilateur et d'autoriser toutes sortes d'optimisations. Le code alors compilé et optimisé est prouvé correct *a posteriori*.

3.5 Synthèse

Dans ce chapitre, nous avons proposé un cadre générique permettant de définir la syntaxe et la sémantique de langages dédiés embarqués dans un langage généraliste. Une

des particularités de ces langages est de partager une partie de leur environnement d'exécution avec le code hôte environnant. Il n'est donc pas possible de raisonner localement sur les îlots car leur exécution peut dépendre de l'exécution du programme global. Le mécanisme d'ancrage proposé dans ce chapitre met en relation les constructions syntaxiques et les valeurs manipulées par les deux languages, afin de combiner à la fois leurs syntaxes et leurs sémantiques. Une fois ces ancrages spécifiés, la certification du compilateur d'un langage dédié assure que la traduction des îlots dans le langage hôte préserve leurs sémantiques. Ce formalisme d'îlots a été utilisé par Antoine Reilles pour concevoir un compilateur certifiant pour le langage Tom [KMR05b]. À chaque compilation d'un programme Tom, le compilateur peut générer un certificat de correction.

Jusqu'à présent, cette approche a été utilisée pour certifier la correction du filtrage syntaxique. Nous envisageons de poursuivre ces travaux aux autres constructions du langage. Par exemple, nous avons commençé à certifier la compilation des anti-motifs proposés dans la thèse de Radu Kopetz [Kop08] en se fondant sur l'outil Why [FM07]. Une autre perspective de ce travail concerne le typage des programmes Tom. En effet, une des particularités de Tom est de permettre à l'utilisateur de définir lui-même ses propres ancrages de données. Pour l'instant, il n'est pas possible de typer complètement des expressions Tom contenant du code Java car le code Java n'est pas analysé. La solution envisagée est d'étendre un typeur de programmes Java afin de prendre en compte les types Tom introduits par les ancrages. Un premier prototype a été écrit en se fondant sur le compilateur Java extensible développé en JastAdd [EH07a], un langage fondé sur les grammaires attribuées. La définition des attributs de typage a donc été étendue afin de prendre en compte les ancrages Tom. Cette information pourra à terme être exploitée par le typeur de Tom.

4 Un langage de stratégies pour l'analyse et la transformation de programmes

Contexte Au début de cette thèse, nous souhaitions appliquer les primitives proposées par Tom, en particulier le filtrage et les stratégies, aux problèmes d'analyse de programmes. En réécriture, le concept de stratégies a été introduit afin de contrôler l'application d'un système de règles. Au lieu d'appliquer un système de règles comme une procédure de normalisation (les règles sont donc appliquées tant que possible et à n'importe quelle position dans le terme), la stratégie définit la manière d'appliquer les règles. Pour cela, les langages fondés sur la réécriture ont proposé plusieurs approches permettant à l'utilisateur de définir ses propres stratégies et donc de pouvoir contrôler finement l'application du système de règles.

De manière générale, une stratégie offre deux types de contrôle : quelle règle doit être appliquée et à quelle position dans le terme. Cette deuxième forme de contrôle peut être intéressante dans d'autres contextes que celui du contrôle de règles. En effet, en analyse de programmes, il est parfois nécessaire de décrire des parcours complexes dans des structures arborescentes afin de collecter de l'information. Dans cette thèse, nous proposons un langage de stratégies adapté à la description de traversées d'arbres et de graphes.

Contributions La principale contribution de ce chapitre a été de formaliser et d'implémenter un langage de stratégies qui permette d'exprimer facilement des transformations et des analyses complexes d'arbres mais aussi de graphes. Ce langage que nous appellerons par la suite SL s'inspire des langages ELAN et Stratego du point de vue des constructions syntaxiques offertes. Comme nous sommes dans un contexte Java, les choix techniques au niveau de la bibliothèque ont pour leur part été influencés par les idées développées dans la bibliothèque JJTraveler. Une autre des contributions de ce chapitre a été de montrer formellement la relation entre opérateurs de logique temporelle et combinateurs de stratégies. Concrètement, cela a permis d'encoder sous forme de stratégies la vérification de formules temporelles et ainsi d'exprimer de manière plus concise [Lac03] des analyses non élémentaires.

Plan du chapitre La section 4.1 présente un tour d'horizon des formalismes développés pour la traversée de structures arborescentes. D'une part, nous présentons les différents formalismes de stratégies introduits dans les langages à base de règles et d'autre part, nous montrons des approches de traversées génériques développées dans les langages généralistes comme Java ou Haskell inspirées des langages de stratégies. Dans la section 4.2, nous présentons la syntaxe et la sémantique du langage de stratégies SL.

Dans la section 4.3, nous illustrons au travers d'exemples comment ce langage se combine avec Tom puis nous présenterons une comparaison détaillée avec la bibliothèque JJTraveler dans la section 4.4. Nous mettrons ensuite en exergue dans la section 4.5 trois aspects qui le rendent plus adapté à l'analyse de programmes que les autres langages de stratégies :

– son indépendance par rapport aux structures de données,
– l'environnement d'exécution des stratégies permettant de connaître le contexte d'évaluation (en particulier la position dans le terme global) mais aussi de définir des parcours plus complexes comme ceux dans les graphes,
– son extensibilité.

Nous proposons ensuite dans la section 4.6 une formalisation de la relation entre opérateurs de logique temporelle et combinateurs de stratégies. Enfin, nous présentons dans la section 4.7 l'architecture de la bibliothèque ainsi que des techniques d'implémentation.

4.1 Tour d'horizon

Nous présentons dans ce chapitre plusieurs approches possibles pour définir de manière modulaire des transformations sur des structures arborescentes. Tout d'abord, nous proposons un tour d'horizon des langages à base de réécriture. Ces langages ont proposé très tôt différentes techniques destinées à contrôler l'application des règles. Ces langages de stratégies s'avèrent aussi très adaptés à la description de parcours complexes d'arbres. Dans un deuxième temps, nous montrons comment ces idées ont eu un impact dans les langages généralistes. En particulier, nous présentons des techniques mises en place dans des langages tels que Haskell et Java afin de répondre au problème de la traversée générique d'arbres typés et qui se sont inspirés des langages à base de règles.

4.1.1 Contrôle dans les langages à base de règles

OBJ

OBJ [GWM+93] est l'un des tous premiers langages à avoir introduit une forme explicite de stratégies, appelées *stratégies d'évaluation*. Chaque opérateur pouvait se voir associer une liste d'entiers permettant de spécifier dans quel ordre les arguments devaient être évalués.

Par exemple, on peut déclarer l'opérateur d'addition + avec la stratégie suivante :

```
op _+_ : Int Int -> Int { strat: (1 2 0) }
```

Dans cet exemple, les entiers 1 et 2 correspondent aux deux arguments de l'opérateur + et l'entier 0 correspond à la racine. Avec cette stratégie, les arguments de l'addition sont ainsi évalués en premier (appel par valeur).

L'utilisation de stratégies d'évaluation a deux conséquences. D'une part, on ne peut pas avoir plus d'une stratégie par opérateur. D'autre part, la définition d'une transformation en fonction des résultats intermédiaires n'est pas possible car la stratégie est au niveau des opérateurs.

ELAN

Le langage ELAN [BKK+98] est un langage de spécification qui a été conçu au sein de notre équipe de recherche. ELAN puise ses origines dans OBJ mais a suivi une autre approche concernant le contrôle de l'application des règles. Le langage ELAN distingue deux sortes de règles : les règles anonymes qui s'appliquent systématiquement sur les termes dès que c'est possible et les règles étiquetées qui peuvent être déclenchées à la demande et dont l'application est contrôlée par une stratégie (contrairement aux règles anonymes qui servent à normaliser le terme et qui sont appliquées "en profondeur d'abord" (*innermost*). Appliquer une règle étiquetée sur un terme retourne un multi-ensemble de termes (gestion du non-déterminisme). Une des caractéristiques fortes du langage ELAN est donc la gestion du non-déterminisme.

Le premier langage de stratégies d'ELAN a été proposé par Marian Vittek pendant sa thèse [Vit94, KKV93]. Ce langage se fonde sur un ensemble de combinateurs génériques, faisant ainsi des stratégies des objets à part entière du langage. Ces combinateurs permettent à la fois d'exprimer finement l'ordre d'évaluation des règles mais de contrôler leur application suivant le contexte courant d'évaluation. Une règle étiquetée est vue comme une stratégie élémentaire. Deux autres stratégies élémentaires sont l'identité et l'échec. Les stratégies élémentaires ne s'appliquent qu'à la racine du terme sur lequel elles sont appliquées. ELAN propose en plus un ensemble de combinateurs qui permettent de composer les stratégies et de contrôler leurs applications. Les principaux combinateurs sont :

- l'opérateur de séquence `S1 ; S2`,
- l'opérateur de choix non-déterministe `dk(S1,...,Sn)` (`don't know`) qui choisit de manière non-déterministe une des stratégies données en argument et l'évalue,
- l'opérateur `dc(S1,...,Sn)` (`don't care`) qui choisit de manière non-déterministe une des stratégies en argument qui n'échoue pas et évalue l'application de cette stratégie sur le sujet,
- l'opérateur `first(S1,...,Sn)` qui choisit la première des stratégies en argument qui n'échoue pas et l'évalue,
- l'opérateur `one` utilisé avec `dc` ou `first` permet de ne récupérer que l'un des termes de l'ensemble de résultats,
- l'opérateur `repeat*(S)` itère sur la stratégie argument jusqu'à un échec puis retourne le dernier résultat obtenu,
- l'opérateur `iterate*(S)` est similaire à `repeat` excepté qu'il retourne tous les résultats intermédiaires.

Supposons définie une signature permettant d'écrire des expressions contenant des entiers, des opérateurs d'addition (`plus`) et de multiplication (`mult`). On considère ici que les opérateurs `plus` et `mult` sont seulement associatifs commutatifs. L'élément neutre n'a pas été déclaré. On définit donc une règle étiquetée `elimNeutral` qui élimine le 0 pour l'opérateur d'addition :

```
[elimNeutral] plus(0,x) => x end
```

Si on applique `elimNeutral` sur le terme `plus(1,0)`, on obtient `1`. Par contre si

on l'applique sur le terme `mult(2,plus(0,1))`, il y a échec car la règle ne peut pas s'appliquer à la racine. Si on applique la stratégie composée `repeat*(elimNeutral)` sur le terme `plus(0,plus(0,1))`, on obtient le terme `1`. Pour obtenir tous les résultats intermédiaires, on peut appliquer `iterate*(elimNeutral)`.

Pour descendre dans les sous-termes, il faut utiliser la clause `where` qui permet de rappeler une stratégie dans une règle. Par exemple, on peut déclarer trois règles étiquetées par `elimNeutral` :

```
[elimNeutral] plus(0,x) => x end
[elimNeutral] mult(x,y) => mult(x',y') where x':=(elimNeutral)x
                                      where y':=(elimneutral)y
                                      end
[elimNeutral] plus(x,y) => plus(x',y') where x':=(elimNeutral)x
                                      where y':=(elimneutral)y
                                      end
```

Remarquons que ce type de règles est contraire au principe de séparation règle/contrôle puisque la partie traversée est encodée dans les règles. De plus, il n'est pas possible de nommer des nouvelles stratégies composées. Peter Borovanský a donc proposé dans sa thèse [Bor98] un nouveau langage de stratégies plus puissant permettant d'encoder la traversée directement dans la stratégie et de voir les stratégies comme des règles de réécriture.

Pour cela, on enrichit le langage à l'aide de deux nouveaux types d'opérateurs :

- pour chaque opérateur `f`, un opérateur de congruence `F` de la même arité est généré ; cet opérateur permet d'appliquer des stratégies différentes à chacun des sous-termes d'un terme dont le symbole de tête est `f`,
- l'opérateur `normalize(S)` normalise le sujet en appliquant les règles à n'importe quelle position. Il est défini à partir des opérateurs de congruence et normalise de manière *leftmost-innermost*.

Ces nouveaux opérateurs offrent donc un moyen de contrôler la traversée. Cependant, ils sont dépendants d'une signature donnée. Peter Borovanský a donc proposé par la suite des opérateurs de traversée polymorphes. Pour que ces opérateurs fonctionnent sur des termes typés, le langage ELAN propose un système de constructeurs et destructeurs polymorphes pour les termes (ces fonctions travaillent sur la sorte *Any* qui est définie dans un module paramétré par n'importe quelle sorte X et définissant une coercion entre X et *Any*). La fonction *explode* permet de récupérer les sous-termes d'un terme de sorte *Any*, la fonction *implode* permet de construire un terme à partir de sous-termes de types *Any* et enfin la fonction *functor* permet de récupérer le symbole de tête d'un terme non-typé. Ces fonctions transforment donc les termes typés en termes non typés et permettent de définir des nouveaux opérateurs polymorphes de stratégies comme par exemple l'application d'une stratégie au i^e fils. Cette approche non-typée a beaucoup inspiré le langage Stratego [VBT98] qui a choisi de manière plus radicale uniquement la transformation de termes non typés. Ce travail a aussi beaucoup inspiré les fonctions polymorphes introduites dans la librairie Strafunski [LV02].

Un autre des apports du langage proposé par Peter Borovanský est de pouvoir définir ses propres opérateurs de stratégies. Cela permet de définir les stratégies de manière récursive. `repeat` n'est alors plus une brique de base du langage mais peut se définir récursivement :

```
try(s) = first one(s, id)
repeat(s) = try(s ; repeat(s))
```

Le langage de stratégies d'ELAN est donc particulièrement expressif et permet de séparer clairement les règles de leur mode d'évaluation.

Maude

Le langage de spécification Maude [CEL99] est fondé sur la logique de réécriture. Ce langage propose une solution plus générale au problème de l'application des règles en utilisant ses capacités réflexives [CM00]. Chaque objet du langage peut en effet être représenté au *méta-niveau*. L'application des règles peut alors être réalisée à l'aide de l'opérateur `meta-apply`. Cet opérateur prend comme argument la *méta-représentation* de tout le contexte d'évaluation.

En Maude, une spécification est composée d'équations qui servent à normaliser systématiquement les termes et un ensemble de règles qui peuvent être appliquées suivant une stratégie. L'évaluation de `meta-apply` normalise donc le terme à l'aide des équations du module puis filtre le terme sur l'ensemble des règles. L'évaluation retourne la méta-représentation du terme obtenu après réécriture par la règle déclenchée puis normalisation à l'aide des équations.

L'application d'un ensemble de règles peut donc être contrôlée par un autre programme défini par réécriture. Ce programme peut lui même être contrôlé par un autre programme du niveau méta-méta et ainsi de suite. Cette tour de réflexivité est très expressive mais relativement difficile à utiliser car les programmes utilisant le niveau méta sont très difficiles à déboguer.

Stratego

ELAN a beaucoup inspiré le langage de transformation de programmes Stratego [VBT98] dont le concept clé est la combinaison de stratégies élémentaires. Contrairement à ELAN, Stratego ne propose qu'un nombre restreint de combinateurs et s'est affranchi du typage. Les programmes Stratego sont donc plus simples que les programmes ELAN équivalents mais l'expressivité en est plus limitée. En particulier, il n'y a pas de gestion du non-déterminisme et le *backtracking* est uniquement local aux opérateurs de choix.

Ces sept stratégies de base sont donc la composition séquentielle, les choix déterministe et non-déterministe, l'identité, l'échec ainsi que 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 stratégie `try(s)` essaie d'appliquer `s` et en cas d'échec renvoie l'identité. La négation `not(s)` se comporte comme l'identité si l'application de `s` échoue, et échoue sinon. La définition récursive de stratégies est rendue possible par un opérateur

de récursion μ`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) =` μ`x(try(s ; x))`.

Une des caractéristiques de Stratego est de proposer des opérateurs de congruence génériques. L'opération fondamentale pour les traversées de termes est l'application d'une stratégie sur un sous-terme direct donné d'un terme. Tout d'abord, Stratego propose un opérateur `i(s)` qui applique la stratégie sur le `i`ème fils du sujet. Si `i` ne respecte pas l'arité du symbole fonctionnel du sujet, la stratégie échoue. Cependant, cet opérateur étant dépendant de l'arité, il n'est pas suffisant pour décrire de manière générique des stratégies telles que le *top-down* qui applique les règles sur le terme de la racine vers les feuilles. Trois opérateurs supplémentaires sont donc proposés :

- `All(s)`, qui applique la stratégie `s` à tous les sous-termes 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. L'application de `One(s)` échoue s'il n'y a pas de fils sur lequel `s` s'applique ; 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 sous-terme 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 indépendamment d'une signature des stratégies telles que `BottomUp(s)` ou `TopDown(s)`, respectivement par μ`x(All(x) ; s)` et μ`x(s ; All(x))`.

Par sa simplicité, Stratego est un langage adapté à la transformation de programmes. Cependant, comme ELAN, en tant que langage dédié, il est assez limité pour tout ce qui ne relève pas de la transformation d'arbres comme par exemple les entrées-sorties ou encore la gestion de structures de données non-arborescentes telles que les tables de hachage qui facilitent la définition d'analyses de programmes.

ASF+SDF

ASF+SDF [BHKO02] est avant tout un langage de transformations sans stratégie. Récemment, un langage de stratégies similaire à celui de OBJ a été ajouté. Chaque opérateur peut être annoté par une *fonction de traversée* [BKV03b]. Les traversées peuvent être de trois types :

- *transformation* (noté `trafo`) qui traverse le premier argument. Afin de préserver les types, le type de retour d'un symbole défini avec une fonction de traversée `trafo` est donc celui de son premier argument.

```
f(s1,...,sn) -> s1 {traversal(trafo,...)}
```

- *accumulateur* (noté `accu`) qui permet de partager une variable globale entre toutes les applications de la règle. En effet, lorsque la règle est appliquée, elle réutilise la

valeur calculée par la dernière application. Le type de retour d'un symbole défini avec une fonction de traversée `accu` est donc celui de son second argument.

```
f(s1,...,sn) -> s2 {traversal(accu,...)}
```

– *transformation avec accumulateur* correspond à la combinaison des deux précédents types de fonctions de traversées. Pour préserver les types, le type de retour d'un symbole déclaré est un couple des types des deux premiers arguments.

```
f(s1,...,sn) -> <s1,s2> {traversal(trafo,accu,...)}
```

Les deux premiers arguments d'un symbole fonctionnel jouent donc un rôle particulier. Le premier correspond à l'arbre que l'on souhaite transformer et le second au type d'information que l'on souhaite accumuler lors de la traversée (lors d'analyse de programmes par exemple).

En plus de ces trois types de traversée, ASF+SDF propose deux ordres de visite (`bottom-up` ou `top-down`) et deux opérateurs de contrôle (`break` et `continue`) qui permettent respectivement de stopper ou de continuer l'évaluation après filtrage.

On peut par exemple définir la somme des valeurs entières contenu dans un arbre comme suit :

```
module Tree-sum
imports Tree-syntax
exports
  context-free syntax
    sum(Tree, Integer) -> Integer {traversal(accu, top-down, continue)}

hiddens
  variables
    "N"[0-9]* -> Integer

equations
  sum(N1, N2) = N1 + N2
```

Le constructeur `sum` prend deux arguments : l'arbre à évaluer et un entier qui joue le rôle d'accumulateur. La fonction de traversée est donc de type `accu`. La stratégie de parcours choisie est `top-down`.

Dans cette approche, seules les notions de traversée sont traitées, et le contrôle est très limité (opérateurs `break` et `continue`). Les notions de répétition, condition 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.

4.1.2 Traversée générique dans les langages typés

Les langages à base de règles, en particulier les langages ELAN et Stratego, ont inspiré les langages généralistes qui ont proposé des *design patterns* adaptés au parcours

générique de structures arborescentes typées. Nous présentons ici deux bibliothèques de combinateurs génériques : la bibliothèque Strafunski développée pour Haskell et JJTraveler pour Java. Contrairement à Stratego, le principal challenge est de pouvoir définir ces combinateurs dans un contexte typé.

Strafunski

La bibliothèque Strafunski [LV02] a été développée afin de pouvoir définir plus simplement des transformations et des analyses de programmes dans le langage Haskell. L'idée clé est de voir les stratégies comme des fonctions génériques appelées *stratégies fonctionnelles*. Comme Haskell est un langage typé, une stratégie doit préserver les types. Strafunski fournit une bibliothèque de combinateurs génériques de fonctions.

Haskell n'offre pas de support pour la programmation générique. Chaque type des structures de données sur lesquels on souhaite appliquer une stratégie fonctionnelle doit instancier la classe Term. Cette classe est composée de deux fonctions : `explode` qui construit à partir d'un terme une représentation générique de ce terme (type `TermRepr`) et la fonction inverse `implode` qui permet de reconstruire le terme à partir de sa représentation générique. Les combinateurs génériques travaillent donc au niveau du type `TermRepr`.

En plus d'une bibliothèque de combinateurs génériques, Strafunski offre un outil permettant de générer à partir de la définition syntaxique de types de données algébriques l'implémentation de la classe Term ce qui permet d'utiliser directement la bibliothèque de combinateurs. Cet outil est indispensable car la plupart des ASTs sont composés d'un nombre conséquent de types de nœuds. De plus, plusieurs formats de termes sont proposés : Strafunski offre en particulier un support pour les ATerms [BJKO00] et pour les schémas XML.

JJTraveler

La bibliothèque JJTraveler [Vis01] propose des combinateurs de traversée pour Java. Ces travaux montrent que le *design pattern* visiteur [GHJV93] utilisé généralement pour décrire des traversées d'arbres dans les langages orientés objet souffre de deux principales limites qui sont le manque de contrôle et le peu de modularité (les visiteurs peuvent être seulement spécialisés par héritage). En effet, il n'est pas possible de séparer transformation et contrôle. La stratégie de parcours est soit intégrée au code du visiteur lui-même, soit dans la méthode `accept` des classes représentant l'arbre.

JJTraveler propose un ensemble de combinateurs de visiteurs génériques et réutilisables. Chaque classe représentant des nœuds de l'arbre doit implémenter une interface `Visitable` qui décrit comment visiter un nœud, comment modifier et accéder à ses sous-termes. Elle joue un rôle similaire à celui des ancrages de Tom en explicitant la structure arborescente. Contrairement à Stratego, JJTraveler manipule des arbres typés et doit donc proposer des visiteurs préservant le type des nœuds visités. Ce langage ayant beaucoup inspiré le développement de la librairie SL, une comparaison plus détaillée est proposée dans la section 4.4.

4.2 Syntaxe et sémantique du langage de stratégies SL

Les langages ELAN et Stratego ont beaucoup inspiré la formalisation du langage de stratégies SL défini ci-dessous. Cependant, l'intérêt d'embarquer ce type de constructions dans des langages généralistes est de pouvoir profiter du langage hôte pour tout ce qui ne concerne pas directement la transformation et le parcours de structures arborescentes. De plus, les constructions de Tom étant intensivement mêlées à celles de Java, il est possible d'exprimer de manière concise certaines transformations et collectes d'information en profitant des constructions et de la bibliothèque standard de Java.

Une des principales contributions de cette thèse est de proposer un langage de stratégies assez expressif pour pouvoir à la fois contrôler des transformations de structures arborescentes mais aussi exprimer de manière élégante des analyses de programmes. Dans cette section, nous allons présenter la syntaxe et la sémantique du langage de stratégies SL.

En SL, une stratégie `s` est représentée par un objet de la classe `Strategy`. Appliquer une stratégie `s` sur un terme representé par l'objet `t` consiste à appeler la méthode `visit` de la classe `Strategy` : `s.visit(t)` retourne un objet `t'` qui correspond à l'application de `s` sur `t`. Comme Tom manipule des structures arborescentes typés, les stratégies préservent les types (autrement dit, `t'` est du même type que `t`).

Dans la suite, chaque élément de SL est illustré à l'aide d'un exemple fondé sur des ASTs d'un langage de programmation de type impératif présenté en figure 4.1. Ce langage est composé d'une instruction de séquence, d'instructions de déclaration et d'assignation de variables. La portée des variables définies par l'instruction `Declare(v,e,i)` correspond au bloc `i`. Dans les programmes bien-formés, les instructions `Assign(v,e)` ne peuvent apparaître que dans la portée d'une variable `v` et pour simplifier, une instruction `Declare(v,e,i)` ne peut pas être utilisée dans la portée d'une variable `v`. Par exemple, le programme `Declare("x",Cst(1),Declare("x",Cst(2),Print(Var("x"))))` n'est pas bien formé puisque la variable `x` est redéclarée dans son bloc de portée.

```
Instruction = Sequence(i1:Instruction,i2:Instruction)
            | Declare(varname:String,e:Expression,i:Instruction)
            | Assign(varname:String,e:Expression)
            | Print(e:Expression)

Expression = Plus(e1:Expression,e2:Expression)
           | Mult(e1:Expression,e2:Expression)
           | Var(name:String)
           | Cst(n:int)
```

FIG. 4.1: Syntaxe du langage impératif utilisé pour illustrer SL

4.2.1 Stratégies élémentaires

Les deux premières stratégies élémentaires sont l'identité et l'échec, définies par les classes `Identity` et `Fail` de la bibliothèque. Ces deux stratégies n'ont aucun effet sur le terme. L'application de la stratégie identité retourne le terme inchangé et l'application de la stratégie échec déclenche une exception Java de type `VisitFailure`. La troisième et dernière stratégie élémentaire correspond à un ensemble de règles de réécriture (qui est appliqué une seule fois en position racine). Un ensemble de règles est implémenté à l'aide de la construction Tom `%strategy`. Cette construction est définie en étendant une stratégie par défaut. La stratégie `ElimNeutral` par exemple élimine les éléments neutres de l'addition et la multiplication et étend la stratégie `Fail` :

```
%strategy ElimNeutral() extends Fail() {
  visit Expression {
    Plus(Cst(0),e) -> e
    Plus(e,Cst(0)) -> e
    Mult(Cst(1),e) -> e
    Mult(e,Cst(1)) -> e
  }
}
```

La stratégie `EvalConst` qui évalue les expressions constantes en utilisant les opérateurs arithmétiques Java peut être définie comme suit :

```
%strategy EvalConst() extends Fail() {
  visit Expression {
    Plus(Cst(c1),Cst(c2)) -> Cst(c1 + c2)
    Mult(Cst(c1),Cst(c2)) -> Cst(c1 * c2)
  }
}
```

Le système de règles doit préserver les types (le type de la partie gauche d'une règle est exactement le même que celui de la partie droite, il n'y a pas de notion de sous-typage en Tom).

4.2.2 Combinateurs de stratégies

Le langage SL propose un ensemble de combinateurs génériques de stratégies inspiré de celui de Stratego. Chaque combinateur correspond à une classe de la bibliothèque SL. On distingue deux types de combinateurs :

- les combinateurs de composition qui permettent de construire un stratégie complexe en composant des stratégies élémentaires,
- les combinateurs de traversée qui permettent d'appliquer une stratégie à une position quelconque dans le terme.

Combinateurs de composition

L'ensemble de combinateurs de composition est défini comme suit :

```
Strategy = Sequence(s1:Strategy,s2:Strategy)
         | Choice(s1:Strategy,s2:Strategy)
         | Mu(var:String,s:Strategy)
         | MuVar(var:String)
         | Not(s:Strategy)
         | IfThenElse(s1:Strategy,s2:Strategy,s3:Strategy)
```

On retrouve des opérateurs présents dans les langages ELAN et Stratego comme `Sequence(s1,s2)` et `Choice(s1,s2)`. Pour construire des stratégies récursives, SL propose deux opérateurs `Mu` et `MuVar`. Comme dans les définitions formelles, l'opérateur `Mu` est paramétré par un nom de variable et par une stratégie. Pour réaliser des appels récursifs, il est possible d'utiliser le combinateur `MuVar` (avec pour seul argument le nom de la variable) à l'intérieur de la stratégie argument.

Combinateurs de traversée

Les combinateurs de traversée du langage SL sont :

```
Strategy = All(s:Strategy)
         | One(s:Strategy)
         | Omega(i:int,s:Strategy)
         | Up(s:Strategy)
```

On retrouve les combinateurs `All(s)`, `One(s)` ainsi que l'opérateur `Omega(i,s)` qui permet d'appliquer la stratégie `s` au i^e fils et correspond à l'opérateur `i(s)` de Stratego. Par rapport aux autres langages de stratégie, SL propose un nouveau combinateur de base : l'opérateur `Up(s)` qui se comporte de manière inverse par rapport aux opérateurs de traversée `All` et `One` en appliquant la stratégie argument au père du nœud courant. Cette stratégie est très utile pour définir des analyses sémantiques telles que la résolution de noms en Java.

4.2.3 Stratégies composées

Toutes les stratégies de réduction classiques peuvent être définies en utilisant ces combinateurs. Par exemple :

```
Try(s) = Choice(s,Identity())
Repeat(s) = Mu("x",Try(Sequence(s,MuVar("x"))))
OnceBottomUp(s) = Mu("x",Choice(One(MuVar("x")),s))
BottomUp(s) = Mu("x",Sequence(All(MuVar("x")),s))
TopDown(s) = Mu("x",Sequence(s,All(MuVar("x"))))
Innermost(s) = Mu("x",Sequence(All(MuVar("x")),Try(Sequence(s,MuVar("x")))))
```

La stratégie `Try` n'échoue jamais : elle essaie d'appliquer `s` et si cela réussit, le résultat est retourné. Sinon la stratégie `Identity` est appliquée et le sujet originel est retourné. Par exemple, `Try(ElimNeutral())` appliquée au sujet `Cst(1)` renvoie `Cst(1)`. La stratégie `Repeat(s)` applique `s` autant de fois que possible, jusqu'à un échec. Le dernier résultat avant l'échec est alors retourné. Par exemple, `Repeat(ElimNeutral())` appliquée à `Plus(Plus(Plus(Cst(1),Cst(0)),Cst(0),Cst(0)))` renvoie `Cst(1)`. La stratégie `OnceBottomUp` essaie d'appliquer `s` une seule fois et en profondeur d'abord. Par exemple, `OnceBottomUp(EvalConst())` appliquée à `Plus(Plus(Cst(1),Cst(2)),Cst(3))` renvoie `Plus(Cst(3,Cst(3)))`. La stratégie `BottomUp` se comporte comme `OnceBottomUp` mais applique `s` à tous les nœuds. Si `s` échoue sur au moins un nœud, `BottomUp(s)` échoue. `BottomUp(EvalConst())` échoue sur `Plus(Plus(Cst(1),Cst(2)),Cst(3))` car `EvalConst` échoue sur les constantes. La stratégie `TopDown` est similaire `BottomUp` sauf qu'elle parcourt l'arbre de la racine aux feuilles. Elle échouera donc aussi sur l'exemple précédent. Enfin, la stratégie `Innermost(s)` permet de calculer la forme normale d'un terme en appliquent s autant de fois que possible en profondeur d'abord. Par exemple, si on évalue toujours sur le même sujet `TopDown(Try(EvalConst()))`, le résultat obtenu est `Plus(Cst(3),Cst(3))`. Pour obtenir une évaluation complète, on peut utiliser la stratégie `Innermost(EvalConst())`, qui permet d'obtenir directement le résultat `Cst(6)`.

4.2.4 Sémantique opérationnelle de SL

Dans cette thèse, nous proposons une sémantique opérationnelle pour le langage SL. Cette sémantique se distingue de celle de Stratego par le fait que les stratégies SL sont appliquées dans un contexte. La première originalité de cette sémantique est d'introduire explicitement cet environnement. En effet, le combinateur `Up` ne pourrait pas être défini dans une sémantique comme celle de Stratego où aucun effet de bord dans le terme global n'est possible. De plus, dans le langage SL, les stratégies sont aussi considérées comme des termes (cette caractéristique sera présentée en détail dans la section 4.3.2). Pour insister sur ce fait, nous indiquons explicitement dans quel sous-terme de la stratégie globale nous nous trouvons. À chaque pas de réduction, les termes qui représentent la stratégie globale et le sujet global sont conservés.

L'application d'une stratégie s sur un sujet t correspond au pas de réduction $\langle \underline{s}, t, \epsilon \rangle \rightarrow t'$ où ϵ correspond à la position racine du sujet et $\underline{s}$ indique que la position dans le terme de stratégie correspond à la racine. Dans chaque règle d'inférence, les variables de contexte (notées $S\{\ldots\}$) sont utilisées dans le terme de stratégie pour indiquer que la position courante n'est pas nécessairement la racine. Le sous-terme souligné dans le terme de stratégie correspond au sous-terme courant à appliquer. Par exemple, pour définir la règle d'inférence du combinateur μ, des variables de contexte permettent d'indiquer pour chaque variable récursive le lieur μ correspondant. Pour simplifier, nous considérons que dans $\mu x(S\{x\})$, il n'y pas d'autre stratégie μx dans le chemin entre la position de μx et la variable x dans le contexte S. Notons que dans les règles de la sémantique, on suppose que $t, t' \in \mathcal{T}(\mathcal{F})$ et $r \in \mathcal{T}(\mathcal{F}) \uplus \{\mathsf{fail}\}$.

Pour présenter la sémantique opérationnelle de SL, nous avons séparé le langage en

trois ensembles d'opérateurs :

– les stratégies élémentaires :

$$\frac{\exists \sigma, t_{|\omega} = \sigma(lhs)}{\langle S\{\underline{lhs \rightarrow rhs}\}, t, \omega\rangle \rightarrow t[\sigma(rhs)]_\omega} \qquad \frac{\nexists \sigma, t_{|\omega} = \sigma(lhs)}{\langle S\{\underline{lhs \rightarrow rhs}\}, t, \omega\rangle \rightarrow \mathsf{fail}}$$

$$\frac{}{\langle S\{\underline{Identity}\}, t, \omega\rangle \rightarrow t} \qquad \frac{}{\langle S\{\underline{Fail}\}, t, \omega\rangle \rightarrow \mathsf{fail}}$$

– les combinateurs de composition (la position courante du sujet est inchangée) :

$$\frac{\langle S\{Choice(\underline{s_1}, s_2)\}, t, \omega\rangle \rightarrow t'}{\langle S\{\underline{Choice(s_1, s_2)}\}, t, \omega\rangle \rightarrow t'} \qquad \frac{\langle S\{Choice(\underline{s_1}, s_2)\}, t, \omega\rangle \rightarrow \mathsf{fail} \quad \langle S\{Choice(s_1, \underline{s_2})\}, t, \omega\rangle \rightarrow r}{\langle S\{\underline{Choice(s_1, s_2)}\}, t, \omega\rangle \rightarrow r}$$

$$\frac{\langle S\{Sequence(\underline{s_1}, s_2)\}, t, \omega\rangle \rightarrow t' \quad \langle S\{Sequence(s_1, \underline{s_2})\}, t', \omega\rangle \rightarrow r}{\langle S\{\underline{Sequence(s_1, s_2)}\}, t, \omega\rangle \rightarrow r} \qquad \frac{\langle S\{Sequence(\underline{s_1}, s_2)\}, t, \omega\rangle \rightarrow \mathsf{fail}}{\langle S\{\underline{Sequence(s_1, s_2)}\}, t, \omega\rangle \rightarrow \mathsf{fail}}$$

$$\frac{\langle S\{IfThenElse(\underline{s1}, s2, s3)\}, t, \omega\rangle \rightarrow \mathsf{fail} \quad \langle S\{IfThenElse(s1, s2, \underline{s3})\}, t, \omega\rangle \rightarrow r}{\langle S\{\underline{IfThenElse(s1, s2, s3)}\}, t, \omega\rangle \rightarrow r}$$

$$\frac{\langle S\{IfThenElse(\underline{s1}, s2, s3)\}, t, \omega\rangle \rightarrow t' \quad \langle S\{IfThenElse(s1, \underline{s2}, s3)\}, t, \omega\rangle \rightarrow r}{\langle S\{\underline{IfThenElse(s1, s2, s3)}\}, t, \omega\rangle \rightarrow r}$$

$$\frac{\langle S\{\mu x(\underline{s})\}, t, \omega\rangle \rightarrow r}{\langle S\{\underline{\mu x(s)}\}, t, \omega\rangle \rightarrow r} \qquad \frac{\langle S\{\underline{\mu x(S\{x\})}\}, t, \omega\rangle \rightarrow r}{\langle S\{\mu x(S\{\underline{x}\})\}, t, \omega\rangle \rightarrow r}$$

– les combinateurs de traversée :

$$\frac{\langle S\{Omega(i,\underline{s})\},t,\omega\,.\,i\rangle \to r}{\langle S\{\underline{Omega(i,s)}\},t,\omega\rangle \to r}\ i \in [1,ar(symb(t_{|\omega}))] \qquad \frac{}{\langle S\{\underline{Omega(i,s)}\},t,\omega\rangle \to \mathsf{fail}}\ i \notin [1,ar(symb(t_{|\omega}))]$$

$$\frac{\forall i \in [1,ar(symb(t_{|\omega}))]\ \exists t_i \in \mathcal{T}(\mathcal{F}),\ \langle S\{All(\underline{s})\},t,\omega\,.\,i\rangle \to t[t_i]_{\omega.i}}{\langle S\{\underline{All(s)}\},t,\omega\rangle \to t[t_1]_{\omega.1}\ldots[t_n]_{\omega.n}} \qquad \frac{\exists i \in [1,ar(symb(t_{|\omega}))],\ \langle S\{All(\underline{s})\},t,\omega\,.\,i\rangle \to \mathsf{fail}}{\langle S\{\underline{All(s)}\},t,\omega\rangle \to \mathsf{fail}}$$

$$\frac{\exists i \in [1,ar(symb(t_{|\omega}))]\ \langle S\{One(\underline{s})\},t,\omega\,.\,i\rangle \to t'}{\langle S\{\underline{One(s)}\},t,\omega\rangle \to t'} \qquad \frac{\forall i \in [1,ar(symb(t_{|\omega}))]\langle S\{One(\underline{s})\},t,\omega\,.\,i\rangle \to \mathsf{fail}}{\langle S\{\underline{One(s)}\},t,\omega\rangle \to \mathsf{fail}}$$

$$\frac{\langle S\{Up(\underline{s})\},t,\omega\rangle \to t'}{\langle S\{\underline{Up(s)}\},t,\omega\,.\,i\rangle \to t'}\ (\omega\,.\,i) \in Pos(t') \qquad \frac{}{\langle S\{\underline{Up(s)}\},t,\epsilon\rangle \to t}$$

$$\frac{\langle S\{Up(\underline{s})\},t,\omega\rangle \to \mathsf{fail}}{\langle S\{\underline{Up(s)}\},t,\omega\,.\,i\rangle \to \mathsf{fail}} \qquad \frac{\langle S\{Up(\underline{s})\},t,\omega\rangle \to t'}{\langle S\{\underline{Up(s)}\},t,\omega\,.\,i\rangle \to \mathsf{fail}}\ (\omega\,.\,i) \notin Pos(t')$$

Remarquons que seul l'opérateur `One` et les règles de réécriture ont une sémantique non déterministe. Pour les règles de réécriture, seules les règles équationnelles peuvent entraîner plusieurs substitutions possibles. Concrètement, le système choisit de manière déterministe l'une d'elle. Un choix déterministe similaire est réalisé pour l'opérateur `One`.

Exemple 11. Construisons l'arbre de dérivation pour un exemple simple sans effet de bord. Prenons la stratégie `BottomUp(Eval())` où la stratégie élémentaire `Eval` est définie par les deux règles suivantes :

– `r1 : Plus(Cst(x),Cst(y)) -> Cst(x+y)`,
– `r2 : Mult(Cst(x),Cst(y)) -> Cst(x*y)`.

Dans cet exemple, on choisit comme stratégie par défaut `Identity`. On peut donc la décomposer en `Choice(r1,r2,Identity())` et ainsi utiliser la règle de dérivation d'une règle de réécriture. Après décomposition du combinateur `BottomUp`, on obtient donc le terme de stratégies suivant :

```
BottomUp(Eval()) =
  Mu("x",Sequence(All(MuVar("x")),Choice(r1,r2,Identity())))
```

L'arbre de dérivation obtenu pour le sujet `Plus(Mult(Cst(1),Cst(2)),Cst(3))` est présenté dans la Figure 4.2.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_1}{\langle C\{\underline{x}\}, t, 1\,.\,1\rangle \to t} \quad \dfrac{\Pi_2}{\langle C\{\underline{x}\}, t, 1\,.\,2\rangle \to t}}{\langle C\{\underline{All(x)}\}, t, 1\rangle \to t} \quad \Pi_3}{\langle \mu x(\underline{Seq(All(x), Eval)}), t, 1\rangle \to Plus(Cst(2), Cst(3))}}{\langle \mu x(\underline{Seq(All(x), Eval)}), t, 1\rangle \to Plus(Cst(2), Cst(3))}}{\langle \mu x(Seq(All(\underline{x}), Eval)), t, 1\rangle \to Plus(Cst(2), Cst(3))} \quad \dfrac{\Pi_4}{\langle \mu x(Seq(All(\underline{x}), Eval)), t, 2\rangle \to t}}{\langle \mu x(Seq(\underline{All(x)}, Eval)), t, \epsilon\rangle \to Plus(Cst(2), Cst(3))} \quad \Pi_5}{\langle \mu x(\underline{Seq(All(x), Eval)}), t, \epsilon\rangle \to Cst(5)}}{\langle \underline{BottomUp(Eval)}, Plus(Mult(Cst(1), Cst(2)), Cst(3)), \epsilon\rangle \to Cst(5)}$$

Arbre de dérivation Π_1 (similaire pour Π_2 et Π_4) :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{arity(t_{|1.1.1}) = 0}{\langle C\{\underline{All(x)}\}, t, 1\,.\,1\,.\,1\rangle \to t} \quad \dfrac{\Pi_6}{\langle C\{\underline{Eval}\}, t, 1\,.\,1\,.\,1\rangle \to t}}{\langle \mu x(\underline{Seq(All(x), Eval)}), t, 1\,.\,1\,.\,1\rangle \to t}}{\langle \mu x(Seq(\underline{All(x)}, Eval)), t, 1\,.\,1\rangle \to t} \quad \langle C\{\underline{Eval}\}, t, 1\,.\,1\,.\,1\rangle \to t}{\langle \mu x(\underline{Seq(All(x), Eval)}), t, 1\,.\,1\rangle \to t}}{\langle \mu x(\underline{Mu x(Seq(All(x), Eval))}), t, 1\,.\,1\rangle \to t}}{\langle \mu x(Seq(All(\underline{x}), Eval)), t, 1\,.\,1\rangle \to t}$$

Arbre de dérivation Π_3 (similaire pour Π_6) :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\nexists \sigma, Mult(Cst(1), Cst(2)) = \sigma(Plus(x, y))}{\langle C\{Choice(\underline{r1}, Choice(r2, Id))\}, t, 1\rangle \to \mathsf{fail}} \quad \dfrac{\dfrac{t_{|1} = Mult(Cst(1), Cst(2)) = \{x \mapsto Cst(1), y \mapsto Cst(2)\}(Mult(x, y))}{\langle C\{Choice(r1, Choice(\underline{r2}, Id))\}, t, 1\rangle \to Plus(Cst(2), Cst(3))}}{\langle C\{Choice(r1, \underline{Choice(r2, Id)})\}, t, 1\rangle \to Plus(Cst(2), Cst(3))}}{\langle C\{\underline{Choice(r1, Choice(r2, Id))}\}, t, 1\rangle \to Plus(Cst(2), Cst(3))}}{\langle C\{\underline{Eval}\}, t, 1\rangle \to Plus(Cst(2), Cst(3))}$$

Arbre de dérivation Π_5 :

$$\dfrac{\dfrac{t_{|\epsilon} = Plus(Cst(2), Cst(3)) = \{x \mapsto Cst(2), y \mapsto Cst(3)\}(Plus(x, y))}{\langle C\{Choice(\underline{r1}, Choice(r2, Id))\}, t, 1\rangle \to Cst(5)}}{\langle C\{\underline{Eval}\}, Plus(Cst(2), Cst(3)), \epsilon\rangle \to Cst(5)}$$

FIG. 4.2: Arbre de dérivation de l'application de la stratégie `BottomUp(Eval())` au sujet `Plus(Mult(Cst(1),Cst(2)),Cst(3))`

Remarques sur la sémantique du combinateur `Up`

La présence de l'opérateur `Up` modifie profondément le comportement des stratégies. En effet, dans un langage comme Stratego, les stratégies réalisent des transformations locales et n'ont donc d'effet que sur le sous-terme sur lequel elles s'appliquent. L'opérateur `Up` permet de réaliser des transformations sur son père et donc s'il est utilisé de manière récursive, peut réaliser un effet de bord sur n'importe quel sous-terme du terme global. Cela explique pourquoi le pas de réduction retourne le terme global courant et non simplement le sous-terme courant modifié.

Cette extension est sans effet pour la plupart des opérateurs de stratégie qui n'agissent que sur un seul sous-terme. Par exemple, pour l'opérateur `One(s)`, le sujet global obtenu après application de `s` sur l'un des sous-termes est retourné. Le seul problème se pose pour l'opérateur `All(s)` qui a un comportement parallèle (`s` est appliquée simultanément à tous les sous-termes). Nous souhaitons être conservatif par rapport à la sémantique proposée par Stratego. Autrement dit, la sémantique de SL pour une stratégie sans `Up` doit être la même que celle de Stratego. Pour être cohérente avec la sémantique du `All` parallèle, l'application de `s` ne doit pas réaliser d'effet de bord dans le terme global. Plus formellement, $\forall i \in [1,ar(symb(t_{|\omega}))]\ \exists t_i \in \mathcal{T}(\mathcal{F}),\ \langle S\{All(\underline{s})\}, t, \omega \centerdot i\rangle \rightarrow t[t_i]_{\omega \centerdot i}$. Autrement dit, seul le sous-terme à la position $\omega \centerdot i$ est modifié.

Cette condition peut paraître assez restrictive mais nous verrons par la suite que l'opérateur `Up` a surtout été introduit pour obtenir localement de l'information sur le contexte et donc qu'il est principalement utilisé pour contrôler l'application des règles en fonction du contexte. Son comportement se limite à utiliser l'identité et l'échec pour calculer une condition sur le contexte et ne réalise donc pas de transformation. Cependant, si l'utilisateur souhaite l'utiliser sous un opérateur `All` pour réaliser des transformations, il est possible d'utiliser une version séquentielle du `All` autorisant les effets de bord. Ce nouvel opérateur sera présenté en détail dans le chapitre 5 lorsque les stratégies seront utilisées dans des graphes.

Notons aussi que le comportement de l'opérateur `Up` lorsqu'il est appliqué à la position racine est celui de l'identité (comme le `All` aux feuilles). Il aurait été tout à fait possible de choisir plutôt l'échec comme c'est le cas pour l'opérateur `One`. En effet, si un nœud pouvait avoir plusieurs pères (comme dans les graphes), on pourrait imaginer définir deux opérateurs `AllUp` et `OneUp` qui se comporteraient comme le `All` et le `One` mais sur l'ensemble des pères d'un nœud.

4.3 Programmer avec les langages Tom et SL

Nous présentons dans cette section au travers d'exemples fondés sur le langage de programmation présenté en figure 4.1 l'interaction entre Tom et le langage de stratégies SL.

4.3.1 Interaction avec Java dans les stratégies élémentaires

La syntaxe de la construction `%strategy` permet d'encoder plus que des systèmes de règles. En effet, il existe une syntaxe plus générale qui permet de définir une action Java en partie droite des règles (similaire à la syntaxe de l'instruction `%match`) :

```
%strategy PrintCst() extends Fail() {
  visit Expression {
    Cst(x) -> { System.out.println("cst:␣" + ‘x); }
    Var(x) -> { System.out.println("var:␣" + ‘x); }
  }
}
```

Lorsque la stratégie élémentaire `PrintCst()` est appliquée sur un nœud de type `Expression`, comme dans une construction `%match`, les motifs sont évalués de manière séquentielle et les actions Java correspondantes sont déclenchées. Si une instruction Java `return` est exécutée alors une transformation est réalisée à la position courante. Sinon, la stratégie par défaut est exécutée (`Fail` dans l'exemple ci-dessus). La syntaxe allégée où l'on peut directement écrire `l -> r` peut être vue comme du sucre syntaxique pour `l -> { return ‘r;}`.

Afin de profiter pleinement des parties droites Java, il est nécessaire de pouvoir partager une information contextuelle entre les stratégies combinées. Pour cela, Tom permet de passer des paramètres Java à la construction `%strategy`. En particulier, collecter des données peut être réalisé en paramétrant la construction `%strategy` par une structure de données Java. Dans la stratégie suivante, nous collectons dans un ensemble Java (instance de la classe `Set`) le nom des variables présentes dans un programme :

```
%strategy CollectVar(set:Set) extends Identity() {
  visit Expression {
    Var(v) -> { set.add(‘v); }
  }
}
```

Comme les stratégies élémentaires ne s'appliquent qu'en position racine du sujet, elles n'ont d'intérêt que combinées avec des opérateurs de traversée. Appliquer la stratégie `CollectVar` sur le programme `Declare("x",Cst(1),Print(Var("x")))` ne collecte pas la variable `x` puisque le symbole de tête du programme est `Declare`. le motif `Var(v)` ne filtre donc pas ce programme en tête. Pour collecter réellement l'ensemble des variables du programme, il faut appliquer la stratégie élémentaire `CollectVar` sur tous les nœuds de l'arbre (en utilisant `TopDown` par exemple).

4.3.2 Réflexivité par ancrage

L'une des principales originalités du langage Tom est sa capacité à filtrer n'importe quelle structure de données grâce aux mécanisme d'ancrage. En particulier, il est possible

de filtrer les objets utilisés dans la bibliothèque qui implémente le langage SL. Par conséquent, n'importe quelle stratégie SL peut être vue comme un terme Tom.

Le mécanisme d'ancrages permet donc de définir très facilement des nouveaux opérateurs de stratégie. Par exemple, l'un des ancrages proposés par la librarie SL est celui de l'opérateur `Sequence` d'arité variable défini uniquement à partir de la classe `Sequence` (qui implémente un combinateur d'arité 2) :

```
%oplist Strategy Sequence(Strategy*) {
  is_fsym(t) { (t instanceof Sequence) }
  make_empty() { null }
  make_insert(head,tail) { (tail==null)?head:‘Sequence(head,tail) }
  get_head(t) { t.getChildAt(Sequence.FIRST) }
  get_tail(t) { t.getChildAt(Sequence.THEN) }
  is_empty(t) { (t == null) }
}
```

À l'aide de cet ancrage, on peut définir par exemple en Tom la stratégie suivante :

```
Strategy s = ‘Sequence(s1 ,s2 ,s3 ,s4);
```

au lieu du code Java équivalent :

```
Strategy s = new Sequence(s1, new Sequence(s2, new Sequence(s3, s4)));
```

À partir de ces ancrages, l'utilisation de Tom couplée à SL permet de filtrer, transformer ou créer dynamiquement un terme de stratégie et une stratégie peut s'appliquer sur une stratégie. Cela offre des capacités réflexives qui peuvent être utilisées par exemple pour effectuer des optimisations/compilations à la volée de stratégies.

On peut transformer une stratégie en une stratégie opérationnellement équivalente mais plus efficace. On peut par exemple définir un système de règles qui normalise un terme de stratégie en une nouvelle stratégie plus efficace :

```
%strategy Optimize() extends Identity() {
  Sequence(Identity(), x) -> x
  Choice(Fail(), x)     -> x
  Not(One(Not(x)))      -> All(x)
  Not(All(Not(x)))      -> One(x)
}
```

Ces mécanismes réflexifs sont par exemple utilisés pour enrichir une stratégie avec des opérateurs de déboguage. Pour illustrer ce type de mécanisme, on définit la stratégie suivante qui tisse dans le terme de stratégie un point d'arrêt avant chaque opérateur de traversée descendante :

```
%strategy AddBreakPoint() extends Identity() {
  t@All(MuVar(x)) -> Sequence(bp(),t)
  t@One(MuVar(x)) -> Sequence(bp(),t)
}
```

La stratégie `bp()` est une stratégie élémentaire implémentée en Java qui définit un point d'arrêt. On peut appliquer la stratégie composée `BottomUp(AddBreakPoint())` pour ajouter des points d'arrêts avant chaque descente récursive (stratégies de la forme `All(MuVar(x))` ou `One(MuVar(x))`) dans un terme de stratégie complexe.

4.3.3 Génération de stratégies de congruence et de construction

Tom fournit une construction spéciale (voir [Rei06a] pour plus d'explications) pour décrire une signature typée algébrique. Étant donnée une signature, il génère une implémentation Java et fournit un support pour la traversée des termes.

En particulier, des opérateurs de congruence et de construction sont générés automatiquement. Ceux-ci sont utilisés pour discriminer les constructeurs au niveau des stratégies et ainsi permettre de décrire les stratégies d'ordre supérieur tels que la fonction *map* (qui applique une stratégie à chaque élément d'une liste et retourne la liste modifiée). Prenons par exemple la signature suivante :

```
List = Cons(head:Element,tail:List)
     | Nil()
```

À partir de cette signature, Tom génère automatiquement les opérateurs de congruence `_Cons(s1,s2)` et `_Nil()`. L'arité d'un opérateur de congruence est la même que celle du constructeur associé et tous les arguments sont des stratégies.

$$\frac{\langle S\{\texttt{\_Cons}(\underline{s_1}, s_2)\}, t, \omega \cdot 1\rangle \to t[t_1]_{\omega.1}\langle S\{\texttt{\_Cons}(s_1, \underline{s_2})\}, t, \omega \cdot 2\rangle \to t[t_2]_{\omega.2}}{\langle S\{\underline{\texttt{\_Cons}(s_1, s_2)}\}, t, \omega\rangle \to t[t_1]_{\omega.1}[t_2]_{\omega.n}}\ symb(t_{|\omega}) = \texttt{Cons}$$

$$\frac{\langle S\{\texttt{\_Cons}(\underline{s_1}, s_2)\}, t, \omega \cdot 1\rangle \to \mathsf{fail}}{\langle S\{\underline{\texttt{\_Cons}(s_1, s_2)}\}, t, \omega\rangle \to \mathsf{fail}}\ symb(t_{|\omega}) = \texttt{Cons}$$

$$\frac{\langle S\{\texttt{\_Cons}(s_1, \underline{s_2})\}, t, \omega \cdot 2\rangle \to \mathsf{fail}}{\langle S\{\underline{\texttt{\_Cons}(s_1, s_2)}\}, t, \omega\rangle \to \mathsf{fail}}\ symb(t_{|\omega}) = \texttt{Cons}$$

$$\frac{}{\langle S\{\underline{\texttt{\_Cons}(s_1, s_2)}\}, t, \omega\rangle \to \mathsf{fail}}\ symb(t_{|\omega}) \neq \texttt{Cons}$$

$$\frac{}{\langle S\{\underline{\texttt{\_Nil()}}\}, t, \omega\rangle \to t}\ symb(t_{|\omega}) = \texttt{Nil}$$

$$\frac{}{\langle S\{\underline{\texttt{\_Nil()}}\}, t, \omega\rangle \to \mathsf{fail}}\ symb(t_{|\omega}) \neq \texttt{Nil}$$

La stratégie `map(s)` peut donc être définie par :

```
Strategy map = `Mu("x",Choice(_Cons(s,MuVar("x")),_Nil()));
```

L'application de cette stratégie teste tout d'abord si le sujet a comme symbole de tête `Cons`. Si ce test réussit alors `s` est appliqué sur le premier fils de `Cons` (i.e. la tête de la liste) et `map(s)` est appelée récursivement sur son second fils (i.e. la queue de la liste). Si le sujet n'a pas comme symbole de tête `Cons`, la stratégie `_Nil` est appliquée pour vérifier qu'on ait bien à la fin de la liste.

Tom génère aussi des opérateurs de construction préfixés par `Make_` permettant de construire des termes au niveau des stratégies. Ce type de stratégie ne tient donc pas compte du sujet.

$$\frac{\langle S\{\texttt{Make\_Cons}(\underline{s_1}, s_2)\}, t, \omega \cdot 1\rangle \to t[t_1]_{\omega\cdot 1} \langle S\{\texttt{Make\_Cons}(s_1, \underline{s_2})\}, t, \omega \cdot 2\rangle \to t[t_2]_{\omega\cdot 2}}{\langle S\{\underline{\texttt{Make\_Cons}(s_1, s_2)}\}, t, \omega\rangle \to t[Cons(t_1, t_2)]_\omega}$$

$$\frac{\langle S\{\texttt{Make\_Cons}(\underline{s_1}, s_2)\}, t, \omega \cdot 1\rangle \to \mathsf{fail}}{\langle S\{\underline{\texttt{Make\_Cons}(s_1, s_2)}\}, t, \omega\rangle \to \mathsf{fail}}$$

$$\frac{\langle S\{\texttt{Make\_Cons}(s_1, \underline{s_2})\}, t, \omega \cdot 2\rangle \to \mathsf{fail}}{\langle S\{\underline{\texttt{Make\_Cons}(s_1, s_2)}\}, t, \omega\rangle \to \mathsf{fail}}$$

$$\frac{}{\langle S\{\underline{\texttt{Make\_Nil}()}\}, t, \omega\rangle \to t[Nil()]_\omega}$$

En définissant deux stratégies `Get` et `Set` partageant une même `HashMap` en interne, nous pouvons implémenter la règle de réécriture $f(x) \to g(x)$ par la stratégie :

```
Sequence(_f(Set("x")),Make_g(Get("x")))
```

Notons qu'il est intéressant d'implémenter une règle de réécriture au niveau des stratégies seulement lorsqu'il est nécessaire de la modifier dynamiquement, en utilisant les traits réflexifs de SL.

Avec de telles constructions, on peut par exemple encoder des meta-règles telles que $f(X) \to g(Y) \Rightarrow g(X) \to f(Y)$. Cette meta-règle permet par exemple de transformer la règle $f(g(x)) \to g(x)$ (encodée par `Sequence(_f(_g(Set("x"))),Make_g(Get("x")))`.) en la règle $g(g(x)) \to f(x)$. Cette méta-règle s'encode à l'aide de la stratégie suivante :

```
%strategy MetaRule() extends Identity() {
  visit Strategy {
    Sequence(_f(X),Make_g(Y)) -> Sequence(_g(X),Make_f(Y))
  }
}
```

Si on applique `MetaRule()` sur `Sequence(_f(_g(Set("x"))),Make_g(Get("x")))`, on obtient comme résultat `Sequence(_g(_g(Set("x"))),Make_f(Get("x")))`, les variables X et Y du motif ayant été respectivement instanciées par `_g(Set("x"))` et `Get("x")`.

On peut ainsi utiliser toutes les constructions de Tom (filtrage, stratégie) pour modifier dynamiquement un ensemble de règles en fonction du contexte d'exécution.

4.3.4 Environnement d'évaluation : réification des positions

Une des principales originalités du langage SL est la capacité de chaque stratégie à connaître la position à laquelle elle s'applique. Cette position permet d'identifier sur quel sous-terme du sujet global est appliquée la stratégie. Ces positions sont représentées par des objets Java de la classe `Position`.

La classe `Position` offre plusieurs méthodes permettant de construire des stratégies. La méthode `Strategy getOmega(Strategy s)` retourne une stratégie qui applique la stratégie `s` au sous-terme situé à la position représentée par l'instance de la classe. La méthode `Strategy getSubterm()` retourne une stratégie qui lorsqu'elle est appliquée à un terme `t` retourne le sous-terme de `t` situé à cette position. La méthode `Strategy getReplace(Object t)` retourne une stratégie qui remplace par `t` le sous-terme situé à la position représentée par l'instance de la classe.

Dans une stratégie, la méthode `getPosition()` permet à l'utilisateur d'accéder à la position courante d'application. Par exemple, la stratégie `CollectVarPos` utilisée sous une stratégie `TopDown` collecte l'ensemble des positions de la variable dont le nom est donné par le paramètre `nameVar`. Cet ensemble est collecté dans l'objet de type `Set` donné lui-aussi en paramètre :

```
%strategy CollectVarPos(set:Set,nameVar:String) {
  visit Expression {
    Var(name) && name == nameVar -> { set.add(getPosition()); }
  }
}
```

Lorsque l'on manipule des systèmes de règles non confluents, il est souvent utile de calculer l'ensemble de tous les successeurs possibles. Pour résoudre ce problème, nous avons besoin de trouver tous les réduits et pour chacun d'eux de calculer toutes les substitutions solutions du problème de filtrage. Autrement dit, nous devons calculer :

$\{t[\sigma_1(r)]_{\omega_1}, t[\sigma_2(r)]_{\omega_1}, \ldots, t[\sigma_p(r)]_{\omega_n}, \ldots, t[\sigma_q(r)]_{\omega_n}\}$, où ω_i représente la position d'un réduit, et σ_j est une substitution telle que $\sigma_j(l) = t_{|\omega_i}$.

Contrairement aux autres langages à base de règles, la réification des positions rend possible l'implémentation d'un tel calcul. En effet, pour calculer l'ensemble de tous les successeurs de t, il suffit d'appliquer de manière *top-down* la stratégie `Collect` paramétrée par le terme t et par une collection Java. Pour chaque position ω où un des motifs filtre, la stratégie mémorise $t[\sigma(r)]_\omega$ dans `bag`, en utilisant `getPosition()` pour obtenir ω et `getReplace` pour réaliser $t[tt]_p$.

```
%strategy Collect(e:Expression, bag:Collection) extends Identity() {
  visit Expression {
    Plus(Cst(c1),Cst(c2)) ->
      { bag.add(getPosition().getReplace(`Cst(c1 + c2)).visit(e)); }
```

```
    Mult(Cst(c1),Cst(c2)) ->
      { bag.add(getPosition().getReplace(‘Cst(c1 * c2)).visit(e)); }
  }
}
Expression e = ‘Plus(Mult(Cst(1),Cst(2)),Plus(Cst(3),Cst(4)));
‘TopDown(Collect(e,bag)).visit(e);
```

Après évaluation, la variable `bag` contient tous les résultats intermédiaires après un pas de réécriture, c'est à dire les deux termes `Plus(Cst(2),Plus(Cst(3),Cst(4)))` et `Plus(Mult(Cst(1),Cst(2)),Cst(7))`. Le langage Tom est le premier langage à réifier cette notion de position, ce qui permet d'écrire des programmes encore plus proche des formalismes qui utilisent cette notion. Enfin, les positions s'avèrent très utiles pour l'implémentation d'analyseurs où il est souvent nécessaire de traiter des problèmes d'atteignabilité.

La réification des positions a donc deux principaux intérêts :

- cela permet de définir des stratégies plus complexes. En effet, le contrôle de l'évaluation des stratégies peut dépendre non seulement des informations relatives au nœud lui-même, mais aussi de sa position dans l'arbre.
- cela permet de gérer le non-déterminisme, ce qui n'est possible ni dans JJTraveler, ni dans Stratego. On peut ainsi par exemple calculer l'ensemble des termes atteignables par un système de règles pour un sujet donné. Une application intéressante est la construction de traces de réécriture en mémorisant les règles et les positions à laquelle elles ont été appliquées. Ces traces peuvent alors servir soit à réaliser du *debugging*, soit à certifier des preuves comme dans la thèse de Quang Huy Nguyen [Ngu02].

4.4 Comparaison avec la bibliothèque JJTraveler

La bibliothèque SL se distingue essentiellement de JJTraveler par les trois points suivants :

- le combinateur de récursion `Mu` est explicite ce qui permet de limiter le nombre de briques de base à celui du langage Stratego (contrairement à JJTraveler qui en possède plus de vingt),
- l'environnement d'évaluation permet la réification des positions et la définition du combinateur `Up`,
- le support pour le langage Tom qui permet au programmeur Tom d'utiliser SL sans connaître son interface. En effet, la construction `%strategy` et les ancrages Tom des combinateurs SL peuvent être vus comme un langage dédié à la programmation par stratégies fondé sur la bibliothèque SL.

Opérateur de récursion explicite

La bibliothèque JJTraveler ne propose pas le combinateur de récursion μ explicite de Stratego. Les stratégies récursives génériques telles que `BottomUp` ne sont pas définies

par composition mais directement comme une brique du langage. Comme la récursion est définie à l'aide d'un graphe d'objets `Visitor`, les briques telles que `BottomUp` sont implémentées en utilisant des astuces de programmation qui seront présentées dans la sous-section 4.7.2.

La bibliothèque JJTraveler fournit donc plus de vingt briques de base telles que `TopDown` et `DoWhileSuccess` pour compenser le manque d'expressivité dû à l'absence de combinateur `Mu`. La composition de stratégies est donc finalement assez limitée. Enfin, les astuces de programmation utilisées dénaturent l'adéquation entre la définition formelle et le code.

Pour éviter ces désagréments, la bibliothèque SL propose un opérateur de récursion explicite `Mu` dont l'implémentation est présentée dans la sous-section 4.7.2. Les stratégies se composent donc exactement comme dans les définitions formelles :

```
%op Strategy BottomUp(s:Strategy) {
  make(s) { `Mu("x",new Sequence(new All(MuVar("x")),s)) }
}
```

Environnement d'évaluation

Contrairement à l'interface `VisitableVisitor` de JJTraveler, la méthode `visit`, en plus d'appliquer la stratégie sur le sujet, maintient un environnement. À l'origine, ce concept d'*environnement* avait été introduit dans la bibliothèque SL afin de pouvoir fournir à l'utilisateur la position courante d'évaluation de la stratégie dans le sujet. Nous verrons dans le Chapitre 5 comment ce concept a été étendu afin de pouvoir traverser des structures de termes-graphes.

Support pour le langage Tom

Contrairement à JJTraveler, la bibliothèque SL peut soit être utilisée dans une application Java pure, soit tirer avantage de son interaction avec le langage Tom.

Tout d'abord, la bibliothèque SL propose des ancrages Tom pour l'ensemble du langage de stratégies, ce qui permet d'offrir très facilement de la réflexivité.

De plus, Tom offre un outil [Rei06a] qui génère à partir d'une signature algébrique un ensemble de classes Java implémentant l'interface `Visitable` et donc directement utilisable par SL. L'outil JJForester [KV00] est un outil similaire pour JJTraveler. À partir d'une grammaire spécifiée dans le format SDF [BHKO02], JJForester génère une structure de données Java *visitable* pour JJTraveler et le générateur de parseur développé pour le format SDF permet directement d'obtenir un parseur pour la grammaire spécifiée. Par rapport à l'outil fourni par Tom, les deux principales limitations de JJForester sont l'absence de constructions pour enrichir les classes générées (systèmes de *hooks* présentés dans le chapitre 2) et l'absence de partage maximal sur les structures de données.

Enfin, la construction Tom `%strategy` permet de définir de manière déclarative un système de règles comme une stratégie élémentaire. Chaque instruction `%strategy` est compilée en une classe statique interne implémentant l'interface `Strategy`. Cela simplifie

énormément l'utilisation de la bibliothèque puisque l'utilisateur n'a pas réellement besoin de connaître l'interface de la bibliothèque pour pouvoir l'utiliser. De plus, la vérification de typage de Tom assure que la stratégie générée préserve les types. Contrairement à JJTraveler, il n'y a pas besoin d'imposer aux programmeurs l'utilisation d'un *design pattern* complexe introduisant du *double-dispatch*.

Autrement dit, l'instruction `%strategy` et les ancrages des combinateurs peuvent être présentés comme un DSL dédié à la programmation par stratégies. Ce DSL fondé sur la bibliothèque SL est embarqué dans Java ce qui permet d'obtenir un code plus déclaratif et mieux typé.

4.5 Un langage de stratégies adapté à l'analyse de programmes

Nous allons détailler dans cette section en quoi trois caractéristiques originales du langage SL le rendent mieux adapté à l'analyse de programmes que les autres langages de stratégies. Ces caractéristiques sont les suivantes :

- l'environnement d'exécution des stratégies permettant de connaître le contexte d'évaluation (en particulier la position dans le terme global) mais aussi de définir des parcours plus complexes comme ceux dans les graphes,
- l'extensibilité du langage,
- l'indépendance du langage par rapport aux structures de données.

4.5.1 Environnement d'évaluation

L'environnement d'évaluation présenté dans la section 4.3.4 est un atout dans le développement de transformations et d'analyses de programmes par stratégies. Nous allons donc montrer au travers de trois utilisations possibles en quoi cette caractéristique originale de SL facilite l'implémentation d'analyses de programmes et le rend donc plus adapté que les autres langages de stratégies.

Identifier des sous-termes dans des structures avec partage maximal

Les ASTs peuvent comporter plusieurs milliers de nœuds et il est donc intéressant de disposer d'une représentation mémoire efficace des termes. Une technique classique utilisée dans les langages à base de règles mais aussi les langages fonctionnels est le partage maximal de sous-termes qui assure qu'une seule instance de chaque sous-terme existe dans la mémoire. Cela est souvent réalisé par *hash-consing* [All78]. Le hash-consing consiste à assurer que si le même nœud est construit deux fois, un pointeur sur le nœud déjà existant est retourné. En plus d'optimiser l'usage de la mémoire, cette technique permet d'avoir un test d'égalité de termes en temps constant, puisqu'il suffit simplement de comparer les pointeurs.

Lorsqu'on définit des analyses sur de telles structures, il est nécessaire de pouvoir identifier de manière unique un sous-terme, pour par exemple utiliser cette référence comme clé dans une table de hachage. Or un des inconvénients des structures avec

partage maximal est que les pointeurs ne permettent pas d'identifier un sous-terme. En effet, un pointeur peut représenter plusieurs sous-termes égaux à cause du *hash-consing*.

La réification des positions grâce aux environnements est donc un moyen naturel d'identifier un sous-terme dans des structures avec partage maximal. Dans les parties droites des actions de la construction `%strategy`, il est possible d'identifier le sous-terme courant par sa position (obtenue par la méthode `getPosition()`).

Combinateur de point-fixe pour les graphes de flot

La plupart du temps, l'évaluation de stratégies sur des graphes ne termine pas. En effet, appliquer simplement la stratégie `TopDown(s)` sur un terme-graphe ayant des cycles est non-terminant. Cependant, grâce au concept d'environnement du langage SL, il est possible de détecter un état stationnaire si il existe.

Pour cela, nous définissons un nouvel opérateur de récursion `MuCycle` ayant presque la même sémantique que `Mu`. La seule différence est qu'à chaque appel récursif, l'opérateur `MuCycle` mémorise une copie de l'environnement et associe cette copie à lui même dans une table de hachage partagée. Ainsi, si l'opérateur `MuCycle` est évalué dans le même environnement que lors de son dernier appel, il s'arrête et retourne l'identité. Ce type d'opérateur est indispensable pour l'analyse et la transformation de structures de graphes. Des exemples d'utilisation seront présentés dans le Chapitre 5.

Analyse locale ascendante

L'environnement permet de décrire des traversées ascendantes puisqu'il est possible d'accéder au père du nœud courant. C'est grâce à la disponibilité de cette information que le combinateur de stratégies `Up` a pu être défini. Cet opérateur a peu d'intérêt dans une utilisation classique des stratégies (contrôle de l'application de règles de réécriture). En revanche, dans le contexte de l'analyse de programmes, il est parfois nécessaire de décrire des traversées ascendantes à l'intérieur de l'AST. Or ce type de parcours ne peut pas être défini avec les opérateurs de Stratego ou de JJTraveler.

Un exemple d'utilisation de ce combinateur sera présenté en détail dans le Chapitre 6. Dans l'implémentation des outils de refactoring, on a parfois besoin de calculer une information locale sans parcourir l'ensemble de l'arbre. L'analyse de noms en Java demande par exemple de parcourir l'AST depuis l'utilisation de l'identifiant jusqu'à la déclaration correspondante.

Un autre exemple d'utilisation est la définition d'optimisations sous forme de règles de réécriture conditionnelles à base de logique temporelle. Certaines optimisations sont parfois fondées sur des opérateurs arrières qui réalisent des vérifications en remontant dans l'arborescence de traces [Lac03]. Ce type d'opérateur ne peut être encodé qu'à l'aide d'un combinateur tel que `Up`. L'encodage par stratégies de la logique temporelle arborescente sera présenté dans la section 4.6.

4.5.2 Extensibilité et modularité du langage

Afin de pouvoir définir des analyses et des transformations complexes, il est primordial que le langage SL permette de définir une stratégie de manière modulaire. De plus, certains traitements spécifiques peuvent nécessiter l'introduction de nouveaux combinateurs. Le langage doit donc être facilement extensible et modulaire.

Définition de nouveaux combinateurs

Comme cela sera présenté dans la Section 4.7, l'architecture de la librairie permet facilement de définir de nouveaux combinateurs. La classe abstraite `AbstractCombinator` fournit une implémentation partielle de l'interface `Strategy`. On peut donc définir un nouveau combinateur en étendant `AbstractCombinator`.

Exemple 12 (Arrêt des stratégies sur les formes normales)**.** La plupart du temps, la stratégie `Fail` n'est pas utilisée pour détecter des cas concrets d'erreurs mais seulement pour contrôler l'application ou l'arrêter. Par exemple, la stratégie composée `Repeat` utilise l'échec pour détecter les formes normales. Comme l'échec est réalisé dans notre bibliothèque à l'aide du mécanisme d'exceptions de Java, une utilisation trop intensive de l'échec peut avoir un coût non négligeable. En particulier, la transformation de programmes nécessite souvent le calcul de nombreux points fixes. En effet, on compose des règles de transformation en stratégies. Ces stratégies sont souvent appliquées elles-mêmes sous un combinateur `Innermost` qui sont ensuite composées.

La réalisation d'un optimiseur [BM06] pour Tom utilisant intensivement les stratégies nous a amené à proposer dans la bibliothèque des variantes des combinateurs de base où la notion d'échec a été remplacée par l'identité. En effet, dans un optimiseur, la plupart des stratégies normalisent l'arbre abstrait du programme à partir d'un ensemble de règles. Or si on utilise la stratégie composée `Innermost` fondée sur l'échec, à chaque fois qu'une règle ne peut pas s'appliquer, une exception Java est déclenchée. Ce mécanisme est coûteux en temps et dans le cas de la normalisation n'est pas indispensable. On peut donc modéliser l'échec d'une règle par l'identité (une règle qui ne peut pas s'appliquer ne modifie pas le terme courant).

Cette extension améliore considérablement les performances en temps lors d'un calcul de forme normale. Par exemple, dans le cas de l'optimiseur de Tom, le temps d'optimisation d'une centaine de règles de filtrage syntaxique est passé de 90 minutes à 3 minutes.

Le principe de ces nouveaux opérateurs est de considérer comme un échec le fait qu'une stratégie ne modifie pas un terme mais sans lever d'exception Java. Par exemple, on peut définir le combinateur `SequenceId` comme suit :

`SequenceId(s1,s2).visit(t)` = `s2.visit(t')` si `t'`=`s1.visit(t)` et `t'`≠`t`
`t` sinon.

À l'aide de ce nouvel opérateur, on peut redéfinir une variante du combinateur `Innermost` plus efficace. Le combinateur `InnermostId` est défini comme suit :

```
InnermostId(s) = Mu("x",Sequence(All(MuVar("x")),
```

```
SequenceId(s,MuVar("x"))))
```

Dans ce cas, la stratégie `s` doit adopter la stratégie `Identity` comme stratégie par défaut au lieu de la stratégie `Fail`.

Exemple 13 (Combinateurs probabilistes)**.** La classe `Random` Java permet de définir facilement des combinateurs probabilistes. Par exemple, la bibliothèque propose le combinateur de stratégies probabiliste suivant :
`Pselect(p:int,q:int,s1:Strategy,s2:Strategy)`
`Pselect` applique la stratégie `s1` avec une probabilité de $\frac{p}{q}$ et `s2` avec une probabilité de $\frac{q-p}{q}$. à Partir de ce combinateur, il est maintenant possible de définir en Tom un alias réalisant un choix indéterministe :

```
%op Strategy UndeterministicChoice(s1:Strategy, s2:Strategy) {
  make(s1,s2) { `Pselect(1,2,s1,s2) }
}
```

Supposons maintenant que nous voulions tester le compilateur du petit langage présenté dans la figure 4.1. Pour cela, il faut pouvoir générer de manière aléatoire des ASTs à partir de la grammaire. Nous supposons que la stratégie `UndeterministicChoice` peut prendre un nombre arbitraire d'arguments. Le générateur peut s'écrire uniquement à l'aide de stratégies :

```
Strategy generateInstruction = `Mu("x",
  UndeterministicChoice(
      Make_Sequence(MuVar("x"),MuVar("x")),
      Make_Declare(generateName,generateExpression,MuVar("x")),
      Make_Assign(generateName,generateExpression),
      Make_Print(generateExpression)
  ))
```

où `generateName` est une stratégie élémentaire qui génère des noms de variables de manière aléatoire et `generateExpression` est une stratégie similaire à `generateInstruction` qui génère aléatoirement des termes de type `Expression`. Chaque appel à la stratégie `generateInstruction` construit un nouveau programme.

Cette technique a été utilisée pour générer aléatoirement des programmes écrits dans un sous-ensemble de Java. Le but était de pouvoir tester les outils de *refactoring* développés dans l'équipe *Programming Tools* de l'université d'Oxford.

Modularité des stratégies complexes

Le mécanisme des ancrages de Tom offre un moyen élégant de définir des macros (partie constructive). Dans le cas de stratégies complexes, cela permet de découper l'expression de la stratégie globale en sous-stratégies de la même manière que l'on définirait des fonctions intermédiaires dans un langage fonctionnel. De plus, à la création, l'utilisation du *backquote* permet d'éviter les imbrications de `new`.

Par exemple, dans le cas d'analyses Java, on a souvent envie de considérer la classe englobante depuis n'importe quel point de programme. Pour cela, il suffit d'écrire une stratégie récursive utilisant le combinateur `Up` et qui remonte jusqu'à la déclaration de la classe :

```
%op Strategy ApplyAtEnclosingClass(s:Strategy) {
  make(s) { `Mu(MuVar("x"),IfThenElse(Is_ClassDecl(),s,Up(MuVar("x")))) }
}
```

Dans le chapitre 6, nous présentons en détail l'implémentation de l'analyse de noms en Java. L'utilisation des ancrages a permis de définir de manière modulaire cette stratégie qui nécessite plus d'une centaine de combinateurs. L'avantage d'utiliser des ancrages à la place de fonctions statiques Java équivalentes est que le compilateur du langage Tom *inline* ces définitions comme dans un système de macros. Le code résultant est donc plus efficace.

4.5.3 Introspection des structures de données par ancrage

Jusqu'à présent, la bibliothèque SL n'était pas restreinte à une représentation particulière de termes mais la seule contrainte à respecter était que la structure de données manipulée implémente une interface permettant d'expliciter la structure arborescente des objets manipulés. Cependant, cette condition s'avère parfois trop contraignante. En effet, l'utilisation du langage Tom se veut le moins intrusif possible dans le projet Java et une telle dépendance avec la bibliothèque SL est la plupart du temps non souhaitée. De plus, cette condition peut parfois ne pas être satisfaite du tout, lorsque les sources de la structure de donnée ne sont pas disponibles par exemple. Dans le contexte de l'analyse de programmes, une telle condition ne peut généralement être respectée car les structures de données représentant l'AST sont la plupart du temps générées à partir d'une spécification de grammaire et il n'est donc pas du tout raisonnable d'aller modifier du code qui est généré. Une des contributions de cette thèse a donc été de proposer un mécanisme fondé sur les ancrages de Tom permettant de supporter automatiquement la bibliothèque SL sans modifier les structures de données de l'utilisateur. L'utilisateur n'a donc qu'à fournir des ancrages pour pouvoir utiliser à la fois le langage Tom et le langage de stratégies SL.

Pour faciliter encore plus l'intégration de programme Tom dans un programme Java, il existe un générateur d'ancrages [KM08] qui construit automatiquement des ancrages Tom à partir d'une hiérarchie de classes Java en utilisant la réflexivité du langage Java et la présence de *getters* et *setters*.

Dans le cas de l'analyse de programmes, il est en général assez facile de générer automatiquement les ancrages à partir de grammaires. Durant cette thèse, nous avons par exemple proposé un générateur d'ancrages pour les grammaires utilisées par l'outil JastAdd [EH07a] permettant de définir des grammaires attribuées en Java. Il a été ainsi possible de connecter en moins d'une journée le langage Tom à celui de JastAdd.

4.6 Simulation des opérateurs de logiques temporelles

Dans cette section, nous allons montrer comment coder les opérateurs de la logigue temporelle arborescente dans le langage SL. L'intérêt pratique est de pouvoir exprimer de manière concise des analyses et de pouvoir raisonner sur les stratégies correspondantes. David Lacey a montré dans sa thèse [LM01] que ce type de formules logiques couplées à la réécriture permet de définir des optimisations de programmes complexes de manière concise et déclarative.

Chaque formule CTL peut être traduite en une expression de stratégies. Au lieu d'utiliser un *model-checker* pour vérifier la formule, le mécanisme d'évaluation des stratégies offre alors une procédure de vérification de cette formule puisque les structures manipulées sont des arbres finis. Si la stratégie termine sans échouer alors la formule correspondante est vraie

4.6.1 Introduction à la logique CTL

Logiques temporelles

Une partie des logiques temporelles se fonde sur une approche modale du temps. Contrairement à l'approche prédicative, la conception modale du temps est proche de celle de la langue naturelle. En effet, elles consistent à étendre une logique (en général la logique propositionnelle) par un ensemble de *modalités* permettant d'exprimer de l'information qualitative sur le temps (par exemple les expressions *un jour*, *toujours*).

Les logiques temporelles ont trouvé de nombreuses applications dans la vérification de systèmes informatiques et en particulier dans la modélisation des systèmes concurrents et répartis qui nécessitent de raisonner sur des états qui dépendent temporellement les uns des autres. Le plus connu des langages de spécification est TLA+ de Leslie Lamport. Il est fondé à la fois sur une logique temporelle et une logique d'actions.

On distingue principalement deux types de logiques temporelles suivant la manière dont est interprété le temps. Dans la logique temporelle linéaire [Pnu77] (notée LTL), le temps est linéaire et le futur est déterminé à l'avance. Lorsqu'on veut considérer plusieurs alternatives, il faut analyser plusieurs traces séparément. Dans la logique temporelle arborescente [CES86] (notée CTL), le modèle du temps prend en compte tous les futurs possibles sous la forme de branchements. Le temps n'est plus linéaire mais arborescent et on peut donc raisonner sur plusieurs branches simultanément.

CTL

La logique temporelle arborescente (CTL) a été introduite en 1986 par Edmund Clarke et E. Allen Emerson [CES86] afin de pouvoir raisonner sur plusieurs traces simultanément. Le temps y est représenté sous une forme arborescente potentiellement infinie. Chaque branche de l'arbre correspond à une succession possible d'états dans le temps et les branchements permettent d'indiquer des points de choix. Autrement dit, le futur n'est pas déterminé à l'avance. Comme pour toutes les logiques temporelles, le modèle

d'interprétation considéré est un système de transitions $\mathcal{M} = \{S, \rightarrow, F\}$ où S est l'ensemble des états, $\rightarrow$ la relation de transitions et F la fonction d'étiquetage des états. Une formule CTL ϕ est donc vérifiée pour un état s du modèle $\mathcal{M}$ et se note $\mathcal{M}, s \models \phi$.

En plus des opérateurs de logique usuels comme $\neg$ ou $\wedge$, les formules temporelles sont composées d'opérateurs temporels. En CTL, un opérateur temporel est la combinaison d'un opérateur de chemin suivi d'un opérateur d'état. Le concept d'opérateurs de chemin n'existe pas en LTL.

Il existe deux opérateurs de chemin :
- $A\phi$ est vérifiée si ϕ est vérifiée pour tous les chemins à partir du nœud courant,
- $E\phi$ est vérifiée si ϕ est vérifiée pour au moins un chemin à partir du nœud courant.

Les opérateurs d'états proviennent de LTL et informent sur le futur des chemins :
- $X\phi$ est vérifiée si ϕ est vérifiée par le prochain état du chemin,
- $G\phi$ est vérifiée si ϕ est vérifiée par tous les états du reste du chemin, l'état courant inclus,
- $F\phi$ est vérifiée si ϕ est finalement vérifiée (quelque part dans le reste du chemin),
- $\phi U \phi'$ est vérifiée si ϕ est vérifiée jusqu'à atteindre un état qui vérifie ϕ'. Cela implique que ϕ' doit nécessairement être vérifiée dans le futur.

Ce type de logique permet de définir en particulier des propriétés de sûreté (i.e. $AG\neg\phi$: ϕ n'est *jamais vérifiée*) et de vivacité (i.e. $AG(\phi_1 \Rightarrow EX\phi_2)$: quand ϕ est vrai, il est possible d'obtenir par la suite un état tel que ϕ_2 soit vrai).

4.6.2 Encodage des formules CTL en stratégies SL

Nous allons voir maintenant comment coder les formules CTL par des stratégies. L'idée de base est de représenter un prédicat p par une stratégie élémentaire. Les opérateurs temporels et les connecteurs logiques sont définis par composition des combinateurs de stratégie. Au lieu d'utiliser un *model-checker* pour vérifier une formule temporelle sur l'AST, on évalue la stratégie qui simule la vérification de la formule sur l'AST à la position correspondant à l'état courant. Si la stratégie termine sans échouer alors la formule est vérifiée.

Étant donné un terme $t \in \mathcal{T}(\mathcal{F})$ et un ensemble de prédicats $\mathcal{P}$, on considère donc ici comme modèle d'interprétation des formules temporelles sur t le système de transition étiqueté $\mathcal{M} = \{S, \rightarrow, F\}$ où l'ensemble des états correspond à $\mathcal{P}os(t)$. La relation de transition considérée est la relation père/fils des positions. Plus formellement, $\forall \omega_1, \omega_2 \in \mathcal{P}os(t)$, $\omega_1 \rightarrow \omega_2$ si et seulement si $\exists i \in \mathbb{N}, \omega_2 = \omega_1 \centerdot i$. La fonction d'étiquetage $F : \mathcal{P}os(t) \rightarrow 2^{\mathcal{P}}$ associe à chaque position un ensemble de prédicats. Nous pouvons remarquer que comme t est fini et qu'il s'agit d'un terme non cyclique, les suites de transitions du modèle $\mathcal{M}$ correspondant sont donc toujours finies.

La figure 4.3 présente la fonction d'encodage $[\![\ ]\!]$ des formules CTL vers les stratégies. Les valeurs booléennes *false* et *true* sont représentées par les stratégies élémentaires `Fail` et `Identity`. Les opérateurs CTL AX et EX correspondent respectivement aux combinateurs de traversée `All` et `One`. L'opérateur AG s'encode par une stratégie `TopDown`. L'encodage de l'opérateur EG est un peu plus complexe car selon la sémantique de SL,

$[\![true]\!]$ ≡ `Identity`
$[\![false]\!]$ ≡ `Fail`
$[\![p_1 \vee p_2]\!]$ ≡ `Choice(`$[\![p_1]\!]$`,`$[\![p_2]\!]$`)`
$[\![\neg p]\!]$ ≡ `Not(`$[\![p]\!]$`)`
$[\![EX(p)]\!]$ ≡ `One(`$[\![p]\!]$`)`
$[\![EG(p)]\!]$ ≡ μ`x.Sequence(`$[\![p]\!]$`,IfThenElse(One(Identity),One(x),Identity))`
$[\![EU(p_1,p_2)]\!]$ ≡ μ`x.Choice(`$[\![p_2]\!]$`,Sequence(`$[\![p_1]\!]$`,One(x)))`
$[\![AX(p)]\!]$ ≡ `All(`$[\![p]\!]$`)`
$[\![AG(p)]\!]$ ≡ μ`x.Sequence(`$[\![p]\!]$`,All(x))`
$[\![AU(p_1,p_2)]\!]$ ≡ μ`x.Choice(`$[\![p_2]\!]$`,Sequence(`$[\![p_1]\!]$`,All(x),One(Identity))`

FIG. 4.3: Définition de la fonction d'encodage $[\![\ ]\!]$ des formules CTL

la stratégie `One` échoue aux feuilles. On ne peut donc pas encoder EG par la stratégie μ`x. Sequence(`$[\![p]\!]$`,One(x))` qui échoue systématiquement. Pour gérer le cas des feuilles, on remplace `One(x)` par l'expression `IfThenElse(One(Identity),One(x),Identity)`. L'expression `One(Identity())` permet de tester si le nœud courant n'est pas une feuille. Dans ce cas, la stratégie se comporte comme `One(x)`. Dans le cas où le nœud courant est une feuille, cette expression se comporte comme `Identity`. L'encodage de l'opérateur AU utilise aussi la stratégie `One(Identity)` pour tester si le nœud courant n'est pas une feuille. En effet, tant que la deuxième proposition p_2 n'est pas vérifiée, on vérifie p_1 pour toutes les branches (stratégie récursive `All(x)`). Or le combinateur `All` se comporte comme l'identité aux feuilles. IL faut donc forcer l'échec (à l'aide de la stratégie `One(Identity)`) si p_2 n'a jamais été vérifié.

Concrètement, cet encodage est disponible dans la bibliothèque SL sous forme d'un ensemble de ancrages Tom. Seule la partie constructive de l'ancrage (construction `make`) nous intéresse car on obtient ainsi des *macros* de stratégies. L'évaluation de ces macros n'est réalisée que dans une expression *backquote*.

Exemple 14. L'ancrage de l'opérateur `EU` est défini comme suit :

```
%op Strategy EU(s1:Strategy, s2:Strategy) {
  make(s1,s2) {( `mu(MuVar("x"),Choice(s2,Sequence(s1,One(MuVar("x"))))))}
}
```

Lemme 1. *Soit t un terme, ϕ une formule CTL et ω une position de t :*

$$\langle \underline{[\![\phi]\!]}, t, \omega \rangle \rightarrow t \oplus \langle \underline{[\![\phi]\!]}, t, \omega \rangle \rightarrow \mathit{fail}$$

Démonstration. Par récurrence structurelle sur la formule ϕ.

Par définition, l'encodage des prédicats ne modifie pas le terme et la sémantique des opérateurs de stratégie utilisés dans la traduction des opérateurs temporels préserve aussi les termes. □

Lemme 2. *Soit t un terme, s une stratégie SL μ-close et ω une position de t :*

$$\forall S,\ \langle \underline{s}, t, \omega\rangle \rightarrow t' \ \Leftrightarrow\ \langle S\{\underline{s}\}, t, \omega\rangle \rightarrow t'$$

où S est un contexte.

Démonstration. Par définition de la sémantique des opérateurs SL. □

Théorème 2. *Soit t un terme, $\mathcal{M}$ le modèle correspondant, ϕ une formule CTL et ω une position de t :*

$$\mathcal{M}, \omega \models \phi \ \Leftrightarrow\ \langle \underline{[\![\phi]\!]}, t, \omega\rangle \rightarrow t$$

Démonstration. Par récurrence structurelle sur la formule ϕ.

On suppose tout d'abord que ce théorème est vérifié pour l'encodage des prédicats puisque c'est à l'utilisateur de s'assurer que les constructions `%strategy` définissant les prédicats vérifient ce théorème. Il reste maintenant à vérifier que l'encodage des opérateurs comme défini dans la figure 4.3 respecte le théorème.

Dans cette preuve, nous ne détaillerons que le cas de l'opérateur $AG(\phi)$. La preuve peut être réalisée de la même manière pour les autres opérateurs.

($\Rightarrow$)

On suppose que $\mathcal{M}, \omega \models AG(\phi)$ et on veut montrer que $\langle \underline{\mu x.Seq([\![\phi]\!], All(x))}, t, \omega\rangle \rightarrow t$. D'après la sémantique de AG, on sait que $\forall \omega' \in \mathcal{P}os(t)$ telle que ω est préfixe de ω', $\mathcal{M}, \omega' \models \phi$ (cette hypothèse est notée $DefAG$). Comme $[\![AG(\phi)]\!] = \mu x.Sequence([\![\phi]\!], All(x))$, on a comme hypothèse d'induction $\mathcal{M}, \omega \models \phi \ \Leftrightarrow\ \langle \underline{[\![\phi]\!]}, t, \omega\rangle \rightarrow t$ (notée $HypInd$).

D'après les règles de la sémantique de SL, on obtient l'arbre de dérivation suivant (on note n la valeur $ar(symb(t_{|\omega}))$ et Seq l'opérateur de stratégie $Sequence$) :

$$\dfrac{\dfrac{\dfrac{\text{HypInd + DefAG + Lemme 2}}{\langle \mu x(Seq(\underline{[\![\phi]\!]}, All(x))), t, \omega\rangle \rightarrow t} \quad \dfrac{\dfrac{\forall i \in [1,n]\ \Pi(\omega \centerdot i)}{\forall i \in [1,n]\ \langle \underline{\mu x(Seq([\![\phi]\!], All(x))}, t, \omega \centerdot i\rangle \rightarrow t}}{\forall i \in [1,n]\ \langle \mu x(Seq([\![\phi]\!], All(\underline{x})), t, \omega \centerdot i\rangle \rightarrow t}}{\langle \mu x(Seq([\![\phi]\!], \underline{All(x)}), t, \omega\rangle \rightarrow t}}{\langle \mu x(\underline{Seq([\![\phi]\!], All(x))}, t, \omega\rangle \rightarrow t}}{\langle \underline{\mu x(Seq([\![\phi]\!], All(x)))}, t, \omega\rangle \rightarrow t}$$

Pour construire $\Pi(\omega \centerdot i)$, on raisonne par récurrence sur la profondeur de $t_{|\omega}$. En effet, on peut réaliser la même réduction que celle pour ω jusqu'à atteindre les feuilles de $t_{|\omega}$. On obtient donc pour chaque position $\omega' \in \mathcal{P}os(t)$ telle que ω est préfixe de ω' les arbres de dérivation suivant (noté $\Pi(\omega')$) :

– si $t_{|\omega'}$ est une feuille :

$$\dfrac{\dfrac{\dfrac{\text{HypInd + DefAG + Lemme 2}}{\langle \mu x(Seq(\underline{[\![\phi]\!]}, All(x))), t, \omega'\rangle \rightarrow t} \quad \dfrac{}{\langle \mu x(Seq([\![\phi]\!], \underline{All(x)}), t, \omega'\rangle \rightarrow t}}{\langle \mu x(\underline{Seq([\![\phi]\!], All(x))}, t, \omega'\rangle \rightarrow t}}{\langle \underline{\mu x(Seq([\![\phi]\!], All(x)))}, t, \omega'\rangle \rightarrow t}$$

– si $t_{|\omega'}$ n'est pas une feuille :

$$\dfrac{\dfrac{\text{HypInd + DefAG + Lemme 2}}{\langle \mu x(Seq(\underline{[\![\phi]\!]}, All(x))), t, \omega' \rangle \to t} \quad \dfrac{\dfrac{\forall i \in [1,n]\ \Pi(\omega' \centerdot i)}{\forall i \in [1,n]\ \langle \underline{\mu x(Seq([\![\phi]\!], All(x))}, t, \omega' \centerdot i \rangle \to t}}{\forall i \in [1,n]\ \langle \mu x(Seq([\![\phi]\!], All(\underline{x})), t, \omega' \centerdot i \rangle \to t}}{\langle \mu x(Seq([\![\phi]\!], \underline{All(x)}), t, \omega' \rangle \to t}}{\dfrac{\langle \mu x(\underline{Seq([\![\phi]\!], All(x))}), t, \omega' \rangle \to t}{\langle \underline{\mu x(Seq([\![\phi]\!], All(x)))}, t, \omega' \rangle \to t}}$$

Où $n = ar(symb(t_{|\omega'}))$

($\Leftarrow$)

On suppose que $\langle \underline{\mu x.Seq([\![\phi]\!], All(x))}, t, \omega \rangle \to t$ et on veut montrer que $\mathcal{M}, \omega \models AG(\phi)$. D'après la sémantique de AG, il suffit de montrer que $\forall \omega' \in \mathcal{P}os(t)$ telle que ω est préfixe de ω', $\mathcal{M}, \omega' \models \phi$.

On considère donc l'arbre unique de dérivation de $\langle \underline{\mu x.Seq([\![\phi]\!], All(x))}, t, \omega \rangle \to t$. Par inversion des règles de la sémantique de SL, on a inévitablement la base suivante :

$$\dfrac{\dfrac{\dfrac{\Pi_1}{\langle \mu x(Seq(\underline{[\![\phi]\!]}, All(x))), t, \omega \rangle \to t'} \quad \dfrac{\Pi_2}{\langle \mu x(Seq([\![\phi]\!], \underline{All(x)}), t', \omega \rangle \to t}}{\langle \mu x(\underline{Seq([\![\phi]\!], All(x))}), t, \omega \rangle \to t}}{\langle \underline{\mu x(Seq([\![\phi]\!], All(x)))}, t, \omega \rangle \to t}$$

où $t, t' \in \mathcal{T}(\mathcal{F})$.

Nous allons maintenant prouver que $t = t'$. D'après le lemme 2, on peut déduire que $\langle \mu x(Seq(\underline{[\![\phi]\!]}, All(x))), t, \omega \rangle \to t'$ implique que $\langle \underline{[\![\phi]\!]}, t, \omega \rangle \to t'$. Or d'après le lemme 1, comme $[\![\phi]\!]$ est l'interprétation de la formule CTL ϕ, $\langle \underline{[\![\phi]\!]}, t, \omega \rangle \to t \ \vee\ \langle \underline{[\![\phi]\!]}, t, \omega \rangle \to \mathsf{fail}$ donc on a forcément $t = t'$.

D'après l'hypothèse de récurrence, on a donc prouvé que $\mathcal{M}, \omega \models \phi$. On peut donc poursuivre le même raisonnement sur le reste de l'arbre de dérivation et on prouve ainsi que $\forall \omega' \in \mathcal{P}os(t)$ tel que ω est préfixe de ω', $\mathcal{M}, \omega' \models \phi$.

□

Corollaire 1.

$$\mathcal{M}, \omega \not\models \phi \ \Leftrightarrow\ \langle \underline{[\![\phi]\!]}, t, \omega \rangle \to \mathit{fail}$$

Démonstration. Conséquence du théorème 2 et du lemme 1. □

Le lien entre les stratégies et la logique temporelle avait déjà été constaté dans [Kie01] où une logique de plus faibles pré-conditions fondée sur CTL avait permis de définir partiellement la sémantique du langage Stratego.

4.6.3 Inlining de variables

Supposons que nous voulions définir l'*inlining* de variables dans le langage introduit dans la figure 4.1. Si une variable n'est utilisée qu'une seule fois alors sa déclaration n'est pas nécessaire et son utilisation est remplacée par l'expression d'assignation. Pour vérifier les conditions nécessaires à cette optimisation, David Lacey [Lac03] a proposé dans sa thèse d'utiliser des formules CTL.

Définition CTL

Dans notre cas, pour chaque instruction `Declare(v,e,i)` d'un programme, nous devons vérifier la formule CTL suivante :

$$A(\neg \mathsf{Modified}(e))U(\mathsf{Used}(v) \wedge AX(AG\neg \mathsf{Used}(v)))$$

Où le prédicat $\mathsf{Modified}(e)$ vérifie si il existe une variable contenue dans `e` qui est modifiée au nœud courant et le prédicat `Used(v)` vérifie si `v` est utilisée au nœud courant.

La formule $\mathsf{Used}(v) \wedge AX(AG\neg \mathsf{Used}(v))$ vérifie que `v` est utilisée une seule fois. Finalement, la formule complète permet d'exprimer la propriété suivante : les variables de `e` ne sont pas modifiée jusqu'à atteindre un point du programme tel que `v` est utilisée au nœud et plus jamais dans le reste du sous-arbre.

Vérifier cette formule sur l'AST plutôt que sur le graphe de flot de contrôle (CFG) est ici correct car le langage est assez simple. Dans le cas général, il faudrait appliquer la formule temporelle sur le CFG.

Encodage en Tom

Le point délicat concerne la définition des prédicats. En effet, chaque prédicat est encodé par une instruction `%strategy` qui se comporte comme la stratégie identité si le prédicat est vérifié et échoue sinon.

```
%strategy Modified(vars:Set) extends Fail() {
  visit Instruction {
    i@Assign(name,e) -> {
      if(vars.contains(‘name)) {
        return ‘i;
      }
    }
  }
}
```

Nous rappelons que le symbole `@` permet de déclarer une variable dont la valeur est celle du sous-terme du motif qui suit ce symbole et auquel on a appliqué la substitution.

```
%strategy Used(varName:String) extends Fail() {
  visit Expression {
    v@Var(name) -> {
```

```
      if(varName.equals(‘name)) {
        return ‘v;
       }
     }
   }
}
```

Remarquons que ces deux stratégies élémentaires ont un comportement par défaut qui est l'échec et le filtrage nous permet de discriminer les cas de succès.

En utilisant les ancrages CTL de Tom, nous pouvons définir la stratégie `IsUsedOnce` :

```
%strategy IsUsedOnce() extends Fail() {
  visit Instruction {
    Declare(name,e,i) -> {
      Set vars = new HashSet();
      vars.add(‘name);
      vars.addAll(‘e.getVariables());
      return ‘AU(Not(Modified(vars)),
                And(Used(name),
                    AX(AG(Not(Used(name)))))).visit(‘i);
    }
  }
}
```

Cette stratégie filtre uniquement les déclarations de variables. En cas de déclaration, elle collecte les variables contenues dans `e` puis retourne l'évaluation de l'encodage de la formule temporelle. Dans tous les autres cas, il y a échec.

Finalement l'inlining dans un bloc d'instructions `i` peut être défini de la manière suivante :

```
‘TopDown(If(IsUsedOnce(),Inline(),Identity())).visit(i);
```

Où `Inline` est une stratégie qui remplace la variable par sa valeur dans le bloc d'instructions correspondant à sa portée.

Nous verrons dans le chapitre 6 comment cet encodage a été utilisé pour définir de manière élégante des propriétés de sécurité sur les arbres abstraits du Bytecode Java.

4.7 Apports techniques de la bibliothèque SL

En plus de la formalisation du langage SL, un des apports de cette thèse a été le développement de la bibliothèque.

4.7.1 Fonctionnement général

La bibliothèque SL offre deux interfaces principales `Visitable` et `Strategy`.

L'interface `Visitable` explicite la structure arborescente d'un objet. Cette interface contient des méthodes pour accéder et modifier les *enfants* (sous-arbres) de l'objet vu comme un nœud dans un arbre.

L'interface `Strategy` décrit les méthodes que doivent implémenter à la fois les combinateurs génériques de stratégies mais aussi les stratégies élémentaires comme `Identity()` ou la représentation de la construction `%strategy` du langage Tom. La méthode principale de cette interface est `Visitable visit(Visitable)` qui visite le sujet en maintenant l'environnement à chaque pas d'évaluation.

L'application d'une stratégie sur un terme (généralement appelé *sujet*) en utilisant la méthode `visit` retourne un objet de type `Visitable`. En cas d'échec, une exception Java de type `VisitFailure` est déclenchée. Afin de permettre l'introspection des stratégies, la bibliothèque SL propose des ancrages pour chacun des combinateurs et l'interface `Strategy` étend l'interface `Visitable` ce qui permet de visiter une stratégie à l'aide d'une autre stratégie.

L'environnement d'évaluation est représenté par une instance de la classe `Environment` qui représente à la fois la position courante (représentée par une liste d'entiers) et le sous-terme sur lequel la stratégie doit s'appliquer. Toute stratégie peut accéder à son environnement courant à l'aide de la méthode `getEnvironment()` de l'interface `Strategy`.

La bibliothèque offre une implémentation partielle de l'interface `Strategy` qui correspond à la classe abstraite `AbstractCombinator`. Cette classe permet de définir de nouveaux combinateurs génériques de stratégies. Une autre classe abstraite dénommée `AbstractBasicStrategy` implémente aussi partiellement l'interface `Strategy` et est utilisée comme base d'implémentation pour les stratégies élémentaires comme par exemple la classe interne générée pour l'instruction `%strategy`.

Comparaison avec l'architecture de JJTraveler

La Figure 4.4 présente le fonctionnement général de la bibliothèque JJTraveler. Les structures de données des utilisateurs doivent implémenter une interface `Visitable` qui permet d'expliciter la structure arborescente et les stratégies doivent implémenter l'interface `Visitor`. Afin d'assurer la préservation des types, une interface `HVisitor` (spécifique à un ensemble de types `H`) étend `Visitor`. Elle propose une méthode `T visitT(T v)` pour chaque type `T`. Une implémentation par défaut (classe `HFwd`) définit la méthode générique `visit` en appelant en fonction du type de l'argument la méthode `visitT` correspondante. Elle propose aussi une implémentation par défaut des méthodes typées. Chaque `visitT` appelle la stratégie par défaut donnée en argument du constructeur (champ `default` sur la Figure). Lorsque l'utilisateur veut appliquer un traitement particulier, il n'a plus qu'à étendre la classe `HFwd` et redéfinir les méthodes typées utiles.

Dans le cas d'une application Tom utilisant SL, on voit que la préservation des types `A` et `B` est assurée par la délégation aux méthodes `visitA` et `visitB`. Comme ce code est généré, il n'est pas nécessaire comme pour JJTraveler de passer par une interface `HVisitor` et par une classe abstraite `HFwd` afin de s'assurer que le typage est respecté. En effet, au moment de la compilation du code Tom, le typage même de l'instruction `%strategy`

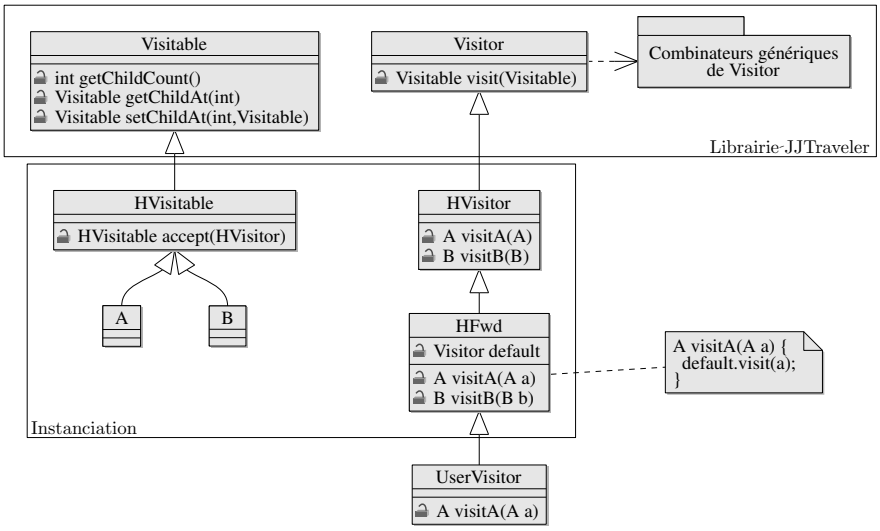


FIG. 4.4: Design-pattern Visiteur de la bibliothèque JJTraveler

assure que les types seront préservés par la stratégie élémentaire correspondante.

Comme la bibliothèque SL peut être utilisée indépendamment du langage Tom, un programmeur Java qui souhaite implémenter un traitement sur une structure arborescente a aussi la possibilité d'étendre la classe abstraite `AbstractBasicStrategy` et d'utiliser les combinateurs pour décrire la traversée. Cependant, comme dans JJTraveler, sa structure de données doit implémenter l'interface `Visitable`. Nous verrons dans la sous-section 4.5.3 comment la bibliothèque SL couplée au langage Tom permet d'éviter cette dépendance.

La Figure 4.5 montre comment étendre la bibliothèque SL pour définir de nouveaux combinateurs génériques ou des stratégies élémentaires. Un exemple de code généré pour une instruction `%strategy` est présenté. Chaque instruction `%strategy` est compilée en une classe statique interne étendant la classe abstraite `AbstractBasicStrategy`.

Cependant, si la bibliothèque SL est utilisée dans une application Java pure, il est possible de s'inspirer du design pattern de JJTraveler pour assurer la préservation des types. Dans ce cas, le programmeur devra implémenter une classe abstraite `HBasicStrategy` et une interface `HStrategy`, respectivement similaires à `HFwd` et `HVisitor`. Pour réutiliser au mieux SL, la classe `HBasicStrategy` implémentera l'interface `HStrategy` et étendra la classe `AbstractBasicStrategy`.

4.7.2 Implémentation de l'opérateur `Mu`

Nous avons vu dans la section 4.4 qu'une des principales différences entre JJTraveler et SL est la présence explicite dans SL des opérateurs `Mu` et `MuVar`.

Dans la bibliothèque JJTraveler, la brique correspondant à la stratégie `BottomUp` est définie récursivement par `BottomUp(v) = Sequence(All(BottomUp(v)),v)` et est im-

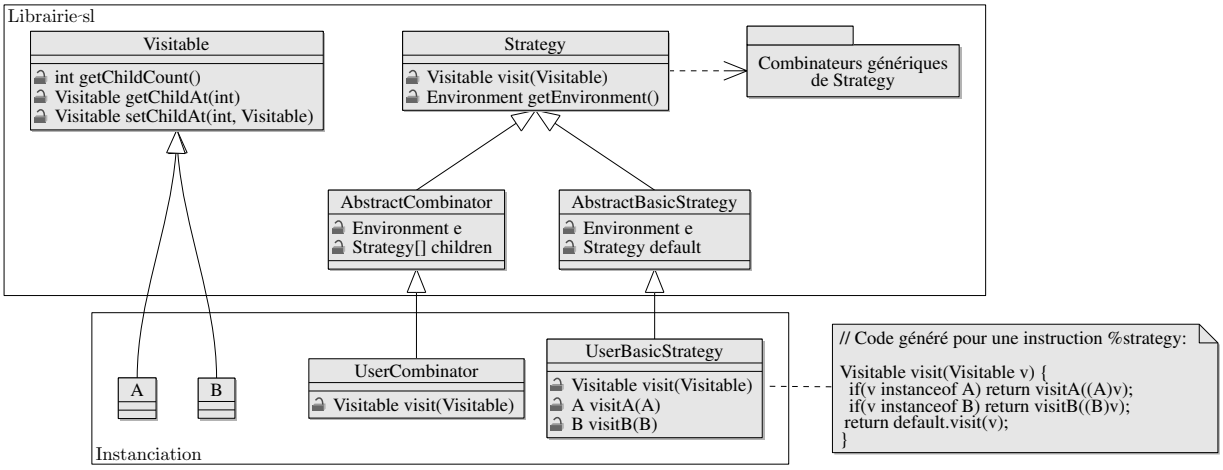


FIG. 4.5: Architecture de la bibliothèque SL

plémentée par la classe suivante :

```
package jjtraveler;

public class BottomUp extends Sequence {
  public BottomUp(Visitor v) {
    super(null,v);
    first = new All(this);
  }
}
```

La stratégie correspond donc à un graphe d'objets. La création de ce graphe est réalisée à l'aide des mécanismes d'héritage ainsi que des pointeurs `null` (qui sont ensuite initialisés après l'appel à `super`).

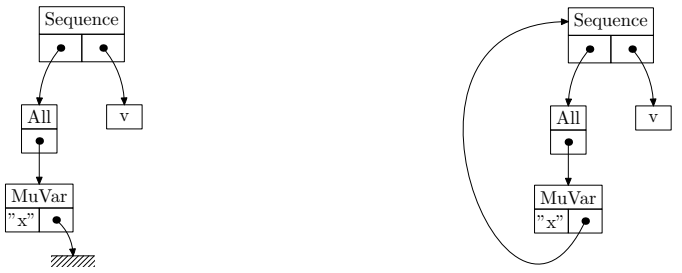
Dans le cas de SL, ce graphe est construit directement à partir de la définition de la stratégie sous forme d'arbre. Par exemple, pour `BottomUp`, l'arbre de stratégie initial est `Mu("x",Sequence(All(MuVar("x")),s))`. Afin d'obtenir le graphe d'objets `Strategy` correspondant, le combinateur `MuVar`, même s'il est d'arité 1 du point de vue de l'utilisateur (c'est à dire dans la définition de son ancrage), possède un pointeur vers l'objet de stratégie que cette variable représente.

Dans l'exemple de la stratégie `BottomUp`, l'objet `MuVar` a un attribut qui pointe vers l'objet `Sequence`. Cette opération réalisée dans le constructeur de la classe `Mu` est appelée *μ-expansion* et permet de construire à partir du terme de stratégies le graphe correspondant.

Le pointeur associé au nœud `MuVar` est initialement un pointeur `null`. La méthode statique `expand` de la classe `Mu` prend un arbre représentant une stratégie, lequel contient des nœuds `MuVar`, parcourt cet arbre à la recherche de `MuVar` et dont le pointeur n'est

pas encore initialisé. Elle fait alors pointer son nœud vers la tête de l'arbre de stratégie fils du dernier `Mu` portant le même nom, construisant ainsi le graphe.

La figure 4.6b représente le graphe obtenu après appel de la méthode `expand` sur la stratégie `BottomUp` représentée par l'arbre de la figure 4.6a.



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

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

FIG. 4.6: Graphes de stratégies avec le combinateur `MuVar`

Grâce au mécanisme d'ancrage, ce graphe peut tout de même toujours être *vu* comme l'arbre de départ même après *μ-expansion*.

4.7.3 Gestion de l'environnement

Gestion du contexte

En plus de gérer la position courante d'application (représentée par une liste d'entiers), la classe `Environment` met à jour une pile des contextes. Cette pile de contextes est nécessaire pour implémenter la réécriture de termes-graphes (comme cela sera présenté dans le chapitre 5) et le combinateur de traversée `Up` puisque le terme global doit toujours être accessible. Au lieu de simplement maintenir un sujet global, l'utilisation d'une pile permet d'accéder directement au sous-terme courant (tête de la pile) et au père de ce sous-terme (en dépilant). Comme la plupart des combinateurs de stratégie n'ont pas besoin du contexte, cette gestion permet de minimiser les déplacements dans le sujet global.

À l'initialisation, la pile contient le sujet global. Les opérateurs de stratégie agissent sur cette pile au travers des méthodes `down(i)` et `up()` qui permettent respectivement d'empiler le i^{e} fils de la tête de la pile pour indiquer une descente et de dépiler le i^{e} sous-terme en appliquant ensuite la substitution de ce sous-terme sur la nouvelle tête de la pile pour remonter au père. La méthode `setSubject` permet de modifier directement la tête de la pile, par exemple lors de l'application d'une règle.

La figure 4.7 présente l'évolution de l'environnement lors de l'application de la stratégie `BottomUp(Eval())` sur le terme `Plus(Mult(Cst(1),Cst(2)),Cst(3))` (l'arbre de dérivation avait été présenté dans la figure 4.2 de la section 4.2.4). À chaque étape d'évaluation, l'environnement est mis à jour (pile de contextes et position courante) et la sous-

stratégie qui s'évalue est indiquée dans le graphe de la stratégie globale `BottomUp(Eval())` par un nœud grisé (dans ce graphe, la stratégie `v` correspond à `Eval()`).

Gestion de l'échec

L'environnement permet de contrôler l'échec sans utiliser les exceptions du langage Java. Au lieu de propager des exceptions, l'environnement maintient une valeur entière d'état (les deux valeurs sont définies par les constantes statiques `SUCCESS` et `FAILURE`).

Mise à jour de l'environnement par les stratégies

Dans une stratégie (même composée comme `BottomUp`), chaque composant `Strategy` partage le même objet `Environment`. En effet, n'importe quelle stratégie a accès à son environnement courant (méthode `getEnvironment()` qui renvoie l'environnement partagé) et donc à la position courante dans le sujet (méthode `getPosition()` qui renvoie une copie de la position courante). À la création d'une stratégie composée, une instance de la classe `Environment` est initialisée et partagée par toutes les stratégies composantes.

La méthode `visit(Visitable)` de l'interface `Strategy` est définie dans la classe abstraite `AbstractCombinator` en fonction d'une méthode interne `_visit()` déclarée abstraite et implémentée par ses sous-classes. La méthode `protected int _visit()` permet de réaliser à la fois l'évaluation de la stratégie et la mise à jour de l'état de l'environnement.

```
public Visitable visit(Visitable any) throws VisitFailure {
  environment.setSubject(any);
  int status = _visit();
  if(status == Environment.SUCCESS) {
    return environment.getSubject();
  } else {
    throw new VisitFailure();
  }
}
```

Par exemple, le code de la méthode `_visit()` de la classe `Up` est défini de la manière suivante :

```
protected int _visit() {
  //réussit si en position racine
  if (environment.depth()==0) { return Environment.SUCCESS; }
  int index = environment.getPosition();
  environment.up();
  int status = visitors[ARG]._visit(); //applique la stratégie sur le père
  environment.down(index); //remet l'environnement dans le bon état
  return status;
```

(a) Première partie

FIG. 4.7: Exemple de mise à jour de l'environnement : application de la stratégie `BottomUp(Eval())` sur le terme `Plus(Mult(Cst(1),Cst(2)),Cst(3))`.

(b) Deuxième partie

FIG. 4.7: Exemple de mise à jour de l'environnement : application de la stratégie `BottomUp(Eval())` sur le terme `Plus(Mult(Cst(1),Cst(2)),Cst(3))`.

```
}
```

4.7.4 Introspection des structures de données par ancrage

Pour pouvoir proposer aux utilisateurs d'utiliser SL sans que leurs structures de données implémentent l'interface `Visitable`, nous avons mis en place un mécanisme de *proxy* au sein de la bibliothèque. Pour cela, l'interface `Strategy` a été étendue avec une nouvelle méthode :

```
Object visit(Object any, Introspector i)
```

Cette méthode est paramétrée par un objet qui n'implémente pas l'interface `Visitable` mais fournit un *introspecteur*, objet de type `Introspector` qui se comporte comme un *proxy* afin de rendre ces objets visitable. L'interface `Introspector` est composée de méthodes similaires à celle de `Visitable` mais paramétrées par l'objet sur lequel l'introspecteur doit agir :

```
Object setChildren(Object o, Object[] children)
Object[] getChildren(Object o)
Object setChildAt(Object o, int i, Object new_o)
Object getChildAt(Object o, int i)
int getChildCount(Object o)
```

Lorsque la bibliothèque SL est utilisée dans une application Java pure, les utilisateurs doivent fournir leur propre implémentation d'`Introspector` en fonction de leur structure de données. Lorsque la bibliothèque SL est utilisée conjointement à Tom, le compilateur de Tom génère automatiquement à partir des ancrages l'implémentation de l'interface `Introspector` et l'utilisateur peut directement profiter des stratégies sans avoir à modifier sa structure de donnée. De plus, dans le cas de grammaires, il est assez facile de générer automatiquement les ancrages.

4.8 Synthèse

Dans ce chapitre, nous avons formalisé et implémenté un langage de stratégies qui permet d'exprimer facilement des transformations et des analyses complexes d'arbres et de graphes dans un programme Java. Grâce à son interaction avec le langage Tom (au travers des ancrages), ce langage offre des constructions déclaratives pour définir des traversées génériques dans des structures de données typées Java sans modifier les classes définissant ces structures. Une autre originalité de ce langage est d'offrir de manière explicite l'environnement d'exécution, ce qui permet de connaître le contexte d'évaluation (en particulier la position dans le terme global) et ainsi de définir des parcours plus complexes comme ceux dans les graphes. Une autre des contributions de ce chapitre a été de montrer formellement la relation entre opérateurs de logique temporelle et combinateurs de stratégies. Cela permet d'encoder sous forme de stratégies la vérification

de formules temporelles et ainsi d'exprimer de manière plus concise [Lac03] des analyses non élémentaires de programmes.

L'ensemble de ces travaux a été intégré dans le langage Tom et est actuellement utilisé dans la plupart des applications développées en Tom. Par exemple, le compilateur Tom utilise de manière intensive le langage de stratégies et les constructions Tom pour définir des stratégies élémentaires à chaque phase de compilation. L'assistant à la preuve Lemuridae [BHK07] utilise ce langage de stratégies pour collecter de l'information dans les arbres de preuves ou encore encoder son langage de tactiques. Nous verrons dans le chapitre 6 des applications de ce langage à l'analyse et la transformation de programmes.

5 Algèbre de chemins pour la réécriture stratégique de termes-graphes

Contexte Nos travaux sur l'analyse de Bytecode et en particulier sur la représentation de graphes de flot nous ont conduit à réfléchir à une manipulation de ce type de structures dans le langage Tom, qui ne permet que de manipuler des arbres. L'idée était de pouvoir représenter et analyser des graphes de flot. Nous nous sommes donc intéressés aux formalismes de réécriture de graphes. Les deux approches principales sont l'approche catégorielle qui manipule des graphes au sens large et l'approche *réécriture de termes-graphes* qui se présente comme une extension de la réécriture de termes au partage de sous-termes et aux cycles. Le formalisme de réécriture de termes-graphes étant le plus proche de celui de la réécriture de termes, nous avons proposé d'intégrer de la manière la moins intrusive possible ce type de structures de données et le pas de réécriture associé dans le langage Tom.

Contributions Pour réaliser cette extension, la première contribution a été de formaliser une représentation des termes-graphes par des termes et une simulation de la réécriture de termes-graphes. Nous avons ainsi proposé d'utiliser la notion de chemin pour représenter les cycles et les sous-termes partagés. Le deuxième apport a été la définition d'un algorithme original permettant d'exprimer le pas de réécriture de termes-graphes par des opérations de redirection de pointeurs. On obtient ainsi une simulation de la réécriture de termes-graphes dans un langages à base de termes. L'ensemble de ces travaux a été implémenté et intégré dans le langage Tom et profite ainsi à la fois de l'environnement Java mais aussi de la bibliothèque de stratégies présentée dans le chapitre précédent. Une version préliminaire a été publiée dans [BB07] en 2007. Le formalisme tel qu'il est présenté dans ce chapitre a été publié dans [BM08].

Plan du chapitre Nous présentons tout d'abord dans la section 4.1 une étude des formalismes et des langages dédiés à la transformation de graphes. Dans ce contexte, la section 5.2 définit précisément notre problématique : comment étendre aux termes-graphes les langages dédiés à la réécriture de termes. Dans la section 5.3, nous proposons une solution non-intrusive fondée sur la notion de chemin qui peut être vue comme une généralisation du concept de position. La section 5.4 décrit comment représenter les termes-graphes par des termes contenant des chemins : les termes référencés. Les chemins sont interprétés comme des pointeurs. La section 5.5 présente un algorithme permettant de simuler la réécriture de termes-graphes à partir de ce type de structures. Enfin, dans la section 5.6, nous montrons comment tous ces concepts ont été intégrés au langage Tom.

5.1 Travaux reliés à la transformation de graphes

5.1.1 Les formalismes théoriques

La transformation de graphes a donné lieu à différents formalismes théoriques. Les premiers travaux sur la transformation des graphes datent des années 60 et sont issus de la généralisation de la théorie des langages aux structures cycliques (appelée *grammaires de graphes*). La transformation de ce type de structures a été décrite en terme de constructions sur la catégorie des graphes et sont à la base de la plupart des langages de transformation de graphes. Une autre approche de transformation est née de la généralisation de la réécriture de termes aux termes-graphes, la motivation principale de ces travaux étant l'implémentation efficace des langages fonctionnels.

Par rapport à la transformation d'arbres, les principales différences sont :

- la reconnaissance d'un sous-graphe qui est réalisée par bisimulation fonctionnelle (plusieurs nœuds du sujet peuvent par exemple être identifiés par un seul dans le motif et donc le radical peut être plus *déplié* que le motif),
- le remplacement par un nouveau sous-graphe qui nécessite de définir formellement comment le contexte se connecte avec ce nouveau sous-graphe. En effet, contrairement aux arbres, le contexte peut contenir des références vers des sous-termes du réduit. L'application d'une règle doit donc prendre en compte la maintenance de ces références.

Grammaires de graphes

Les grammaires de graphes ont été introduites dans les années 60 comme une généralisation de la théorie des langages. Les domaines d'applications étaient principalement la reconnaissance de motifs, la construction de compilateurs et la spécification de types de données. Dans ce formalisme, la transformation de graphes consiste au remplacement de nœuds ou d'arêtes. Ces transformations sont décrites soit sous la forme de transducteurs d'arbres, soit par des formules monadiques du second-ordre, soit par des grammaires de remplacement d'hyper-arêtes (notées souvent par grammaires HR). L'approche la plus utilisée est le remplacement d'hyper-arête (*hyper-edge replacement*). Un hyper-graphe est une généralisation des graphes où une arête (appelée hyper-arête) peut être connectée à un nombre quelconque de nœuds (on peut voir ces arêtes comme des tentacules). Les règles de la grammaires décrivent comment une hyper-arête peut être remplacée par un hyper-graphe ayant le même nombre de nœuds extérieurs. Ainsi, il est facile de reconnecter le nouveau sous-graphe au contexte. La transformation consiste à enlever l'hyper-arête, à ajouter le nouveau sous-graphe et à le reconnecter au contexte en branchant le i^e nœud extérieur du nouveau sous-graphe avec le i^e nœud d'attachement. La connectivité est donc assurée directement par le type des règles considérées. Cette approche est décrite en détail dans [DKH97].

Approche catégorielle

Une catégorie est une collection d'objets notés $O_0, \ldots, O_n$ et une collection de flèches appelées morphismes et notées $f, g, l, \ldots$ respectant un ensemble défini d'axiomes. De plus, il est possible de composer les morphismes à l'aide d'une opération partielle appelée *composition* et notée _;_. Pour une définition formelle des catégories, le lecteur peut se référer à [AL91]. Dans le cas des graphes, la catégorie considérée a comme collection d'objets l'ensemble des graphes et comme collection de flèches l'ensemble des homomorphismes de graphes. Le principal intérêt de l'approche catégorielle est que les résultats généraux sur les catégories peuvent être réutilisés afin de prouver des propriétés sur les transformations de graphes.

Définition 40 (Graphes)**.** *Un graphe est défini par un triplet (V, E, ϕ) où ϕ est une fonction de E vers $V \times V$. V correspond à l'ensemble des nœuds et E à l'ensemble des arêtes. Les nœuds (respectivement les arêtes) sont parfois étiquetés par un ensemble L. Cette étiquetage est défini par une fonction $l : V \to L$ (respectivement, $E \to L$). On note $\mathcal{G}$ l'ensemble des graphes.*

Définition 41 (Homomorphisme de graphes)**.** *Soient deux graphes $G = (V, E, \phi)$ et $G' = (V', E', \phi')$, et l (respectivement l') la fonction d'étiquetage par L des nœuds de G (respectivement G'). Un homomorphisme de graphes est une fonction f de $V \to V'$ telle que :*

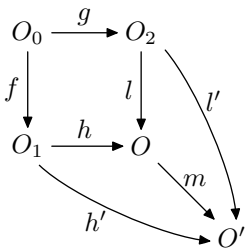
- *pour tout nœud $v \in V$, $l(v) = l'(f(v))$*
- *pour toute arête $e \in E$, on note $(v_1, v_2) = \phi(e)$, il existe une arête $e' \in E'$ telle que $\phi'(e') = (f(v_1), f(v_2))$.*

Dans cette catégorie, l'expression du rattachement du nouveau sous-graphe au contexte (appelé communément *graph gluing*) s'exprime comme une construction *pushout* sur la catégorie.

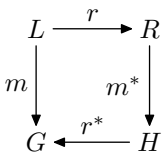
Définition 42 (Pushout)**.** *Soit une catégorie $\mathcal{C}$ et un couple de morphismes $f : O_0 \to O_1$ et $g : O_0 \to O_2$ de $\mathcal{C}$, on appelle* pushout *de (f, g) le triplet (O, h, l) tel que :*

- *$f; h = g; l$ (propriété de commutativité),*
- *pour tout objet O' et morphismes $h' : O_1 \to O'$ et $l' : O_2 \to O'$ tels que $f; h' = g; l'$, il existe un unique morphisme $m : O \to O'$ tel que $h; m = h'$ et $l; m = l'$ (propriété d'universalité).*

La construction *pushout* est illustrée par le diagramme présenté de la figure 5.1a. À partir de cette construction, il est possible de définir une dérivation directe de graphes ou production (dans le sens de la théorie des langages) soit comme un simple *pushout* [Löw93] (noté SPO) ou comme un double *pushout* [SEP73] (noté DPO). Nous allons présenter brièvement l'approche SPO puisque elle est à la base de la plupart des langages de transformations de graphes. Pour une comparaison précise entre les deux approches, on peut se référer à [Roz97].



(a) Diagramme du *Pushout*



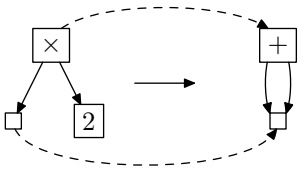
(b) Dérivation SPO

FIG. 5.1: Approche SPO de transformation de graphes

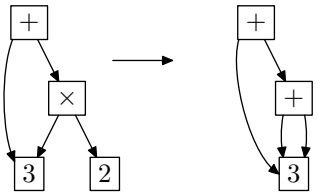
Définition 43 (Production, dérivation)**.** *Une production $p : L \to R$ est un morphisme partiel injectif appelé* morphisme de production *et noté r. Les graphes L et R correspondent respectivement à la partie gauche et droite de la règle. Une dérivation directe de G à H existe si et seulement s'il existe un morphisme m total de L vers G (correspondant à l'opération de filtrage) et une construction* pushout *pour le couple (r, m).*

Les conditions de dérivation sont vérifiées si le diagramme présenté dans la figure 5.1b peut être construit. Informellement, le morphisme partiel entre L et R permet de mettre en correspondance les nœuds de L avec les nœuds de R. Les nœuds pour lesquels le morphisme n'est pas défini seront donc supprimés par l'application de la règle et les nœuds de R n'ayant pas de pré-image seront ajoutés. Le contexte correspond donc aux nœuds préservés par le morphisme et les arêtes dont ils sont la cible peuvent être conservées. Le graphe final obtenu H n'a donc pas besoin d'une phase supplémentaire pour être nettoyé.

Exemple 15. Définissons une règle qui transforme la multiplication par deux en une addition avec partage :



Les flèches en pointillé représentent r le morphisme partiel entre L et R. Le nœud étiqueté par 2 est donc supprimé et il n'y pas de nouveau nœud créé. Les autres nœuds conservent donc leurs arêtes entrantes. Si on applique cette règle sur le graphe suivant, on obtient :



Réécriture de termes-graphes

La réécriture de termes-graphes propose une extension de la réécriture de termes permettant de gérer les sous-termes partagés ainsi que les cycles. La première définition a été introduite en 1987 par Barendregt [BEG+87] et ne gérait pas les cycles. En effet, ce formalisme avait été d'abord introduit pour réaliser des implémentations plus efficaces des langages fonctionnels en partageant les calculs [HP91]. Le concept des termes-graphes a ensuite très vite été étendu aux cycles afin de représenter des termes infinis réguliers comme les termes rationnels [CD97].

Par rapport à la réécriture de termes, les principales difficultés introduites par les termes-graphes sont :
- la nécessité de vérifier l'existence d'une bisimulation pendant le filtrage,
- la gestion du contexte pendant la substitution (contrairement à la réécriture de termes, il est nécessaire de nettoyer le sujet global après réécriture).

Il existe de nombreuses définitions de la réécriture de termes-graphes [BEG+87, AK96, Plu99]. Dans ce chapitre, nous avons choisi de nous baser sur la définition équationnelle introduite par Ariola dans [AK96].

Définition 44 (Systèmes d'équations récursives)**.** *Étant donné un ensemble $\mathcal{F}$ de symboles fonctionnels et un ensemble $\mathcal{X}$ de variables, un système d'équations récursives est de la forme $\{\alpha_1 \mid \alpha_1 = t_1, \ldots, \alpha_n = t_n\}$. Les conditions à vérifier sont que pour tous $i, j \in [1, n]$:*
- *$\alpha_i \in \mathcal{X}$,*
- *$i \neq j \Rightarrow \alpha_i \neq \alpha_j$,*
- *$t_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ est de la forme $f(\beta_1, \ldots, \beta_m)$ avec $f \in \mathcal{F}$ et $\forall k \in [1, m]$ $\beta_k \in \mathcal{X}$,*
- *α_i doit être atteignable depuis α_1.*

On note l'ensemble des systèmes d'équation $\mathcal{G}(\mathcal{F}, \mathcal{X})$

Soit un système d'équations récursives L, la racine est noté $\texttt{root}(L)$ et correspond généralement à la première variable α_1. L'ensemble des équations est noté $\texttt{set}(L)$. Une variable α est dite *liée* si elle apparaît en partie gauche d'une équation. Sinon, α est considérée *libre*. Les systèmes d'équations récursives sont considérés modulo renommage des variables. Dans la définition 44, les systèmes d'équations récursives sont présentés sous *forme aplatie* (c'est à dire que t_i est de la forme $f(\beta_1, \ldots, \beta_m)$). Cela assure l'unicité de la représentation d'un terme-graphe (modulo renommage des variables)

Un exemple de terme-graphe équationnel est présenté dans la figure 5.2. Ce terme-graphe cyclique correspond au système d'équations récursives $\{\alpha \mid \alpha = f(\beta, \gamma), \beta = g(\alpha), \gamma = f(\beta, \alpha)\}$. Il contient du partage de sous-termes (appelé communément *partage horizontal*) : α et γ partagent le même sous-terme β ; ainsi qu'un cycle (appelé *partage vertical*) : α est à la fois sous-terme de β et de γ.

Dans cette définition, les nœuds du terme-graphe sont identifiés à l'aide des variables. On doit donc redéfinir la notion de substitution :

Définition 45 (Substitution de variables)**.** *Une substitution est une fonction de $\mathcal{X}$ vers $\mathcal{X}$, notée lorsque son domaine est fini $\sigma = \{x_1 \mapsto y_1, \ldots, x_k \mapsto y_k\}$. Cette fonction*

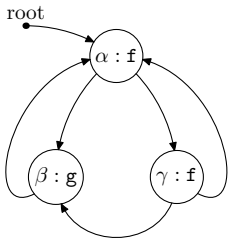


FIG. 5.2: Exemple de terme-graphe sous forme équationnelle

s'étend de manière unique en un endomorphisme $\sigma' : \mathcal{G}(\mathcal{F}, \mathcal{X}) \to \mathcal{G}(\mathcal{F}, \mathcal{X})$ *qui est défini inductivement par :*

$\sigma'(\{x_1 \mid x_1 = t_1, \ldots, x_n = t_n\}) = \{\sigma'(x_1) \mid \sigma'(x_1) = \sigma'(t_1), \ldots, \sigma'(x_n) = \sigma'(t_n)\}$ *tel que*

$$\forall z \in \mathcal{X}, \sigma'(z) = \begin{cases} \sigma(z) & \text{si } z \in \mathcal{D}om(\sigma) \\ z & \text{sinon.} \end{cases}$$

et $\sigma'(f(t_1, \ldots, t_n)) = f(\sigma'(t_1), \ldots, \sigma'(t_n))$.

Pour définir le filtrage de termes-graphes, nous introduisons la notion d'homomorphisme qui est un cas particulier de bisimulation. Un homorphisme est parfois appelé bisimulation fonctionnelle.

Définition 46 (Homomorphisme de termes-graphes)**.** *Un homomorphisme d'un terme-graphe* L_1 *vers un terme-graphe* L_2 *est une substitution* σ *telle que* $\texttt{set}(\sigma(L_1)) \subset \texttt{set}(L_2)$.

La substitution σ étant une fonction, cela signifie qu'un nœud de L_1 ne peut pas être associé à plusieurs nœuds de L_2. Autrement dit, un homomorphisme associe à un terme-graphe un autre terme-graphe ayant la même structure mais avec potentiellement plus de partage. En cas de cycle, L_1 peut donc être plus *déplié* que L_2 mais pas le contraire.

Définition 47 (Règle de réécriture)**.** *Une règle de réécriture* $L_1 \to L_2$ *est composée de deux systèmes d'équations récursives* L_1 *et* L_2 *ayant la même racine et tels que l'ensemble des variables libres de* L_2 *est inclus dans l'ensemble des variables libres de* L_1.

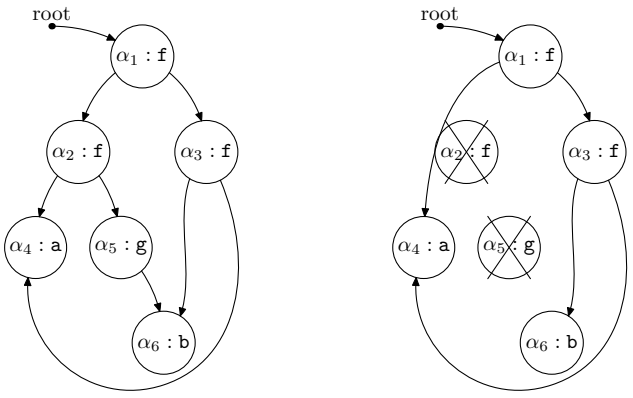
Définition 48 (Réécriture de termes-graphes équationnels)**.** *Un système d'équations récursives* L *se réécrit en* L' *par la règle* $L_1 \to L_2$ *s'il existe une substitution de variables* σ *et une équation* $\alpha = t$ *dans* L *telles que :*

- σ *est un homomorphisme de* L_1 *vers* L,
- $\alpha = \texttt{root}(\sigma(L_1))$,
- $\texttt{root}(L') = \texttt{root}(L)$,
- $\texttt{set}(L') = \texttt{set}(L) \setminus \{\alpha = t\} \cup \texttt{set}(\sigma'(L_2))$ *où* $\sigma'(L_2)$ *correspond à* $\sigma(L_2)$ *dans lequel chaque variable liée (sauf la racine) a été renommée par une variable fraîche.*

Pour obtenir de L' *un système bien défini, les équations correspondant à des variables inatteignables sont supprimées et les équations sont applaties. De plus, les cycles dégénérés (équations de la forme* $\alpha = \alpha$*) sont remplacés par* $\alpha = \bullet$ *où* $\bullet$ *est une constante spéciale appelée trou noir et permettant de gérer les cycles dégénérés. Les équations du type* $\alpha = \beta$ *sont supprimées après avoir renommé* α *en* β.

Remarquons que cette définition préserve celle des termes dans le cas de systèmes linéaires.

Exemple 16. Supposons que nous voulions appliquer la règle $\{\beta_1 \mid \beta_1 = f(\beta_2, \beta_3)\} \rightarrow \{\beta_1 \mid \beta_1 = \beta_2\}$ (qui correspond à $f(x, y) \rightarrow x$) à la position 1 du terme-graphe L suivant :



sujet initial L résultat L'

Le terme-graphe initial L est défini par l'ensemble d'équations $\{\alpha_1 \mid \alpha_1 = f(\alpha_2, \alpha_3), \alpha_2 = f(\alpha_4, \alpha_5), \alpha_3 = f(\alpha_6, \alpha_4),\ \alpha_4 = a,\ \alpha_5 = g(\alpha_6), \alpha_6 = b\}$. En appliquant la règle à la position 1 (i.e. sur α_2), nous avons $\sigma = \{\beta_1 \mapsto \alpha_2, \beta_2 \mapsto \alpha_4, \beta_3 \mapsto \alpha_5\}$ où α_2 est la variable choisie (correspond à la variable α de la définition). $\sigma(L_2) = \{\alpha_2 = \alpha_4\}$ (dans ce cas, le renommage avec des variables fraîches n'est pas nécessaire) et nous obtenons $L' = L \setminus \{\alpha_2 = f(\alpha_4, \alpha_5)\} \cup \sigma(L_2)$ comme résultat. Cela correspond au système $\{\alpha_1 \mid \alpha_1 = f(\alpha_4, \alpha_3), \alpha_3 = f(\alpha_6, \alpha_4), \alpha_4 = a, \alpha_6 = b\}$.

Réécriture de structures de données

Même si les structures de termes-graphes sont assez expressives, la réécriture de termes-graphes ne permet pas d'encoder de manière naturelle certaines transformations de structures de données avec pointeurs. En effet, on souhaite souvent pouvoir après filtrage modifier plusieurs sous-termes à la fois. Rachid Echahed et Nicolas Peltier ont proposé une définition de la réécriture de termes-graphes plus opérationnelle où la partie droite d'une règle correspond à une séquence d'actions [EP06]. Ce formalisme se fonde sur une représentation des termes-graphes où des étiquettes permettent d'identifier un nœud de manière unique. Il existe trois types d'actions : la définition d'un nœud ($\alpha : f(\alpha_1, \ldots, \alpha_n)$ où $\alpha, \alpha_1, \ldots \alpha_n$ sont des étiquettes et f un symbole d'arité n) qui permet de redéfinir un sous-terme, la redirection locale ($\alpha \gg_i \beta$) qui permet de rediriger la i$^{\text{e}}$ arête sortante du nœud α vers β, la redirection globale ($\alpha \gg \beta$) qui redirige toutes les arêtes entrante de α vers β.

Exemple 17. On définit la fonction *rev* qui inverse une liste (définie par les construc-

teurs *cons* et *nil*) par les règles suivantes :

$$
\begin{array}{lcl}
o : rev(p : \bullet) & \rightarrow & o : rev'(p, q : nil) \\
o : rev'(p_1 : cons(n : \bullet, q : nil), p_2 : \bullet) & \rightarrow & p_1 \gg_2 p_2; o \gg p_1 \\
o : rev'(p_1 : cons(n : \bullet, p_2 : cons(m : \bullet, p_3 : \bullet)), p_4 : \bullet) & \rightarrow & p_1 \gg_2 p_4; o \gg_1 p_2; o \gg_2 p_1
\end{array}
$$

Le symbole $\bullet$ permet de ne pas spécifier le contenu d'un nœud et on peut donc les voir comme des variables anonymes. La fonction rev' est une fonction intermédiaire qui a deux arguments. Le premier correspond au reste de la liste à inverser et le deuxième à la liste intermédiaire déjà inversée. La première règle du système initialise la valeur de ces deux arguments (le premier correspond à la liste de départ et le second à *nil*). La seconde règle est le cas terminal lorsqu'il ne reste plus qu'un seul élément à inverser. Dans ce cas, ce dernier élément est simplement concaténé au deuxième argument (action $p_1 \gg_2 p_2$) et cette nouvelle liste est retournée par la règle (action $o \gg p_1$). Enfin, la troisième règle du système réalise réellement l'inversion d'un élément en le concaténant au deuxième argument et en l'enlevant du reste de la liste.

5.1.2 Les outils et langages

Extension pour les langages à base de règles

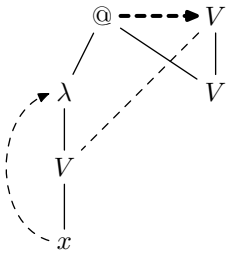
Parmi les langages à base de règles présentés dans la Section 4.1.1, les langages comme ELAN ou Maude qui implémentent le filtrage AC sont souvent utilisés comme base pour prototyper des transformations de graphes. Dans ce cas, la structure de données utilisée est constituée de deux ensembles (un pour les nœuds et un pour les arêtes). Le principal inconvénient de cette approche est que le nettoyage des nœuds et arêtes déconnectés n'est pas automatique et la structure de graphes n'est pas explicite (on ne peut par exemple utiliser le langage de stratégies pour parcourir ces structures). L'utilisateur doit donc maintenir lui-même la connectivité du graphe.

Seul le langage Stratego a proposé un prototype d'extension pour gérer les structures de termes avec pointeurs [KV06]. Le langage des termes a été étendu pour pouvoir gérer une notion de références qui permet d'encoder un pointeur vers n'importe quel sous-terme. Le langage de stratégies a aussi été étendu pour traiter les structures cycliques. Deux nouveaux opérateurs de traversée `wall` et `wone` permettent de contrôler le déréférencement.

D'un point de vue formel, cette solution n'encode pas directement la réécriture de termes-graphes car c'est à l'utilisateur de traiter les problèmes de nœuds et d'arêtes non connectés au reste du terme. Au niveau de l'implémentation, l'intégration de ce type de structures a été réalisée de manière *ad hoc*. Par exemple l'utilisation des références nécessite de maintenir une table d'information globale au programme. Enfin, le partage maximal des termes n'est plus assuré. Le formalisme que nous proposons dans ce chapitre permet de dépasser toutes ces limites.

Outils génériques de transformation de graphes

L'outil HOPS [Kah99] est un langage graphique dédié à la réécriture de termes-graphes typés. En plus d'implémenter la réécriture de termes-graphes, une des principales contributions de ce langage est la réécriture de termes-graphes de second ordre. On peut ainsi définir la β-réduction du lambda-calcul par la règle suivante :



Une règle est donc représentée par un terme-graphe. Les racines des parties gauches et droites de la règles sont liées par une flèche en pointillé. Les variables `V` correspondent à des meta-variables. Même si deux meta-variables portent le même nom, elles ne sont pas forcément identiques. Pour indiquer explicitement que deux meta-variables ayant la même arité sont identiques, l'utilisateur utilise une arête en pointillé entre les deux nœuds. Les variables sont explicitement liées en utilisant les cycles des termes-graphes et il n'y a donc pas de problème d'alpha-conversion dans ce type de définitions.

La plupart des autres outils de transformation de graphes sont fondés sur une structure de donnée plus générale que les termes-graphes : le *graphe orienté*. Ce graphe est souvent attribué par des objets ou des expressions et les transformations suivent l'approche catégorielle. Ces outils se présentent pour la plupart sous la forme d'un langage dédié qui est souvent graphique.

Le système PROGRES [SWZ95] (*PROgrammed Graph REwriting Systems*) est l'un des premiers systèmes graphiques de transformation de graphes. Les spécifications exécutables sont fondées sur les grammaires de graphes attribués et peuvent être traduites vers le langage C. La sémantique des transformations suit l'approche SPO. PROGRES est principalement utilisé pour prototyper des processus et des outils de *reengineering*.

AGG [Tae99] (*Attributed Graph Grammar*) est un langage plus récent. À l'instar de PROGRES, il propose un système visuel permettant de définir des règles de transformation de graphes mais permet aussi des définitions textuelles. L'application des règles est aussi réalisée suivant l'approche catégorielle SPO. Une des originalités de AGG par rapport à PROGRES est d'associer aux règles des attributs représentés par des expressions Java. Ces attributs sont évalués durant l'application des règles. Une autre particularité d'AGG est d'autoriser les conditions négatives en partie gauche des règles. Bien que généraliste, le langage AGG est principalement utilisé dans le cadre de la transformation de modèles.

Le système GrGen.NET [GBG+06] est aussi fondé sur l'approche SPO et constitue l'implémentation actuelle la plus efficace. Un DSL permet de définir de manière déclarative une structure de graphes et des règles associées. La spécification est ensuite traduite en C#. Le langage permet de définir des types, de l'héritage multiple (pour les nœuds

Outil	Structure	Sémantique	Mode	Langage	Notation	Contrôle
PROGRES	GO+schéma	opérationnel	compil/interp	C	visuelle/textuelle	transactions
AGG	GO typé	SPO/NAC	interprété	Java	visuelle	méthodes Java
GRGEN	GO typé	SPO	compilé	C#	textuelle	XGRS

FIG. 5.3: Caractéristiques des principaux langages de transformation de graphes

et les arêtes) ainsi que des attributs. À partir de cette spécification, le système génère un programme `.NET`.

Une des originalités de GrGen.NET est de proposer un langage d'expression (appelé XGRS pour *Extended Graph Rule Sequences*) pour contrôler l'application des règles. Par exemple, la règle `AddRedundancy` peut exécuter plusieurs règles à la suite.

```
rule AddRedundancy {
s: SourceNode; t: DestinationNode;
   modify {
     exec { (x) = DuplicateNode(s) & ConnectNeighbours(s, x)* }
     exec { [DuplicateCriticalEdge] }
     exec { MaxCapacityIncidentEdge(t)* }
   }
}
```

Dans ce langage, le symbole `&` correspond à la séquence de deux règles. Le symbole `*` correspond à la stratégie `Repeat` de SL. Les crochets indiquent que la règle doit être appliquée pour tous les réduits du graphe. On peut donc voir XGRS comme un langage de stratégies pour GrGen.NET.

Nous proposons dans la figure 5.3 un tableau récapitulatif des principales caractéristiques de chacun de ces langages. Dans cette figure, les initiales GO correspondent à *Graphe Orienté*, SPO à *Single Pushout* et NAC à *Negative Application Condition*.

Outils dédiés

Optimix [Ass00] est un générateur d'optimiseurs qui permet de prototyper très facilement des optimisations ou analyses de programmes. Le langage est fondé à la fois sur Datalog et la réécriture de graphes et génère du code C ou Java. Les règles peuvent soit simplement ajouter des arêtes (*Edge Addition Rewrite System* [Ass95]), soit définir une transformation plus complexe en composant des actions du type *supprimer/ajouter un nœud/une arête*. Il est toutefois beaucoup moins expressif que les langages présentés dans la figure 5.3. Les graphes ne sont pas attribués et l'utilisateur doit maintenir manuellement la cohérence de sa structure (en cas d'arête sans cible par exemple). Cependant, comme il est fondé sur Datalog, une de ses caractéristiques est d'assurer la terminaison des règles.

GROOVE [Ren03] est un outil plus récent, qui permet de prototyper de manière graphique des analyses et transformations de modèles (par exemple en UML) ou de

simuler l'exécution d'un programme (les graphes représentent par exemple l'état du système et les transformations encodent la sémantique opérationnelle). GROOVE a par exemple été utilisé pour simuler le fonctionnement de la machine virtuelle Java.

Il existe aussi de nombreux outils dédiés uniquement à la transformation et l'analyse de modèles et qui proposent donc des primitives de transformations de graphes. Les plus connus sont :

- VIATRA [CHM+02] qui est développé dans le cadre du projet Eclipse et permet de spécifier des transformations sur des diagrammes UML.
- Fujaba [FUJ] qui a la particularité de permettre de passer automatiquement du code Java au modèle UML et vice-versa. Les possibilités du langage de transformation sont très similaires à celles de PROGRES.
- ATL [BJ06] (Atlas Transformation Language) qui est un langage dédié à la transformation de modèles proposé dans le cadre d'un projet de l'université de Nantes dirigé par Jean bézivin. Contrairement aux autres outils graphiques, ATL propose un langage textuel hybride fondé sur des constructions impératives et déclaratives.

Pour plus de détails sur les outils et applications de la transformation de graphes, le lecteur peut se référer à [EEKR99].

5.2 Problématique

On souhaite étendre un langage à base de règles de manière à pouvoir représenter et transformer des termes-graphes. Cette extension doit être la moins intrusive possible. Plus précisément, cette extension doit réutiliser au maximum les constructions déjà existantes dans le langage et le même type de structures de données avec partage maximal. L'approche choisie est de représenter des termes-graphes par des termes, ce qui permet de conserver la même structure de données (termes avec partage maximal) et de s'intégrer facilement au langage. L'opération de réécriture doit respecter la sémantique des termes-graphes. Il faut donc proposer une nouvelle primitive de règles qui réutilise au maximum le langage.

Afin de conserver la structure de terme, la première solution envisagée était d'utiliser les positions. En effet, comme nous l'avons montré dans la sous-section 4.3.4, la réification des positions au niveau des stratégies permet d'identifier des sous-termes de manière unique même dans des représentations avec partage maximal. On pourrait donc tout à fait envisager de réifier les positions au niveau des termes, ce qui permettrait ainsi de représenter des pointeurs. Pour cela, il suffit d'étendre une signature algébrique par un ensemble infini de constantes représentant l'ensemble des listes d'entiers positifs.

On peut ainsi représenter des termes avec pointeurs et réutiliser le filtrage syntaxique :

Cependant, comme les positions sont absolues, dès que le filtrage n'est pas réalisé à la racine, les positions des pointeurs du filtre et du sujet ne correspondent plus. Par

exemple, le motif $f(x, 1)$ne filtre pas avec $g(f(a, 1 \,.\, 1))_{|1}$.

Afin que les pointeurs soient indépendants du terme global, une solution est de considérer des positions relatives, appelées *chemins*.

5.3 Algèbre de chemins

Un chemin défini comme une suite d'entiers relatifs permet d'accéder à un sous-terme depuis n'importe quel nœud (contrairement aux positions qui commencent systématiquement depuis la racine du terme).

Définition 49 (Chemin)**.** *L'ensemble $\mathcal{P}$ des chemins est l'ensemble des mots sur $\mathbb{Z}\backslash\{0\}$. Nous notons ϵ le mot vide, $p_1 \,.\, p_2$ la concaténation de deux mots p_1 et p_2 et $size(p)$ la longueur du mot p. Nous notons $\mathcal{P}^*$ l'ensemble $\mathcal{P} \setminus \{\epsilon\}$.*

Un chemin dans un terme t est représenté par une suite ω d'entiers naturels relatifs décrivant le chemin d'un nœud du terme jusqu'à un autre nœud. Un entier positif i est interprété comme un déplacement d'un nœud vers son i-ème fils. Un entier négatif $-i$ est interprété comme un déplacement du i-ème fils vers son nœud parent. Un chemin est relatif au nœud de départ (on dit qu'un chemin s'applique à un nœud) et donc la position destination ne peut être calculée qu'à partir du chemin et de la position source.

Remarquons que les positions sont un sous-ensemble des chemins (chemins composés uniquement d'entiers positifs et appliqués à la racine). Dans le reste de ce chapitre, les positions seront identifiées par les lettres grecques ω, δ.

Exemple 18. Dans le terme $f(a, b)$, le chemin -1 . 2 appliqué à la position 1 décrit comment atteindre le sous-terme b en partant de a. L'entier négatif -1 représente dans ce cas un déplacement de a vers f et l'entier positif 2, un déplacement de f vers b.

Grâce à cette définition des chemins, il est possible de calculer l'*inverse* d'un chemin sans avoir besoin d'information contextuelle. En effet, les déplacements vers le parent sont annotés par un entier négatif correspondant à l'indice du fils.

Définition 50 (Inverse)**.** *Étant donné un chemin p, l'inverse de p noté $\overline{p}$ est l'unique chemin vérifiant les équations suivantes :*

- $\overline{\epsilon} = \epsilon$ *et*
- $\overline{i \,.\, p} = \overline{p} \,.\, \text{-}i$.

Exemple 19. Dans le terme $f(a, b)$, l'inverse du chemin $\overline{\text{-1 . 2}}$ = -2 . 1 appliqué à la position 2 permet d'atteindre depuis le nœud b le sous-terme a.

Étant données deux positions ω_{src} et ω_{dest} (pour *source* et *destination*), notons que le chemin $\overline{\omega_{src}} \,.\, \omega_{dest}$ appliquée à la position ω_{src} connecte le sous-terme $t_{|\omega_{src}}$ à $t_{|\omega_{dest}}$.

Remarquons que plusieurs chemins peuvent représenter le même parcours. Pour obtenir des formes canoniques de terme-graphes, nous allons identifier ces chemins. Informellement, deux chemins sont équivalents, si quelque soit la position source ω, suivre

ces chemins depuis ω amènent à la même position destination. Par exemple, les chemins $1\,.\,2\,.\text{-}2$ et 1 sont équivalents.

Nous introduisons maintenant la notion de *forme canonique* comme le plus petit chemin d'une classe d'équivalence.

Définition 51 (Chemin canonique)**.** *La forme canonique d'un chemin $p \in \mathcal{P}$, notée $(\!|p|\!)$, est la forme normale de p par la règle $i\,.\,\text{-}i \to \epsilon$ si $i \in \mathbb{Z}^*$.*

Propriété 1. *La mise en forme canonique est confluente et terminante.*

Définition 52 (Équivalence de chemins)**.** *Deux chemins p_1 et p_2 sont équivalents si $(\!|p_1|\!) = (\!|p_2|\!)$.*

Notons que quelque soit la position source, le chemin $1\,.\,\text{-}2$ ne peut pas être considéré comme valide. En effet, il n'est pas possible de descendre d'un nœud vers son premier fils puis de remonter par son deuxième fils de manière consécutive. Ce type de chemin est intrinsèquement mal formé. Nous introduisons la constante $\perp$ pour dénoter les chemins mal formés.

Définition 53 (Chemin bien formé)**.** *Un chemin $p \in \mathcal{P}$ est bien formé si $(\!|p|\!) \not\to_R^* \perp$ avec R défini par la règle $p\,.\,i\,.\,\text{-}j\,.\,p' \to \perp$ si $i > 0$, $j > 0$ et $i \neq j$*

Exemple 20. $1\,.\,2\,.\,\text{-}2$ et $2\,.\,3\,.\,\text{-}3\,.\,\text{-}2\,.\,1$ sont des chemins bien formés mais $1\,.\,\text{-}2$ ne l'est pas.

Remarquons que les positions sont un sous-ensemble des chemins bien-formés étant donné qu'elles ne sont composées que d'entiers positifs (la condition est donc trivialement respectée).

Propriété 2. *Les chemins bien-formés ne sont pas clos par concaténation.*

Exemple 21. Les chemins 1 et -2 sont bien-formés mais leur concaténation $1\,.\,\text{-}2$ ne l'est pas.

5.4 Représentation des termes-graphes : les termes référencés

À partir de cette notion de chemins, nous allons introduire les termes référencés afin de simuler les termes-graphes. Un terme référencé est un terme dont certaines feuilles peuvent être des chemins.

Définition 54 (Termes référencés)**.** *Soient un ensemble $\mathcal{F}$ de symboles et un ensemble $\mathcal{X}$ de variables tels que $\mathcal{F}$, $\mathcal{X}$, $\mathcal{P}$ sont disjoints, nous notons par $\mathcal{T}_r(\mathcal{F}, \mathcal{X})$ l'ensemble des* termes référencés *$\mathcal{T}(\mathcal{F} \cup \mathcal{P}, \mathcal{X})$, où les éléments de $\mathcal{P}$ sont considérés comme des symboles d'arité 0.*

Exemple 22. Soit $\mathcal{F} = \{f, g, a\}$, a, g(-1), et $g(f(a, \text{-}2\,.\,1))$ sont des *termes référencés*, éléments de $\mathcal{T}_r(\mathcal{F}, \mathcal{X})$.

Notons que l'ensemble des termes est inclus dans l'ensemble des termes référencés (pour tout ensemble $\mathcal{F}$ et $\mathcal{X}$, $\mathcal{T}(\mathcal{F},\mathcal{X}) \subset \mathcal{T}_r(\mathcal{F},\mathcal{X})$).

Nous introduisons à présent l'opération de déréférencement $\texttt{deref}(t,\omega)$ qui retourne la position pointée par $t_{|\omega}$ si $symb(t_{|\omega}) \in \mathcal{P}$, et ω sinon.

Définition 55 (Déréférencement)**.** *Soit $t \in \mathcal{T}_r(\mathcal{F},\mathcal{X})$ et $\omega \in \mathcal{P}os(t)$, l'opération $\texttt{deref}(t,\omega)$ est définie comme suit :*

$$\texttt{deref}(t,\omega) = \begin{cases} (\!|\omega \texttt{.} symb(t_{|\omega})|\!) & si\ symb(t_{|\omega}) \in \mathcal{P} \\ \omega & sinon. \end{cases}$$

Exemple 23. $\texttt{deref}(g(\text{-}1),1) = \epsilon$, mais $\texttt{deref}(g(a),1) = 1$.

Notons que si $\omega \texttt{.} symb(t_{|\omega})$ est mal-formé, le résultat de $\texttt{deref}(t,\omega)$ n'a pas de sens. Il est donc nécessaire de caractériser la validité d'un terme référencé.

Définition 56 (Terme référencé valide)**.** *Un terme référencé $t \in \mathcal{T}_r(\mathcal{F},\mathcal{X})$ est* valide *si $\forall \omega \in \mathcal{P}os(t)$ tel que $symb(t_{|\omega}) \in \mathcal{P}$, les conditions suivantes sont respectées :*

- $\texttt{deref}(t,\omega) \in \mathcal{P}os(t)$,
- $symb(t_{|\texttt{deref}(t,\omega)}) \notin \mathcal{P}^*$.

Nous notons $\mathcal{T}_{vr}(\mathcal{F},\mathcal{X})$ l'ensemble des termes référencés valides et $\mathcal{T}_{vr}(\mathcal{F})$ l'ensemble des termes référencés valides clos.

La première condition assure que la valeur retournée par $\texttt{deref}(t,\omega)$ est une position de $\mathcal{P}os(t)$. Par exemple, $\texttt{deref}(g(-2),1)$ n'est pas un terme valide. La seconde condition interdit les pointeurs de pointeurs comme dans $f(-1\texttt{.}2,-2\texttt{.}1)$. Cette dernière condition n'est pas nécessaire pour simuler la réécriture de termes-graphes mais simplifie le formalisme. On pourrait tout à fait envisager de considérer de tels termes pour modéliser le tas dans un langage objet par exemple.

Les chemins vides ϵ sont autorisés dans les termes référencés valides afin de pouvoir gérer les cycles dégénérés. Ce type de cycles peut apparaître lorsqu'on applique une règle d'effondrement (*collapsing rule*), c'est à dire une règle de la forme $I(x) \to x$.

Dans le cadre de la réécriture de termes-graphes [AK96], les cycles dégénérés sont généralement remplacés par une constante spéciale appelée *trou noir* et notée $\bullet$. Dans notre cas, il n'est pas nécessaire d'introduire une nouvelle constante puisque le chemin vide correspond intuitivement à cette constante.

Exemple 24. Les termes ϵ, $g(\text{-}1)$, $f(\text{-}1\texttt{.}2,a)$ et $f(\text{-}1\texttt{.}2,\epsilon)$ sont valides, mais $g(3)$ et $f(\text{-}1\texttt{.}2,\text{-}2)$ ne le sont pas. Les termes réduits à un chemin non vide (1, -1.2, etc.) appartiennent à $\mathcal{T}_r(\mathcal{F},\mathcal{X})$ mais ne sont pas valides (i.e. $\notin \mathcal{T}_{vr}(\mathcal{F},\mathcal{X})$). Le terme $t = f(\text{-}1\texttt{.}1\texttt{.}\text{-}1, \text{-}2\texttt{.}3)$ est invalide car $\texttt{deref}(t,2) = (\!|2\texttt{.}\text{-}2\texttt{.}3|\!) = 3$ n'est pas dans $\mathcal{P}os(t) = \{\epsilon,1,2\}$.

5.5 Simulation de réécriture de termes-graphes

5.5.1 Classes d'équivalence entre termes référencés

Afin de simuler la réécriture de termes-graphes à l'aide des termes référencés, il est nécessaire de définir des classes d'équivalence. Un terme-graphe sera alors représenté par

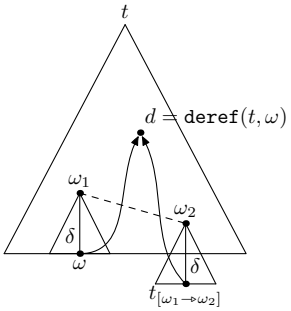


FIG. 5.4: Translation de sous-terme

un représentant canonique d'une classe d'équivalence de termes référencés valides.

Par exemple, $f(\text{-}1\,.\,2, a)$ et $f(a, \text{-}2\,.\,1)$ devraient être considérés comme équivalents. Ils correspondent en effet tous les deux au terme-graphe dont le symbole de tête est f et dont les deux sous-termes a sont partagés. Afin de définir l'équivalence, nous introduisons trois fonctions intermédiaires qui caractérisent la translation, l'expansion et le partage de sous-termes.

Translation de sous-termes

La *translation de sous-terme* permet de définir l'expansion de manière concise. Étant donné un terme t et deux positions ω_1, ω_2, comme illustré dans la figure 5.4, la translation dans t de ω_1 vers ω_2, notée $t_{[\omega_1 \twoheadrightarrow \omega_2]}$, retourne le sous-terme $t_{|\omega_1}$ où les chemins contenus dans $t_{|\omega_1}$ qui pointent à l'extérieur de $t_{|\omega_1}$ ont été mis à jour comme si $t_{|\omega_1}$ avait été déplacé à la position ω_2.

Définition 57 (Translation de sous-terme)**.** *Étant donné un terme $t \in \mathcal{T}_{vr}(\mathcal{F}, \mathcal{X})$ et deux positions ω_1, ω_2, la* translation de sous-terme $t_{[\omega_1 \twoheadrightarrow \omega_2]}$ *est définie telle que $\mathcal{P}os(t_{[\omega_1 \twoheadrightarrow \omega_2]}) = \mathcal{P}os(t_{|\omega_1})$ et $\forall \delta \in \mathcal{P}os(t_{|\omega_1})$:*

$$symb(t_{[\omega_1 \twoheadrightarrow \omega_2]|\delta}) = \begin{cases} (\!|\overline{\omega_2 \,.\, \delta} \,.\, \mathit{deref}(t, \omega_1 \,.\, \delta)|\!) & \text{si } symb(t_{|\omega_1 . \delta}) \in \mathcal{P}^* \\ & \text{et } \omega_1 \not\sqsubseteq \mathit{deref}(t, \omega_1 \,.\, \delta) \\ symb(t_{|\omega_1 . \delta}) & \text{sinon.} \end{cases}$$

La figure 5.4 illustre l'opération de translation. Étant donné un terme t et une position ω_1, supposons qu'à la position ω, le sous-terme $t_{|\omega}$ contienne une référence vers le sous-terme d (i.e. $d = \texttt{deref}(t, \omega)$). La translation de ω_1 vers ω_2 correspond à une mise à jour des pointeurs de $t_{|\omega_1}$ comme si $t_{|\omega_1}$ avait été déplacé en ω_2. Pour maintenir les pointeurs, les chemins contenus dans $t_{|\omega_1}$ sont recalculés en fonction de la nouvelle position source. Par exemple, $f(g(\text{-}1\,.\,\text{-}1), a)_{[1 \twoheadrightarrow 2]} = g(\text{-}1\,.\,\text{-}2)$.

Expansion

L'opération d'*expansion* notée `exp` consiste à dupliquer tous les sous-termes partagés. Étant donné un ensemble $\mathcal{F}$ de symboles fonctionnels, `exp` est une fonction de $\mathcal{T}_{vr}(\mathcal{F})$ vers $\mathcal{T}^{\infty}(\mathcal{F}\cup\{\epsilon\})$ où $\mathcal{T}^{\infty}(\mathcal{F}\cup\{\epsilon\})$ est l'ensemble des termes infinis [CG95] sur $\mathcal{F}\cup\{\epsilon\}$. Les termes infinis sont définis comme des fonctions partielles dont le domaine est l'ensemble infini des positions et le codomaine est $\mathcal{F}\cup\{\epsilon\}$. On note $\perp$ le terme indéfini représenté par la fonction $\emptyset \rightarrow \mathcal{F}\cup\{\epsilon\}$. Remarquons que les termes finis sont inclus dans l'ensemble des termes infinis.

Définition 58 (Expansion)**.** *Soit $t \in \mathcal{T}_{vr}(\mathcal{F})$, on définit la suite $\{t_i\}_{i\in\mathbb{N}}$ de termes dans $\mathcal{T}^{\infty}(\mathcal{F}\cup\mathcal{P})$ comme suit :*

- $t_0 = t$
- $t_{n+1} = t_n[t_{n[\mathtt{deref}(t_n,\omega)\rightarrow\omega]}]_{\omega}$
  où ω est la plus petite position dans $\mathcal{P}os(t_n)$ telle que $symb(t_{n|\omega}) \in \mathcal{P}^*$
  (on considère l'ordre suivant : $\omega < \omega'$ si $|\omega| < |\omega'| \ \vee \ (|\omega| = |\omega'| \wedge \omega <_{\mathcal{P}} \omega')$)

$\mathtt{exp}(t) \in \mathcal{T}^{\infty}(\mathcal{F})$ *est défini comme $\bigcup_{i=0}^{\infty} t'_i$ où t'_i correspond à t_i où chaque chemin $p \in \mathcal{P}^*$ a été remplacé par $\perp$.*

Nous rappelons qu'un terme infini est total si l'arité des symboles est respectée. Autrement dit, t est total si pour toute position ω, $symb(t_{|\omega}) = f \Rightarrow \forall 1 \leq i \leq ar(f),\ (\omega \mathbf{.} i) \in \mathcal{D}om(t)$.

Proposition 1. $\forall t \in \mathcal{T}_{vr}(\mathcal{F})$, $\mathtt{exp}(t)$ *est total.*

Démonstration. Par définition de la suite $\{t_i\}_{i\in\mathbb{N}}$, $\mathtt{exp}(t)$ est une fonction. Comme chaque chemin est remplacé par un sous-terme, $\mathtt{exp}(t)$ ne contient pas $\perp$ et comme t respecte les arités des symboles, $\mathtt{exp}(t)$ est total. □

Exemple 25. $\mathtt{exp}(f(\text{-}1\mathbf{.}2, a)) = \mathtt{exp}(f(a, \text{-}2\mathbf{.}1)) = f(a,a)$ et $\mathtt{exp}(g(\text{-}1)) = g(g(g(\ldots)))$. Notons qu'en cas de cycle, le terme expandé est un terme infini. Un exemple non-trivial est $f(g(\text{-}1\mathbf{.}\text{-}1\mathbf{.}2), h(\text{-}1\mathbf{.}\text{-}2\mathbf{.}1))$ où il y a une dépendance mutuelle entre les deux sous-termes. On obtient donc finalement le terme infini $f(g(h(g(h(\ldots)))), h(g(h(g(\ldots)))))$.

L'expansion exprime une première caractéristique des classes d'équivalence des termes valides référencés. En effet, deux termes équivalents ont la même expansion. Cependant cette condition n'est pas suffisante. En effet, on ne considère pas ici les termes-graphes modulo bissimilarité. On veut pouvoir par exemple distinguer les termes $f(a,a)$ et $f(a, \text{-}2\mathbf{.}1)$. Deux termes sont équivalents s'ils définissent le même partage de sous-termes.

Partage

Pour définir la condition sur le partage, nous introduisons une troisième relation notée `share` qui calcule l'ensemble des positions partagées.

Définition 59 (Partage)**.** *Soit $t \in \mathcal{T}_{vr}(\mathcal{F})$, on définit la suite $\mathtt{share}_i(t)$ comme suit :*

- $\mathit{share}_0(t) = \{\{\omega, \mathit{deref}(t,\omega)\} \mid \omega \in \mathcal{P}os(t) \text{ et } symb(t_{|\omega}) \in \mathcal{P}^*\}$
- $\mathit{share}_{n+1}(t) = \{\{\omega'\,.\,q, q'\} \mid \{\omega, \omega'\}, \{\omega\,.\,q, q'\} \in \mathit{share}_n(t)\}$

La fonction $\mathit{share}(t)$ *est définie comme* $\bigcup_{n=0}^{\infty} \mathit{share}_n(t)$.

Exemple 26. La fonction share calcule l'ensemble des couples de positions partagées. Par exemple, $\mathtt{share}(f(\text{-}1\,.\,2, a)) = \{\{1, 2\}\}$ et $\mathtt{share}(g(\text{-}1)) = \{\{1, \epsilon\}\}$. Un exemple non-trivial est $f(g(\text{-}1\,.\,\text{-}1\,.\,2), h(\text{-}1\,.\,\text{-}2\,.\,1))$. Au premier pas, $\mathtt{share}_0(t) = \{\{1, 2\,.\,1\}, \{2, 1\,.\,1\}\}$. Dans ce cas, il est nécessaire de clore la relation par les préfixes. On obtient finalement l'ensemble infini $\mathtt{share}(t) = \{\{1, 2\,.\,1\}, \{2, 1\,.\,1\}, \ldots, \{1\,.\,(1\,.\,1)^*, 2\,.\,1\,.\,(1\,.\,1)^*\}, \{2\,.\,(1\,.\,1)^*, 1\,.\,1\,.\,(1\,.\,1)^*\}\}$ (* représente la répétition de sous-liste). Comme pour l'expansion, le résultat est infini à cause de la dépendance mutuelle entre les deux sous-termes.

Équivalence

On peut maintenant définir l'équivalence de deux termes référencés valides.

Définition 60 (Equivalence)**.** *Deux termes référencés valides* t_1, t_2 *sont équivalents (noté* $t_1 \sim t_2$*) si* $\mathit{share}(t_1) = \mathit{share}(t_2)$ *et* $\mathit{exp}(t_1) = \mathit{exp}(t_2)$.

Il est facile de montrer que $\sim$ est une relation d'équivalence.

5.5.2 Termes référencés canoniques

Pour chaque classe d'équivalence, on définit une forme canonique à l'aide de l'ordre lexicographique sur les positions (noté $<_{\mathcal{P}}$).

Définition 61 (Termes référencés canoniques)**.** *Un terme référencé valide* $t \in \mathcal{T}_{vr}(\mathcal{F}, \mathcal{X})$ *est canonique si pour chaque position* $\omega \in \mathcal{P}os(t)$ *telle que* $symb(t_{|\omega}) \in \mathcal{P}^*$, $symb(t_{|\omega})$ *est un chemin canonique et* $\mathit{deref}(t, \omega) <_{\mathcal{P}} \omega$.
On note $\mathcal{T}_g(\mathcal{F}, \mathcal{X})$ *l'ensemble des termes référencés canoniques et* $\mathcal{T}_g(\mathcal{F})$ *l'ensemble des termes référencés canoniques clos.*

Exemple 27. Le terme $f(a, \text{-}2\,.\,1)$ est canonique tandis que $f(\text{-}1\,.\,2, a)$ ne l'est pas car $\mathtt{deref}(f(\text{-}1\,.\,2, a), 1) = 2$ (la position du sous-terme pointé a) est plus grande que 1.

Pour définir la fonction de normalisation qui retourne la forme canonique de chaque terme référencé valide, on introduit une fonction de permutation qui permute deux sous-termes et met à jour tous les chemins contenus dans le terme global afin de préserver le partage. La définition de la translation assure dans un premier temps la mise à jour des pointeurs du sous-terme déplacé. Pour obtenir un terme référencé valide, il reste à mettre à jour les chemins extérieurs qui vont de l'extérieur vers une position à l'intérieur du sous-terme.

Définition 62 (Permutation de sous-termes)**.** *Étant donné* $t \in \mathcal{T}_{vr}(\mathcal{F}, \mathcal{X})$ *et deux positions disjointes* ω_1, ω_2 *(*$\omega_1 \not\sqsubseteq \omega_2$ *et* $\omega_2 \not\sqsubseteq \omega_1$*), on considère* $u = t_{[\omega_1 \to \omega_2]}$, $v = t_{[\omega_2 \to \omega_1]}$,

$t' = t[v]_{\omega_1}[u]_{\omega_2}$. La permutation dans t de deux sous-termes à la position ω_1 et ω_2 (notée $t_{[\omega_1 \leftrightarrow \omega_2]}$), est définie telle que pour toute position $\omega \in \mathcal{P}os(t')$, nous avons :

$$symb(t_{[\omega_1 \leftrightarrow \omega_2]}{}_{|\omega}) = \begin{cases} (\!|\overline{\omega} \centerdot \omega_2 \centerdot \delta|\!) & si\ symb(t'_{|\omega}) \in \mathcal{P}^*,\ \omega \not\sqsupseteq \omega_1 \\ & \quad et\ \exists \delta\ s.t.\ \texttt{deref}(t', \omega) = \omega_1 \centerdot \delta \\ (\!|\overline{\omega} \centerdot \omega_1 \centerdot \delta|\!) & si\ symb(t'_{|\omega}) \in \mathcal{P}^*,\ \omega \not\sqsupseteq \omega_2 \\ & \quad et\ \exists \delta\ s.t.\ \texttt{deref}(t', \omega) = \omega_2 \centerdot \delta \\ symb(t'_{|\omega}) & sinon. \end{cases}$$

Exemple 28. $f(a,b)_{[1 \leftrightarrow 2]} = f(b,a)$, $f(g(\text{-}1 \centerdot \text{-}1 \centerdot 2), h(\text{-}1 \centerdot \text{-}2 \centerdot 1))_{[1 \leftrightarrow 2]} = f(h(\text{-}1 \centerdot \text{-}1 \centerdot 2), g(\text{-}1 \centerdot \text{-}2 \centerdot 1))$.

Un exemple plus complexe est la permutation de a et b dans $t = f(f(a,b), \text{-}2 \centerdot 1 \centerdot 2)$. La référence $\text{-}2 \centerdot 1 \centerdot 2$ doit être mise à jour car elle référence b. Le résultat obtenu est $f(f(b,a), \text{-}2 \centerdot 1 \centerdot 1)$.

La fonction de permutation préserve la notion de validité (définition 56). Sa complexité est linéaire en la taille du terme. En effet, dans le pire des cas, toutes les références doivent être mises à jour.

À partir de la fonction de permutation, il est facile d'exprimer la fonction de *normalisation* (notée $[\![\ ]\!]$) qui associe à chaque terme référencé valide sa forme canonique.

Définition 63 (Normalisation)**.** *La fonction $[\![\ ]\!] : \mathcal{T}_{vr}(\mathcal{F}, \mathcal{X}) \to \mathcal{T}_g(\mathcal{F}, \mathcal{X})$ est définie telle que $\forall t \in \mathcal{T}_{vr}(\mathcal{F}, \mathcal{X})$, $[\![t]\!]$ est la forme normale de t' (t où chaque chemin est en forme canonique) par la règle de réécriture conditionnelle suivante : $t' \to t'_{[\omega \leftrightarrow \texttt{deref}(t', \omega)]}$ si $\omega <_{\mathcal{P}} \texttt{deref}(t', \omega)$*

Proposition 2. *Le système formé par la règle conditionnelle $t' \to t'_{[\omega \leftrightarrow \texttt{deref}(t', \omega)]}$ si $\omega <_{\mathcal{P}} \texttt{deref}(t', \omega)$ est convergent.*

Démonstration. Nous devons prouver que le système termine et est confluent.

La propriété de terminaison est triviale à cause de l'ordre total sur les positions et de la définition des termes référencés valides qui ne peuvent pas contenir de pointeurs de pointeurs. Considérons le multi-ensemble $\{\texttt{deref}(t, \omega) \mid t_{|\omega} \in \mathcal{P}\}$ composé des positions référencées dans t et doté de l'extension multi-ensemble de l'ordre sur les positions (un multi-ensemble N est plus petit qu'un multi-ensemble M si N peut être obtenu en remplaçant un nombre fini d'éléments de M par des éléments plus petits).

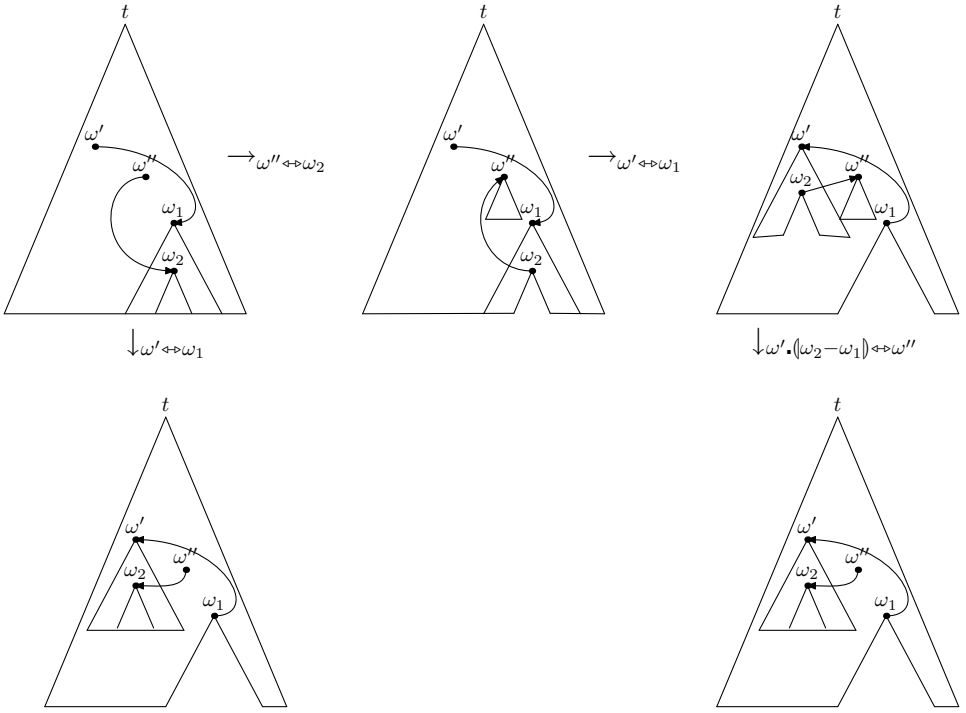
Ce multi-ensemble décroît strictement à chaque pas de réécriture. En effet, après avoir appliqué la règle à une position donnée $t_{|\omega} \notin \mathcal{P}$ et $t_{|\texttt{deref}(t,\omega)} \in \mathcal{P}$. On a donc $\texttt{deref}(t, \texttt{deref}(t, \omega)) = \omega$ et $\omega <_{\mathcal{P}} \texttt{deref}(t, \omega)$. Concernant les autres chemins mis à jour, $\texttt{deref}(t, \omega)$ est soit plus petit (pointeurs vers le sous-terme permuté ou un de ses sous-termes) ou reste inchangé (pointeurs à l'intérieur du sous-terme). Finalement, la cardinalité du multi-ensemble n'a pas évolué et chaque sous-terme remplacé a été substitué par un plus petit. La terminaison est donc assurée.

De plus, nous montrons que pour la même raison la forme normale est unique. Supposons qu'il y ait deux positions distinctes ω' et ω'' où la règle peut s'appliquer. On

introduit $\omega_1 = \texttt{deref}(t, \omega')$ et $\omega_2 = \texttt{deref}(t, \omega'')$. On peut distinguer différents cas relatifs à l'ordre des positions et à la relation de préfixe (notée $\sqsubseteq$). Certaines combinaisons de conditions sont impossibles car la règle ne peut s'appliquer que si $\omega' <_\mathcal{P} \omega_1$ et $\omega'' <_\mathcal{P} \omega_2$. De plus, chaque préfixe $\omega \sqsubseteq \beta$ implique que $\omega <_\mathcal{P} \beta$. Si nous supposons que $\omega' <_\mathcal{P} \omega''$, il y a huit cas à considérer :

$\omega'' < \omega_1 < \omega_2 \wedge \begin{cases} \omega_1 \sqsubseteq \omega_2 \\ \omega_1 \not\sqsubseteq \omega_2 \end{cases}$	$\omega'' < \omega_2 < \omega_1 \wedge \begin{cases} \omega_2 \sqsubseteq \omega_1 \\ \omega_2 \not\sqsubseteq \omega_1 \end{cases}$
$\omega' < \omega_1 < \omega'' < \omega_2 \wedge \begin{cases} \omega_1 \sqsubseteq \omega'' \wedge \omega_1 \sqsubseteq \omega_2 \\ \omega_1 \sqsubseteq \omega'' \wedge \omega_1 \not\sqsubseteq \omega_2 \\ \omega_1 \not\sqsubseteq \omega'' \wedge \omega_1 \not\sqsubseteq \omega_2 \end{cases}$	$\omega_1 = \omega_2$

Pour chaque cas, on peut prouver la confluence locale. Nous ne détaillerons que le premier cas $\omega'' <_\mathcal{P} \omega_1 <_\mathcal{P} \omega_2 \wedge \omega_1 \sqsubseteq \omega_2$, les autres étant similaires.



Si nous commençons par permuter ω' et ω_1, comme $\omega_1 \sqsubseteq \omega_2$, le sous-terme à la position ω_2 est translaté à la position $\omega' \centerdot (\!|\omega_2 - \omega_1|\!)$ et son pointeur à la position ω'' est mis à jour. Maintenant $\texttt{deref}(t, \omega'') = \omega' \centerdot (\!|\omega_2 - \omega_1|\!)$ et comme $\omega' <_\mathcal{P} \omega''$, $\texttt{deref}(t, \omega'') <_\mathcal{P} \omega''$, il n'y a pas besoin d'une seconde permutation.

Si nous commençons par permuter ω'' et ω_2, ω' n'est pas mis à jour et $\omega' <_\mathcal{P} \texttt{deref}(t, \omega')$ avec $\texttt{deref}(t, \omega') = \omega_1$. Nous avons donc besoin de permuter ω' et ω_1 et ainsi translater le pointeur à la position ω_2 vers la position $\omega' \centerdot (\!|\omega_2 - \omega_1|\!)$ ce qui ne respecte pas l'ordre sur les positions ($\omega' \centerdot (\!|\omega_2 - \omega_1|\!) <_\mathcal{P} \texttt{deref}(t, \omega' \centerdot (\!|\omega_2 - \omega_1|\!)) = \omega''$). Nous avons donc besoin d'une troisième permutation entre les positions ω'' et $\omega' \centerdot (\!|\omega_2 - \omega_1|\!)$.

Comme la fonction de normalisation termine, la confluence locale implique la convergence.

□

Exemple 29. $[\![a]\!] = a$, $[\![f(\text{-}1 \centerdot 2, a)]\!] = f(a, \text{-}2 \centerdot 1)$, et $[\![f(g(\text{-}1 \centerdot \text{-}1 \centerdot 2), h(\text{-}1 \centerdot \text{-}2 \centerdot 1))]\!] = f(g(h(\text{-}1 \centerdot \text{-}1)), \text{-}2 \centerdot 1 \centerdot 1)$

Notons que la normalisation est linéaire en la taille du terme lorsque la permutation est appliquée à gauche et en profondeur d'abord (selon la stratégie *innermost*).

Proposition 3. *$\forall t \in \mathcal{T}_{vr}(\mathcal{F}, \mathcal{X})$, nous avons $t \sim [\![t]\!]$.*

Démonstration. Soit $t \in \mathcal{T}_{vr}(\mathcal{F}, \mathcal{X})$, $[\![t]\!]$ est trivialement equivalent à t car chaque pas de normalisation préserve les fonctions $\texttt{share}(t)$ and $\texttt{exp}(t)$. En effet, la permutation entre un pointeur et son sous-terme pointé préserve le partage et comme toutes les références sont mises à jour, l'expansion est identique. □

Proposition 4. *$\forall t_1, t_2 \in \mathcal{T}_{vr}(\mathcal{F}, \mathcal{X})$, nous avons : $[\![t_1]\!] \sim [\![t_2]\!] \Leftrightarrow [\![t_1]\!] = [\![t_2]\!]$.*

Démonstration. Tout d'abord, la preuve que $[\![t_1]\!] = [\![t_2]\!] \Rightarrow [\![t_1]\!] \sim [\![t_2]\!]$ est triviale car comme $\sim$ est une relation d'équivalence, elle est en particulier réflexive. Ensuite, nous avons à prouver que $[\![t_1]\!] \sim [\![t_2]\!] \Rightarrow [\![t_1]\!] = [\![t_2]\!]$. Supposons qu'ils ne soient pas égaux, cela signifie qu'il existe un sous-terme partagé à une position ω référencé à une position ω' dans $[\![t_1]\!]$ et le contraire dans $[\![t_2]\!]$. Comme $[\![t_1]\!]$ et $[\![t_2]\!]$ sont canoniques, cela implique que $\omega <_{\mathcal{P}} \omega'$ et $\omega' <_{\mathcal{P}} \omega$ ce qui est impossible étant donné l'ordre total sur les positions. Finalement $[\![t_1]\!] = [\![t_2]\!]$. □

Proposition 5. *$\forall t_1, t_2 \in \mathcal{T}_{vr}(\mathcal{F}, \mathcal{X})$, nous avons : $t_1 \sim t_2 \Leftrightarrow [\![t_1]\!] = [\![t_2]\!]$*

Démonstration. Conséquence directe des propositions 3 et 4. □

En pratique, pour vérifier que deux termes sont équivalents, au lieu de calculer $\texttt{exp}(t)$ et $\texttt{share}(t)$, on compare directement leurs formes canoniques.

5.5.3 Réécriture de termes référencés canoniques

Un terme référencé canonique représente de manière unique un terme-graphe et il est donc plus facile d'encoder le filtrage et la réécriture car il n'est pas nécessaire de travailler modulo alpha conversion. À partir de cette représentation, nous pouvons donc simuler la réécriture de termes-graphes. L'idée est de proposer un algorithme de filtrage spécialisé et de définir le pas de réécriture de manière à ce qu'il préserve les sous-termes partagés en mettant à jour les chemins.

Filtrage

Dans la figure 5.5, nous présentons un ensemble de règles définissant le filtrage sur des termes référencés canoniques. Cet algorithme est une spécialisation de l'algorithme de filtrage syntaxique présenté dans [KK99]. Dans $\mathcal{T}_g$-Matching, E représente une conjonction de contraintes, Δ un ensemble de paires, x une variable ($\in \mathcal{X}$), f un symbole, élément de $\in \mathcal{F} \cup \{\epsilon\}$, (on a besoin de ϵ pour représenter les cycles dégénérés $\bullet$), s, t, $t_1, \ldots, t_n$ sont des termes référencés clos ($\in \mathcal{T}_g(\mathcal{F})$), π est un chemin non vide ($symb(\pi) \in \mathcal{P}^*$), p est un filtre non restreint à une variable ($\in \mathcal{T}_g(\mathcal{F}, \mathcal{X}) \setminus \mathcal{X}$), p_i sont des filtres ($\in \mathcal{T}_g(\mathcal{F}, \mathcal{X})$), $\wedge$ est le connecteur classique de disjonction. À partir de la contrainte $l \ll_\epsilon^{s,\omega} s_{|\omega} \parallel \emptyset$, la réduction termine soit sur $\top$ (l'élément neutre de $\wedge$) ou une conjonction de contraintes de filtrage la forme $p \ll_\delta^{s,\omega} t$, où δ et ω correspondent à des positions de p et t relativement à l et s (i.e. $p = l_{|\delta}$ et $t = s_{|\omega}$). Le contexte Δ correspond à l'ensemble des positions déjà visitées. Cet ensemble permet de gérer les cycles.

Decompose	$E \wedge f(p_1, \ldots, p_n) \ll_\delta^{s,\omega} f(t_1, \ldots, t_n) \parallel \Delta$	$\mapsto\!\!\!\!\rightarrow$	$E \wedge \bigwedge_{i=1}^{n} p_i \ll_{\delta.i}^{s,\omega.i} t_i \parallel \Delta \cup \{(\delta, \omega)\}$
Variable	$E \wedge x \ll_\delta^{s,\omega} t \parallel \Delta$	$\mapsto\!\!\!\!\rightarrow$	$E \parallel \Delta \cup \{(\delta, \omega)\}$
Stability	$E \wedge \pi \ll_\delta^{s,\omega} f(t_1, \ldots, t_n) \parallel \Delta$	$\mapsto\!\!\!\!\rightarrow$	$E \parallel \Delta \cup \{(\delta, \omega)\}$ si $(\!\lvert \delta \,.\, symb(\pi) \rvert\!), \omega) \in \Delta$
Dereferencing	$E \wedge p \ll_\delta^{s,\omega} \pi \parallel \Delta$	$\mapsto\!\!\!\!\rightarrow$	$E \wedge p \ll_\delta^{s,\omega'} s_{\lvert\omega'} \parallel \Delta \cup \{(\delta, \omega)\}$ où $\omega' = (\!\lvert \omega \,.\, symb(\pi) \rvert\!)$

FIG. 5.5: $\mathcal{T}_g$-Matching : filtrage pour les termes référencés canoniques

Définition 64 (Filtrage)**.** *Soient un terme $s \in \mathcal{T}_g(\mathcal{F})$ et un filtre linéaire $p \in \mathcal{T}_g(\mathcal{F}, \mathcal{X})$, $p \ll s$ si $p \ll_\epsilon^{s,\omega} s_{|\omega} \parallel \emptyset$ se réduit vers $\top \parallel \Delta$ par application de $\mathcal{T}_g$-Matching.*

Notons que l'algorithme présenté dans la figure 5.5 ne calcule pas de substitution. De plus, contrairement aux algorithmes de filtrage syntaxique, il n'y pas de règles pour gérer la non-linéarité des variables. En effet, la non-linéarité n'a pas le même sens en réécriture de termes et en réécriture de termes-graphes. En réécriture de termes, cela signifie que les sous-termes désignés par une même variable sont structurellement identiques alors qu'en réécriture de termes-graphes, comme les variables des équations représentent des sous-termes, la non-linéarité correspond au partage. Par exemple, $\{\alpha \mid \alpha = f(\beta, \beta)\}$ représente un terme de symbole de tête f et dont les deux sous-termes sont *partagés*. Il ne filtre donc pas $\{\alpha \mid \alpha = f(\beta, \gamma), \beta = a, \gamma = a\}$.

Dans notre formalisme, la réécriture de termes référencés *linéaires* est suffisante pour simuler la réécriture de termes-graphes *non-linéaires*. Par exemple, le système d'équations récursives $\{\alpha \mid \alpha = f(\beta, \beta)\}$ peut être représenté par le terme référencé $f(x, -2.1)$, où x n'apparait qu'une fois.

Une extension possible pourrait être d'offrir un mélange des deux notions. Avec notre représentation, comme les variables ne servent pas à représenter le partage, il serait possible d'identifier des sous-graphes identiques mais non partagées comme dans $\{\alpha \mid$

$\alpha = f(\beta,\gamma), \beta = a, \gamma = a\}$. Comme le concept de terme-graphe avait été introduit originellement comme une représentation efficace de termes, le partage était systématique et ce type de test n'avait pas réellement d'intérêt.

Proposition 6. *Étant donné un sujet $s \in \mathcal{T}_g(\mathcal{F})$, un filtre $l \in \mathcal{T}_g(\mathcal{F},\mathcal{X})$, une position $\omega \in \mathcal{P}os(s)$, la réduction de $l \ll^{s,\omega}_{\epsilon} s_{|\omega} \parallel \emptyset$ par $\mathcal{T}_g$-Matching est convergente.*

Démonstration. Tout d'abord, nous prouvons la terminaison en considérant la combinaison lexicographique de la relation de préfixe sur les positions ($\sqsubset$) et de $<_{\mathcal{P}}$. L'ordre strict est bien-fondé car $\mathcal{P}os(l)\times\mathcal{P}os(s)$ est fini. L'extension multi-ensemble de $\uplus_{(p \ll^{s,\omega}_{\delta} t)\in E}(\delta,\omega)$ décroit à chaque application d'une règle de $\mathcal{T}_g$-Matching.

Enfin, prouver la confluence locale est trivial puisqu'il n'y a pas d'interférence entre les règles. Lorsque deux règles r_1 et r_2 s'appliquent sur un sujet t, nous obtenons le même résultat t' en appliquant r_1 suivi de r_2 ou r_2 suivi de r_1. Comme $\mathcal{T}_g$-Matching termine, la confluence locale implique la convergence. □

Réécriture

Définition 65 (Réécriture)**.** *Étant donné $t \in \mathcal{T}_g(\mathcal{F})$ et $l, r \in \mathcal{T}_g(\mathcal{F},\mathcal{X})$:*

- *l et r sont linéaires,*
- *pour chaque variable $x \in \mathcal{V}ar(l)$, nous notons ω_{xl} sa position,*
- *t se réécrit en t' par la règle $l \to r$ si :*
  1. *il existe une position ω telle que $l \ll t_{|\omega}$ (définition 64),*
  2. *$t' = [\![\langle \dot{t}, \dot{r}\rangle_{[1.\omega \leftrightarrow 2]}]\!]_{|1}$, où*
     $\langle \_,\_\rangle$ est un symbole binaire n'appartenant pas à $\mathcal{F}$,
     $\dot{t}$ correspond à t dans lequel chaque chemin vers la position $1.\omega$ a été remplacé par un chemin vers la position 2,
     $\dot{r}$ est le terme clos correspondant à r dans lequel chaque occurrence de x (dont la position est notée ω_{xr}) est remplacée par le chemin $(\!|\overline{\omega_{xr}}.\text{-}2.\texttt{deref}(\langle \dot{t}, r\rangle, 1.\omega.\omega_{xl})|\!)$.

Dans cet algorithme, aucune substitution n'est réellement calculée car la partie droite de la règle est instanciée en remplaçant chaque occurrence de variable par un chemin vers le sous-terme correspondant dans $t_{|\omega}$. Le symbole binaire $\langle \_,\_\rangle$ permet d'appliquer la règle en connectant la partie droite de la règle (où les variables ont été remplacées par des pointeurs) avec le sujet global. Le tout forme une terme référencé $\langle \dot{t}, \dot{r}\rangle$ valide.

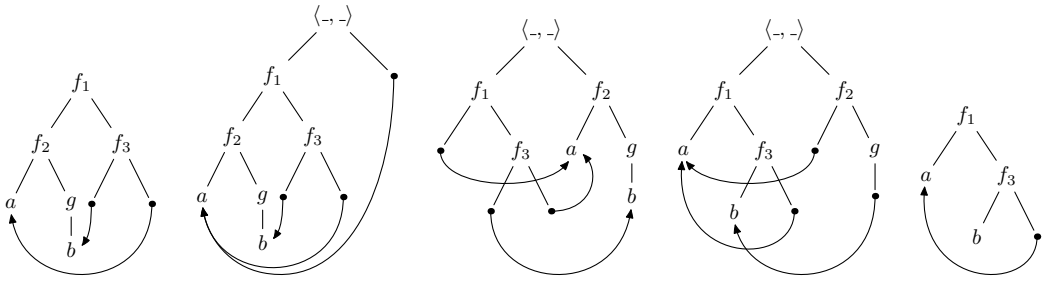
Appliquer la substitution revient simplement à permuter le réduit et la partie droite de la règle ($\langle \dot{t}, \dot{r}\rangle_{[1.\omega \leftrightarrow 2]}$). La principale subtilité de cet algorithme est d'utiliser la normalisation pour récupérer les sous-termes encore atteignables du réduit. En effet tous les sous-termes partagés dans $t_{|\omega}$ qui doivent être conservés sont translatés dans le terme et la partie inatteignable reste dans la partie droite $\langle \dot{t}, \dot{r}\rangle_{[1.\omega \leftrightarrow 2]}$. À la fin, le résultat du pas de réécriture correspond donc exactement au fils gauche de $[\![\langle \dot{t}, \dot{r}\rangle_{[1.\omega \leftrightarrow 2]}]\!]$.

La complexité du pas de réécriture est linéaire en la taille du sujet car la complexité de la permutation et de la normalisation sont linéaires.

Exemple 30. Supposons que nous voulions appliquer la règle $f(x, y) \to f(y, x)$ sur le sujet $t = f(a, b)$. La règle est appliquée en position racine, on obtient donc $\omega_{xl} = 1$, $\omega_{yl} = 2$, $\omega_{xr} = 2$, $\omega_{yr} = 1$, $\dot{t} = f(a, b)$ et $\dot{r} = f(\langle\!|\overline{\omega_{yr}}\,\texttt{.-2.}\,(1\,\texttt{.}\,\epsilon\,\texttt{.}\,\omega_{yl})|\!\rangle, \langle\!|\overline{\omega_{xr}}\,\texttt{.-2.}\,(1\,\texttt{.}\,\epsilon\,\texttt{.}\,\omega_{xl})|\!\rangle) = f(\text{-1}\texttt{.}\text{-2}\texttt{.}1\texttt{.}2, \text{-2}\texttt{.}\text{-2}\texttt{.}1\texttt{.}1)$. À partir de $\langle f(a, b), \dot{r}\rangle$, on calcule $\langle f(a, b), \dot{r}\rangle_{[1\leftrightarrow 2]} = \langle f(\text{-1}\texttt{.}\text{-1}\texttt{.}2\texttt{.}2, \text{-2}\texttt{.}\text{-1}\texttt{.}2\texttt{.}1), f(a, b)\rangle$. Après normalisation, on obtient $[\![\langle f(a, b), \dot{r}\rangle_{[1\leftrightarrow 2]}]\!] = \langle f(b, a), f(\text{-1}\texttt{.}\text{-2}\texttt{.}1\texttt{.}1, \text{-2}\texttt{.}\text{-2}\texttt{.}1\texttt{.}2)\rangle$. Finalement, le résultat $[\![\langle f(a, b), \dot{r}\rangle_{[1\leftrightarrow 2]}]\!]_{|1} = f(b, a)$.

Voyons comment les règles d'effondrement s'appliquent avec un deuxième exemple. Appliquons la règle $f(x) \to x$ sur le sujet $t = f(\text{-1})$. Comme -1 est un chemin vers la racine du réduit, on a $\dot{t} = f(\text{-1}\texttt{.}\text{-1}\texttt{.}2)$. Dans un deuxième temps $\dot{r}$ est évalué à $\dot{r} = \langle\!|\text{-2}\texttt{.}\,\texttt{deref}(\langle f(\text{-1}\texttt{.}\text{-1}\texttt{.}2), x\rangle, 1\texttt{.}1)|\!\rangle = \langle\!|\text{-2}\texttt{.}2|\!\rangle = \epsilon$ et on obtient $\langle f(\text{-1}\texttt{.}\text{-1}\texttt{.}2), \epsilon\rangle_{[1\leftrightarrow 2]} = \langle \epsilon, f(\text{-1}\texttt{.}\text{-2}\texttt{.}1)\rangle$ qui est déjà normalisé. Le résultat du pas de réécriture est donc $\langle \epsilon, f(\text{-1}\texttt{.}\text{-2}\texttt{.}1)\rangle_{|1} = \epsilon$.

Enfin, en appliquant la règle $f(x, y) \to x$ au premier sous-terme, voyons comment sont gérés les pointeurs vers des sous-termes qui disparaissent :



Dans cet exemple, le sous-terme $g(b)$ n'est pas préservé par la règle, la référence vers b est remplacée par une copie du sous-terme.

Proposition 7. *L'ensemble des termes référencés canoniques $\in \mathcal{T}(\mathcal{F})$ est clos par réécriture.*

Démonstration. Étant donné un terme $t \in \mathcal{T}_g(\mathcal{F})$, nous devons prouver que $[\![\langle \dot{t}, \dot{r}\rangle_{[1\text{.}\omega\leftrightarrow 2]}]\!]_{|1} \in \mathcal{T}_g(\mathcal{F})$. Le terme clos $\dot{r}$ n'est pas un terme référencé valide à cause des références qui remplacent les variables. Le terme $\dot{t}$ n'est pas non plus valide car certains chemins ont été remplacés par des pointeurs vers la position 2.

Mais, $\langle \dot{t}, \dot{r}\rangle$ est valide car les chemins non valides dans $\dot{t}$ pointent maintenant vers $\dot{r}$ et les chemins invalides dans $\dot{r}$ référencent des sous-termes de t. Comme la fonction de permutation préserve la validité, la forme normale $[\![\langle \dot{t}, \dot{r}\rangle_{[1\text{.}\omega\leftrightarrow 2]}]\!]$ existe. Grâce à la propriété des termes référencés canoniques (pointeurs seulement de gauche à droite), on peut conclure que $[\![\langle \dot{t}, \dot{r}\rangle_{[1\text{.}\omega\leftrightarrow 2]}]\!]_{|1} \in \mathcal{T}_g(\mathcal{F})$. □

Nous allons maintenant montrer comment la réécriture de termes-graphes (définition 48) peut être simulée par l'algorithme introduit dans la définition 65.

5.5.4 Simulation de la réécriture de termes-graphes

Nous introduisons tout d'abord la fonction ϕ qui traduit un terme référencé valide en un système d'équations récursives. Pour cela, nous avons besoin d'associer à chaque terme

$t \in \mathcal{T}_{vr}(\mathcal{F}, \mathcal{X})$ une fonction totale ψ_t de $\mathcal{P}os(t)$ vers $\mathcal{X}$. Cette fonction permet d'étiqueter tous les nœuds du terme avec une variable des systèmes d'équations récursives.

Définition 66. *Étant donné un terme $t \in \mathcal{T}_{vr}(\mathcal{F}, \mathcal{X})$, la fonction ψ_t est définie par :*

$$\psi_t(\omega) = \begin{cases} x & \text{si } symb(t_{|\omega}) \in \mathcal{F} \cup \{\epsilon\} \\ & \text{où } x \text{ est une variable fraîche} \\ t_{|\omega} & \text{si } t_{|\omega} \in \mathcal{X} \\ \psi_t(\omega') & \text{si } t_{|\omega} \in \mathcal{P}^* \\ & \text{où } \omega' = \texttt{deref}(t, \omega) \end{cases}$$

Définition 67. *Étant donné un terme référencé valide $t \in \mathcal{T}_{vr}(\mathcal{F}, \mathcal{X})$, sa représentation équationnelle $\phi(t) = \{\alpha \mid \Delta\}$ est définie telle que $\alpha = \psi_t(\epsilon)$ et Δ est un ensemble d'équations tel que :*

$$\begin{array}{rcll} \Delta & = & \{\beta = f(\beta_1, \ldots, \beta_n) & \mid \omega \in \mathcal{P}os(t), \beta = \psi_t(\omega), \beta_i = \psi_t(\omega \centerdot i), \\ & & & \quad symb(t_{|\omega}) = f \in \mathcal{F}, arity(f) = n\} \\ & \cup & \{\gamma = \bullet & \mid \omega \in \mathcal{P}os(t), \gamma = \psi_t(\omega), symb(t_{|\omega}) = \epsilon\} \end{array}$$

Théorème 3 (Simulation du pas de réécriture)**.** *Étant donné un terme référencé canonique $t \in \mathcal{T}_g(\mathcal{F})$ et une règle R, nous avons :*

$$t \to_R t' \Leftrightarrow \phi(t) \to_{\phi(R)} \phi(t')$$

Démonstration. Tout d'abord (I), nous allons montrer que les notions de filtrage dans les systèmes d'équations récursives et dans les termes référencés canoniques sont équivalentes. Plus formellement, si on note p la partie gauche de R, nous devons prouver qu'il existe une position ω telle que $p \ll_\epsilon^{t,\omega} t_{|\omega} \parallel \emptyset$ se réduit en $\top \parallel \Delta$ par l'application de $\mathcal{T}_g$-Matching si et seulement si il existe une substitution de variables σ telle que $\texttt{set}(\sigma(\phi(p))) \subseteq \texttt{set}(\phi(t))$.
Ensuite (II), nous montrerons qu'étant donné deux termes t et t' tels que $t \to_R t'$, $\phi(t)$ se réduit en $\phi(t')$ par $\phi(R)$.

(I) ($\Rightarrow$) Soit ω une position telle que $p \ll_\epsilon^{t,\omega} t_{|\omega} \parallel \emptyset$ se réduit vers $\top \parallel \Delta$, Soit σ la relation $\{(\psi_p(\delta), \psi_t(\omega)) \mid (\delta, \omega) \in \Delta\}$. Il est facile de montrer que σ est une bisimulation fonctionnelle (la notion de bisimulation fonctionnelle est présenté dans la définition 3.5 de [AK96]) :

1. σ est une fonction des variables de $\phi(p)$ vers les variables de $\phi(t)$ car c'est une relation avec un domaine et codomaine appropriés par définition et parce que chaque règle de $\mathcal{T}_g$-Matching préserve le fait que pour chaque paire (δ, ω_1) et (δ, ω_2) dans Δ, $\psi_t(\omega_1) = \psi_t(\omega_2)$,
2. le premier pas de réécriture assure que $(\epsilon, \omega) \in \Delta$ donc la racine de $\sigma(p)$ est en relation avec la racine du réduit $\psi_t(\omega)$,
3. la condition de congruence est repectée grâce à la règle Decompose et aussi parce que chaque paire de positions qui apparaît dans une contrainte a été ajoutée dans Δ lorsque la forme normale $\top$ a été atteinte.

Nous avons donc $\texttt{set}(\sigma(\phi(p))) \subseteq \texttt{set}(\phi(t))$.

($\Leftarrow$) Nous considérons maintenant qu'il existe une substitution de variables σ telle que $\texttt{set}(\sigma(\phi(p))) \subseteq \texttt{set}(\phi(t))$. Il faut donc montrer qu'il existe une position ω telle que $p \ll_{\epsilon}^{t,\omega} t_{|\omega} \parallel \emptyset$ se réduit vers $\top \parallel \Delta$ par application de $\mathcal{T}_g$-Matching. Soit ω la position définie telle que $symb(t_{|\omega}) \in \mathcal{F}$ et $\psi_t(\omega) = \texttt{root}(\sigma(\phi(p)))$. Par construction de ψ_t et ϕ_t, ω existe et est unique.

Pour montrer que chaque réduction de $p \ll_{\epsilon}^{t,\omega} t_{|\omega} \parallel \emptyset$ se réduit vers $\top$, nous montrons que tant que la conjonction de contraintes de filtrage E n'est pas réduite à $\top$, il existe une règle de $\mathcal{T}_g$-Matching qui peut s'appliquer. Comme $\mathcal{T}_g$-Matching est convergent, $\top$ est la forme normale. Grâce à la propriété de convergence prouvée dans la proposition 6, on peut choisir n'importe quelle stratégie d'évaluation, en particulier celle qui consiste à sélectionner les contraintes de filtrage ayant le plus petit δ suivant l'ordre $<_{\mathcal{P}}$. La contrainte de filtrage à considérer est unique puisqu'à chaque pas, la conjonction E contient des contraintes avec des δ distincts.
Nous considérons maintenant deux invariants nécessaires pour la suite de la preuve :

- Inv1 : $\forall \delta, \omega\ (\delta, \omega) \in \Delta \cup \Gamma \Rightarrow (\psi_p(\delta), \psi_t(\omega)) \in \sigma$, où Γ est l'ensemble des couples (δ', ω') qui apparaissent dans les contraintes de filtrage de E
- Inv2 : pour chaque position $\delta' \in \mathcal{P}os(p)$ plus petit que la plus petite position δ dans E, $\exists \omega'$ tel que $(\delta', \omega') \in \Delta$

Il est facile de montrer par induction sur les séquences de réductions que Inv1 et Inv2 sont des invariants : ils sont respectés par le terme initial $p \ll_{\epsilon}^{t,\omega} t_{|\omega} \parallel \emptyset$ et ils sont maintenu à chaque pas de réécriture, avec la stratégie choisie.

Maintenant, nous pouvons montrer qu'à chaque pas de dérivation, une règle de $\mathcal{T}_g$-Matching s'applique. En considérant la contrainte de filtrage $p' \ll_{\delta}^{t,\omega} t'$, on peut distinguer quatre cas :

1. $symb(p') \in \mathcal{F} \cup \{\epsilon\}$ et $symb(t') \in \mathcal{F} \cup \{\epsilon\}$. Dans ce cas, $symb(p') = symb(t')$ car à cause de l'invariant Inv1, $(\psi_p(\delta), \psi_t(\omega)) \in \sigma$ et comme la substitution est une bisimulation, la propriété de congruence est respectée. Decompose s'applique donc,
2. $symb(p') \in \mathcal{X}$. La règle Variable s'applique,
3. $symb(t') \in \mathcal{P}^*$. La règle Dereferencing s'applique,
4. $symb(p') \in \mathcal{P}^*$ et $symb(t') \in \mathcal{F} \cup \{\epsilon\}$. On doit montrer que $(\Vert \delta \centerdot symb(\pi) \Vert, \omega) \in \Delta$ pour appliquer la règle Stability. Grâce à Inv2 et comme $\Vert \delta \centerdot symb(\pi) \Vert <_{\mathcal{P}} \delta$ à cause de la définition 61, il existe un couple $(\Vert \delta \centerdot symb(\pi) \Vert, \omega') \in \Delta$. Ce couple étant dans Δ, cela signifie qu'une des quatre règles a été appliquée. Par la définition 61, nous savons que $symb(p_{|\Vert \delta \centerdot symb(\pi) \Vert}) \in \mathcal{F} \cup \{\epsilon\}$. De plus, comme $(\psi_p(\delta), \psi_t(\omega)) \in \sigma$ (Inv1) et $\psi_p(\Vert \delta \centerdot symb(\pi) \Vert) = \psi_p(\delta)$ par définition de ψ, on en déduit que :
   $(\psi_p(\Vert \delta \centerdot symb(\pi) \Vert), \psi_t(\omega)) \in \sigma$. Comme σ est une fonction, $\psi_t(\omega') = \psi_t(\omega)$. Il y a deux cas à considérer :
   - la règle Decompose a été appliquée donc $symb(t_{|\omega'}) \in \mathcal{F}$. Comme $symb(t_{|\omega'}) \in \mathcal{F}$ et $\psi_t(\omega') = \psi_t(\omega)$, $\omega' = \omega$.
   - la règle Dereferencing a été appliquée donc Decompose a été appliquée avec le couple $(\Vert \delta \centerdot symb(\pi) \Vert, \omega)$ et donc il appartient à Δ.

$\mathcal{T}_g$-Matching est convergent. Étant donné la stratégie choisie, une contrainte de filtrage peut toujours être réduite par une règle de $\mathcal{T}_g$-Matching. La forme normale est donc $\top$.

(II) Selon la définition 65, nous avons $t' = [\![\langle \dot{t}, \dot{r} \rangle_{[1.\omega \leftrightarrow 2]}]\!]_{|1}$, où r est la partie droite de la règle de R. Nous devons montrer que pour le système d'équations récursives L tel que $\phi(t) \rightarrow_{\phi(R)} L$ et $\psi_t(\omega) = \mathtt{root}(\sigma(\phi(p)))$ nous avons $L = \phi(t')$. Selon la définition 48, nous savons que $L = \{\alpha_t \mid (\mathtt{set}(\phi(t)) \setminus \{\alpha = \tau\}) \cup \mathtt{set}(\sigma'(\phi(r)))\}$, où $\alpha_t = \mathtt{root}(\phi(t))$, $\alpha = \mathtt{root}(\sigma(\phi(p)))$.

Tout d'abord, $\phi(\langle \dot{t}, \dot{r} \rangle)$ peut s'exprimer comme $\{\alpha' \mid \Delta\}$ où $\Delta = \{\alpha' = \langle \alpha_t, \alpha \rangle\} \cup \mathtt{set}(\phi'(t)) \cup \mathtt{set}(\sigma'(\phi(r)))$. Soit α'' une variable fraîche, $\phi'(t)$ correspond à $\phi(t)$ où l'équation $\alpha = t$ a été remplacée par $\alpha'' = t$ et s'il existe une position β et un entier positif i tel que $\omega = \beta \centerdot i$, l'équation $\psi_t(\beta) = f(\ldots, \alpha, \ldots)$ est remplacée par $\psi_t(\beta) = f(\ldots, \alpha'', \ldots)$.

Il est nécessaire pour la suite d'interpréter l'opération de permutation dans le contexte des systèmes d'équations récursives. À partir des définitions de permutation et de translation (définitions 62 et 57), il est possible de dériver $\phi(t_{[\omega_1 \leftrightarrow \omega_2]})$ de $\phi(t)$. En effet, la permutation consiste juste à échanger deux sous-termes dans t. $\phi(t_{[\omega_1 \leftrightarrow \omega_2]})$ est défini par :

- $\mathtt{root}(t_{[\omega_1 \leftrightarrow \omega_2]}) = \mathtt{root}(t)$,
- $\mathtt{set}(\phi(t_{[\omega_1 \leftrightarrow \omega_2]}))$ is $\mathtt{set}(\phi(t))$ où l'équation $\beta_1 = f(\ldots, \alpha_1, \ldots)$ est remplacée par $\beta_1 = f(\ldots, \alpha_2, \ldots)$ et l'équation $\beta_2 = f(\ldots, \alpha_2, \ldots)$ est remplacée par $\beta_2 = f(\ldots, \alpha_1, \ldots)$. α_1, α_2, β_1, β_2 sont respectivement $\psi_t(\omega_1)$, $\psi_t(\omega_2)$, $\psi_t(\pi_1)$, $\psi_t(\pi_2)$ où $\omega_1 = \pi_1 \centerdot i$, $\omega_2 = \pi_2 \centerdot j$ et $i, j \in \mathbb{N}^*$.

En appliquant l'opération de permutation, nous obtenons $\phi(\langle \dot{t}, \dot{r} \rangle_{[\omega_1 \leftrightarrow \omega_2]}) = \{\alpha' \mid \{\alpha' = \langle \alpha_t, \alpha'' \rangle\} \cup (\phi(t) \setminus \{\alpha = \tau\}) \cup \{\alpha'' = \tau\} \cup \sigma'(\phi(r))\}$. Comme deux termes référencés valides équivalents ont la même représentation par ϕ (modulo renommage des variables), nous pouvons en déduire que $\phi([\![\langle \dot{t}, \dot{r} \rangle_{[\omega_1 \leftrightarrow \omega_2]}]\!]) = \phi(\langle \dot{t}, \dot{r} \rangle_{[\omega_1 \leftrightarrow \omega_2]})$. En général, $\phi(t_{|\omega}) = \{\psi_t(\omega) \mid \Delta\}$ où Δ correspond au sous-ensemble de $\mathtt{set}(\phi(t))$ représentant les variables liées atteignables depuis $\psi_t(\omega)$. Dans notre cas, $\omega = 1$, et donc la racine est α_t. De plus, comme il n'y a pas de pointeur vers α' et α'', on peut enlever les équations correspondantes et nous obtenons finalement : $\phi(t') = \phi([\![\langle \dot{t}, \dot{r} \rangle_{[\omega_1 \leftrightarrow \omega_2]}]\!]_{|1}) = \phi(\langle \dot{t}, \dot{r} \rangle_{[\omega_1 \leftrightarrow \omega_2]_{|1}}) = \{\alpha_t \mid (\mathtt{set}(\phi(t)) \setminus \{\alpha = \tau\}) \cup \mathtt{set}(\sigma'(\phi(r)))\}$ où les équations correspondant aux variables liées inatteignables depuis α_t ont été nettoyées.
Nous avons finalement montré qu'étant donnés t et t' tels que $t \rightarrow_R t'$, la réduction de $\phi(t)$ par $\phi(R)$ est $\phi(t')$.

□

5.5.5 Résumé

Nous avons vu jusqu'à présent comment simuler la réécriture de termes-graphes en représentant de manière explicite les pointeurs dans un terme. Le concept de chemins qui est une généralisation de la notion de position permet d'encoder les pointeurs de terme. Ainsi un terme contenant des chemins est appelé terme référencé et le théorème 3 montre comment simuler la réécriture de termes-graphes à l'aide des termes référencés. Ce type de structure offre donc un moyen d'étendre facilement un langage à base de

règles avec des structures de termes-graphes. Nous allons voir maintenant comment le langage Tom a été étendu afin d'intégrer ce formalisme.

5.6 Intégration dans le langage Tom

Nous montrons dans cette section comment l'ensemble de ce formalisme a été intégré au langage Tom, en particulier au niveau du langage de signature algébrique et du langage de stratégies. L'idée principale est d'étendre automatiquement une signature algébrique de termes pour construire des termes-graphes sur cette même signature et décrire des systèmes de règles de termes-graphes. En plus de cette extension au niveau des signatures, nous avons dû étendre le langage de stratégies pour offrir de nouveaux opérateurs adaptés aux termes-graphes.

5.6.1 Termes avec pointeurs

Prenons la signature Tom suivante :

```
A = a()
  | f(t:A)
  | s(t1:A,t2:A)
  | g(b:B)

B = b()
```

À partir de cette signature, une nouvelle option a été ajoutée au générateur de structures de données (`--termpointer`) pour générer directement une signature étendue permettant de construire des termes référencés sur cette même signature. Pour chaque sorte `T` d'un module, un nouveau construceur variadique `PosT(int*)` est automatiquement généré qui nous assure que les pointeurs respectent aussi le typage des termes. De plus, le sous-terme référencé par ce constructeur doit être du même type `T`. L'exemple précédent peut donc être étendu comme suit :

```
A = a()
  | f(t:A)
  | s(t1:A,t2:A)
  | g(b:B)
  | PosA(int*)

B = b()
  | PosB(int*)
```

Nous pouvons maintenant construire le terme référencé $s(-1.2.1, a)$ avec la syntaxe `s(PosA(-1,2),a())`. Ainsi `PosA(-1,2)` référence le sous-terme `a()`. L'utilisateur peut donc construire directement des termes référencés avec les constructeurs `posT`. Cependant, il est difficile d'assurer que le terme construit à la main n'est composé que de

références correctes (qui pointent vers une position existante du terme global) et préservant les types (le terme référencé par un constructeur `posT` est de type `T`). Pour assurer automatiquement ce type de vérification, un deuxième niveau de représentation est proposée. L'utilisateur manipule des étiquettes pour représenter les références et le système génère automatiquement le terme référencé correspondant en utilisant les constructeurs de chemins. Cette représentation à base d'étiquettes peut être vue comme une implémentation des termes avec adresses présentés dans [DLL06].

La signature est donc étendue avec deux nouveaux types de constructeurs pour définir des étiquettes et étiqueter des sous-termes. Pour chaque sorte `T`, le système génère un constructeur `T LabT(s:String,t:T)` qui permet d'étiqueter un sous-terme `t` de type `T` par une étiquette `s` et un constructeur `RefT(s:String)` qui permet de déclarer une référence vers le sous-terme étiqueté par `s`.

On peut donc construire en Tom le terme `s(&l,f(l:a()))` où la syntaxe `l:t` permet d'étiqueter un sous-terme (sucre syntaxique pour `LabA("l",t)`) et la syntaxe `&l` permet de spécifier une référence (sucre syntaxique pour `refA(l)`). La construction *backquote* permet de construire à partir de ce terme le terme référencé correspondant `s(PosA(-1,2),a())`. Ce système d'étiquettes permet à l'utilisateur de ne jamais manipuler directement la notion de chemin. Le code généré fournit donc des fonctions pour générer à partir de la version étiquetée du terme la version référencée avec chemins. Cette fonction vérifie au moment de l'exécution que le terme étiqueté est bien formé dans le sens où une étiquette `s` ne peut être utilisée dans un constructeur `refT` que si cette étiquette a servi à déclarer un sous-terme `t` du même type à l'aide du constructeur `labT(s,t)`.

Pour l'instant, nous avons utilisé le système d'étiquettes uniquement pour construire des termes référencés quelconques. Les références représentent des pointeurs et on ne les interprète pas comme du partage de sous-termes.

5.6.2 Termes-graphes

Pour pouvoir manipuler directement des termes-graphes (autrement dit des termes référencés canoniques), le générateur de structures de données de Tom a été doté d'une nouvelle option `--termgraph`. L'extension de la signature est la même (constructeur `posT`, `refT` et `labT`). Seule la méthode d'expansion du terme étiqueté a été modifiée afin de construire directement un terme référencé canonique. Par exemple, l'expansion du terme `s(&l,l:a())` construit le terme référencé canonique `s(a(),PosA(-2,1))`. L'utilisateur n'a donc pas à se préoccuper de l'ordre dans lequel il étiquette les sous-termes.

Cette extension de la signature directement au niveau du générateur a plusieurs avantages. D'une part, on conserve le partage maximal des structures de données générées. D'autre part, l'intégration à Tom (génération des ancrages) et tout le support pour les stratégies est offert. On peut donc directement construire ce type de structures dans un programme Tom et utiliser le filtrage ainsi que les stratégies.

```
%strategy a_to_gb() extends Identity() {
  visit A {
    a() -> g(b())
```

```
  }
}

public static void main(String[] args) {
  A t = `s(&l,f(l:a()));
  System.out.println(`TopDown(a_to_gb()).visit(t));
}
```

La syntaxe `l:t` permet d'étiqueter un sous-terme (sucre syntaxique pour `LabA("l",t)`) et la syntaxe `&l` permet de spécifier une référence (sucre syntaxique pour `refA(l)`).

Cet exemple permet de renommer tous les sous-termes `a` en `g(b)`. La stratégie `TopDown` considère le terme `t` comme un arbre. La stratégie `TopDown(a_to_gb())` terminerait même si `t` était cyclique puisque les pointeurs sont considérés comme des constantes. Ce programme affiche donc `s(f(g(b())),PosA(-2,1,1))`.

Il est possible de définir des filtres qui sont des termes-graphes. Cependant, comme le langage Tom considère les termes-graphes générés comme des termes quelconques, les filtres doivent être directement écrit sous la forme d'un terme référencé (canonique si nécessaire). On utilise alors le filtrage syntaxique pour détecter des cycles. Par exemple, le programme suivant détecte le terme infini $f\infty$:

```
%match(t) {
  l:f(&l) -> {
    System.out.println("Terme␣infini␣f(f(...))");
  }
}
```

Cependant, si le programme réalise des transformations, la maintenance des pointeurs doit être réalisée par le programmeur et dans le cas où l'on souhaite manipuler uniquement des termes référencés canoniques, il est très difficile d'implémenter directement en Tom le pas de réécriture comme présenté dans la section 5.5.3. Nous proposons donc dans la section suivante une extension à la signature algébrique qui permet de définir directement un système de réécriture de termes-graphes dont l'implémentation se charge de gérer la maintenance des références et de la forme canonique.

5.6.3 Réécriture de termes-graphes

Afin de fournir à l'utilisateur la possibilité de spécifier un système de règles comme défini dans la section 5.5.3, la solution choisie est d'utiliser le mécanisme d'extension du langage de signatures algébriques qui permet de modifier les classes générées. Le principe est de permettre à l'utilisateur de définir un système de règles de manière déclarative comme dans l'instruction `%strategy` en utilisant la construction `graphrules`. À partir de cette définition, le générateur traite cette construction en générant dans la classe correspondant à la sorte une instruction `%strategy` qui implémente l'algorithme de réécriture présenté dans la définition 65. Un ensemble de règles est associé à une sorte. Pour la sorte `A`, on peut par exemple définir le système suivant :

```
sort A:graphrules(simpleTransformation,Identity) {
  s(&l,l:x) -> f(x)
}
```

On peut utiliser des étiquettes dans la partie gauche et droite des règles. On considère des règles linéaires gauches comme dans le pas de réécriture présenté dans la définition 65.

À partir de cette déclaration, le générateur produit une construction `%strategy` dans la classe `A` (implémentant la sorte `A`) et dont la stratégie par défaut sera `Identity` :

```
%strategy simpleTransformation extend Identity() {
  s(l:x,&l) -> {
      //applique le pas de réécriture
  }
}
```

Notons que le filtre `s(l:x,&l)` est traduit en Java sous la forme d'un terme référencé canonique. Pour l'instant, on utilise le filtrage syntaxique des instructions `%strategy` et donc on ne considère pas l'algorithme de filtrage présenté dans la définition 64 qui permet de vérifier l'existence d'une bisimulation fonctionnelle entre le terme et le filtre. Si l'utilisateur souhaite réaliser cette bisimulation, il est possible de l'encoder au niveau des stratégies en utilisant le nouveau combinateur `Deref` présenté dans la section suivante.

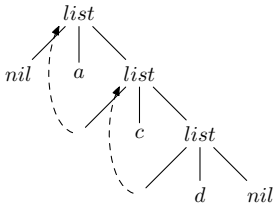
Concernant l'implémentation du pas de réécriture, elle est réalisée en utilisant intensivement les stratégies. Grâce au mécanisme d'extensions qui permet d'étendre les classes générées pour la signature, il est possible d'introduire des fonctions Java pour des opérations telles que la permutation de deux sous-termes ou la normalisation. Ces fonctions sont implémentées en utilisant les stratégies de Tom. Par exemple, la normalisation est réalisée de manière *Innermost* en effectuant une permutation dès qu'une référence est rencontrée avant le sous-terme qu'elle référence. Pour cela, la stratégie travaille sur le terme étiqueté et collecte dans un ensemble les étiquettes déjà rencontrées. Ensuite, l'application du pas de réécriture est composé des étapes suivantes :

1. sauvegarder la position courante,
2. se déplacer jusqu'à la racine du terme global (grâce à l'environnement),
3. construire $\langle \dot{t}, \dot{r} \rangle$,
4. appeler la fonction `swap` (définition 62),
5. appeler la fonction `normalise` (définition 63),
6. retourner à la position courante sauvegardée,
7. retourner le sous-terme courant.

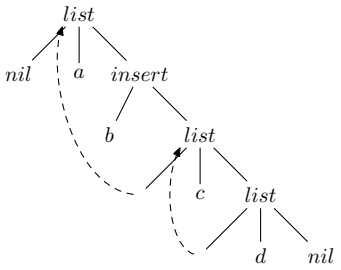
Supposons maintenant que l'on souhaite manipuler les listes doublement chaînées sous forme de termes-graphes. On définit la signature suivante :

```
List = list(previous:List,element:Term,next:List)
     | nil()
     | insert(element:Term,list:List)
```

On peut alors représenter la liste (a, c, d) par le terme-graphe suivant :



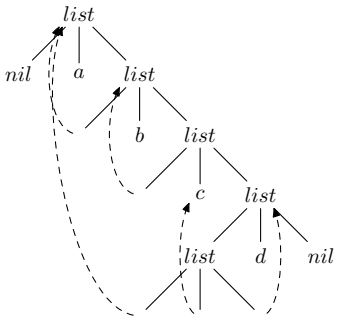
Si l'on souhaite insérer le terme b entre a et c, on construit un nouveau terme-graphe en utilisant le constructeur `insert`.



Pour réaliser concrètement l'insertion, on définit la stratégie `Insertion1` :

```
sort List: graphrules(Insertion1,Identity) {
  insert(x1,list(p,x2,n)) ->
      l:list(p,x1,list(&l,x2,n))
}
```

Si on applique la stratégie `TopDown(Insertion1())` sur le terme précédent, on obtient de manière surprenante le résultat suivant :



En effet, la réécriture de termes-graphes ne permet pas de réaliser des effets de bord. Les pointeurs vers le sous-terme `list(p,x2,n)` avant réécriture pointent toujours sur cet ancien sous-terme après réécriture. Pour que la règle se comporte correctement, il faudrait pouvoir spécifier que ces pointeurs doivent maintenant référencer le nouveau sous-terme `list(&l,x2,n)`.

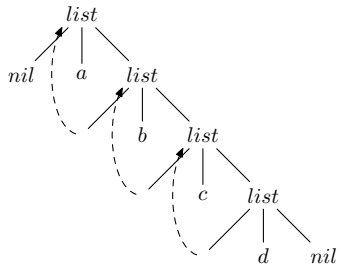
Pour cela, on autorise la réutilisation d'une étiquette de la partie gauche d'une règle en partie droite pour modifier le sous-terme étiqueté et réaliser ainsi un effet de bord

si ce sous-terme est partagé. Les règles définies dans la construction `graphrules` sont donc un peu plus expressives que celles présentées dans le formalisme. Cette extension à la réécriture de termes-graphes a été inspirée par le formalisme proposé par Rachid Echahed dans [EP06] présenté en détail à la fin de la sous-section 5.1.1. Notre extension correspond à l'action de redirection globale $\alpha \gg \beta$ où toutes les arêtes pointant vers le sous-terme étiqueté par α sont redirigées vers l'étiquette β. Dans notre cas, réutiliser une étiquette e de la partie gauche dans la partie droite pour lui assigner une nouvelle valeur t revient à déclarer une étiquette fraîche $e' : t$ et réaliser une redirection globale $e \gg e'$.

On peut donc définir la règle suivante :

```
sort List: graphrules(Insertion2,Identity) {
  insert(x1,v:list(p,x2,n)) ->
      l:list(p,x1,v:list(&l,x2,n))
}
```

Si on applique `TopDown(Insertion2())` sur le terme précédent, on obtient alors le résultat attendu :



Sans cette extension, il ne serait pas possible de définir l'insertion par une simple règle de réécriture de termes-graphes. Son implémentation est réalisée entre les étapes 3 et 4. Pour chaque étiquette partagée, on applique la fonction `globalRedirection` qui redirige tous les pointeurs référençant le sous-terme étiqueté dans le réduit vers le nouveau sous-terme étiqueté dans $\dot{r}$.

5.6.4 Extension du langage de stratégies

Le langage de stratégies tel qu'il a été défini dans le chapitre 4 ne prend pas du tout en compte les cycles et les sous-termes partagés. Le parcours des termes est réalisé comme si les références étaient des constantes de la signature. Nous proposons quelques extensions de la bibliothèque SL permettant de gérer mieux ce nouveau type de structures.

Représentation des chemins

On propose tout d'abord de définir une interface `Path` qui définit ce qu'est un chemin à partir des opérations proposées dans la section 5.3. Cette interface est implémentée à la fois par la classe `Position` déjà présentée dans le chapitre 4 et par les classes correspondant aux constructeurs `posT` générées. Le diagramme de classes présenté dans

la figure 5.6 présente les dépendances entre les classes générées par la signature et la librairie SL.

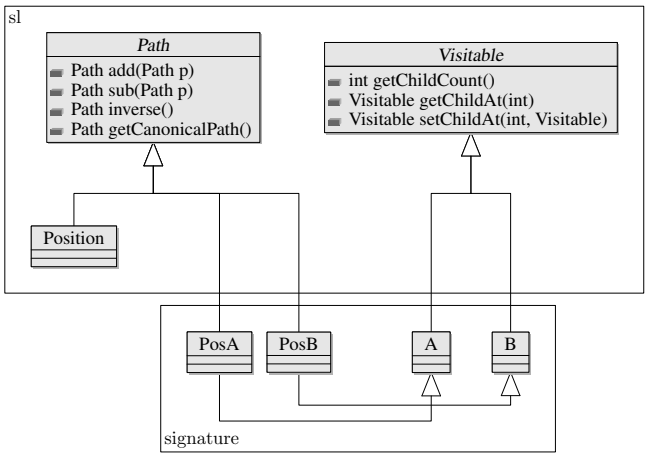


FIG. 5.6: Diagramme de classes liées à la représentation de chemins

Cela permet de composer naturellement les positions maintenues par l'environnement et les objets références contenus dans les termes-graphes. L'algorithme de réécriture peut donc être implémenté aisément.

Nouveaux combinateurs de stratégie

Dans la figure 5.7, nous introduisons la sémantique de trois nouveaux combinateurs de stratégies :

- `Deref` permet de prendre en compte les constructeurs de chemin (implémentant l'interface `Path`). En utilisant la réflexivité de `Java`, il teste si le symbole de tête est un objet d'une classe implémentant `Path`. Si c'est la cas, il réalise un déréférencement. Autrement dit, il applique la stratégie donnée en argument au sous-terme référencé,
- `AllSeq` diffère du combinateur `All` en réalisant les transformations sur les enfants de manière séquentielle (de gauche à droite) autorisant ainsi les effets de bord,
- `OneSeq` diffère du combinateur `One` en évaluant explicitement les enfants de gauche à droite et en appliquant la stratégie sur le premier n'échouant pas.

On peut ainsi redéfinir une nouvelle stratégie `TopDownGraph` qui prend en compte les références :

```
%op Strategy TopDownGraph(s:Strategy) {
  make(s) { `Mu("x",Sequence(s,AllSeq(DeRef(MuVar("x"))))) }
}
```

Supposons que l'on définisse une stratégie élémentaire `s` qui transforme les termes `a()` en `b()` et inversement. L'application de la stratégie `TopDownGraph(s)` sur le terme-

$$\frac{\langle S\{Deref(\underline{s})\}, t, \omega \cdot p\rangle \to t'}{\langle S\{\underline{Deref(s)}\}, t, \omega\rangle \to t'}\ symb(t_{|\omega})(= p) \in Path$$

$$\frac{\langle S\{Deref(\underline{s})\}, t, \omega\rangle \to t'}{\langle S\{\underline{Deref(s)}\}, t, \omega\rangle \to t'}\ symb(t_{|\omega}) \notin Path$$

$$\frac{\forall i \in [1, ar(symb(t_{|\omega}))],\ \langle S\{AllSeq(\underline{s})\}, t_{i-1}, \omega \cdot i\rangle \to t_i \text{ (avec } t_0 = t)}{\langle S\{\underline{AllSeq(s)}\}, t, \omega\rangle \to t_n}$$
$$\frac{\forall i \in [1, ar(symb(t_{|\omega}))],\ \langle S\{AllSeq(\underline{s})\}, t, \omega \cdot i\rangle \to \mathsf{fail}}{\langle S\{\underline{AllSeq(s)}\}, t, \omega\rangle \to \mathsf{fail}}$$

$$\frac{\exists i \in [1, ar(symb(t_{|\omega}))]\ \forall k \in [1, i-1],\ \langle S\{OneSeq(\underline{s})\}, t, \omega \cdot k\rangle \to \mathsf{fail} \quad \langle S\{OneSeq(\underline{s})\}, t, \omega \cdot i\rangle \to t'}{\langle S\{\underline{OneSeq(s)}\}, t, \omega\rangle \to t'}$$
$$\frac{\forall i \in [1, ar(symb(t_{|\omega}))],\ \langle S\{OneSeq(\underline{s})\}, t, \omega \cdot i\rangle \to \mathsf{fail}}{\langle S\{\underline{OneSeq(s)}\}, t, \omega\rangle \to \mathsf{fail}}$$

FIG. 5.7: Combinateurs de traversée séquentiels

graphe `s(l:a(),&l)` retourne le terme inchangé puisque le sous-terme `a()` est successivement transformée en `b()` puis `a()`. De plus, l'application de cette stratégie sur un terme-graphe infini comme `l:f(&l)` ne termine pas. Pour résoudre ce problème, une solution est d'utiliser l'opérateur `MuCycle` qui étend `Mu` en limitant le nombre de déréférencement. En effet, si l'opérateur `MuCycle` est évalué dans le même environnement que lors de son dernier appel, il s'arrête et retourne l'identité.

5.7 Synthèse

Dans ce chapitre, nous avons proposé une représentation des termes-graphes par des termes et une simulation de la réécriture de termes-graphes fondée sur cette représentation. Nous avons ainsi proposé d'utiliser la notion de chemin pour représenter les cycles et les sous-termes partagés. Un autre apport de ce chapitre est la définition d'un algorithme original permettant d'exprimer le pas de réécriture de termes-graphes par des opérations de redirection de pointeurs. On obtient ainsi une simulation de la réécriture de termes-graphes dans un langages à base de termes. L'ensemble de ces travaux a été implémenté et intégré dans le langage Tom et profite ainsi à la fois de l'environnement Java mais aussi de la bibliothèque de stratégies présentée dans le chapitre précédent. En particulier, nous avons étendu le langage de stratégies afin d'offrir des combinateurs

dédiés au graphes.

Ces travaux permettent de manipuler des graphes de flot en utilisant les constructions déclaratives du langage Tom. Dans le cadre d'analyse de programme, ce type de constructions peut par exemple être utilisé pour détecter des virus en recherchant dans le graphe de flot de contrôle d'un programme le graphe de flot de contrôle du virus par filtrage de termes-graphes comme cela a été réalisé dans [Kac08]. Nous allons voir dans le chapitre suivant des applications de ce formalisme dans la représentation de graphe de flot de contrôle du Bytecode Java. Un autre domaine d'application envisagé est celui de la transformation de modèles.

6 Application à l'analyse et la transformation de programmes Java

Contexte L'analyse statique du langage Java a donné lieu à de nombreux outils qui couvrent des problèmes variés allant de la vérification de propriétés de sûreté à l'extraction de métriques du code. La plupart de ces projets sont directement écrits dans des langages généralistes en utilisant des bibliothèques permettant de manipuler plus aisément des structures arborescentes ou des graphes. L'absence d'instruction dédiée rend cependant le code complexe et difficile à maintenir. Depuis quelques années, plusieurs projets proposent des langages dédiés à l'analyse de programmes et donc plus déclaratifs. Cette thèse s'inscrit dans la lignée de ces travaux puisque nous avons proposé des constructions de langage dédiées au prototypage d'outils d'analyse et de transformation de programmes en se fondant sur les notions issues de la réécriture de termes et de termes-graphes.

Contributions L'originalité de notre approche est d'embarquer ces constructions dans Java afin de profiter du langage généraliste lorsque cela s'avère plus adapté. La contribution de ce chapitre est de proposer plusieurs applications de ces constructions à l'analyse du langage Java. D'une part, nous présentons comment encoder une analyse sur le code source Java telle que la résolution de noms en se fondant sur les stratégies. D'autre part, nous avons proposé durant cette thèse une bibliothèque permettant de représenter et transformer un programme Bytecode dans Tom. Deux prototypes d'application de cette bibliothèque sont exposés : la compilation à la volée des termes de stratégies présentée dans [BMR07a] et l'analyse de politique de sécurité présentée dans [BMR07b].

Plan du chapitre Nous présentons tout d'abord dans la section 6.1 une étude des formalismes et des outils d'analyse statique du langage Java. Dans ce contexte, notre objectif est de proposer un langage dédié permettant de définir de manière plus déclarative des analyses sur le code source Java mais aussi directement sur le Bytecode. Dans la section 6.2, nous présentons un exemple d'analyse de code source omniprésent dans les IDE comme Eclipse offrant des fonctions de *refactoring* : la résolution de noms. Au lieu de construire une table des symboles globale au programme, ce type d'analyse peut être réalisé de manière locale en parcourant l'AST d'un nœud représentant l'utilisation d'un nom vers le nœud représentant sa déclaration. Enfin, la section 6.3 décrit comment représenter et transformer un programme Bytecode en Tom. Deux applications sont proposées : la compilation à la volée des termes de stratégie et l'analyse de politique de sécurité.

6.1 Techniques et outils d'analyse de programmes Java

6.1.1 Systèmes de types

Certains travaux sur la certification sont fondés sur l'utilisation de systèmes de types. Les premiers à avoir proposé un système de types pour le langage Java sont Stata et Abadi [SA98].

D'autres travaux s'intéressent directement à la certification du Bytecode. En effet, au moment du chargement des classes au niveau de la machine virtuelle, des vérifications sur le Bytecode sont réalisées par un composant interne à la JVM appelé communément *Bytecode verifier*. Freund et Mitchell [FM99] ont donc proposé un système de types permettant de gérer un sous-ensemble du Bytecode représentatif (initialisation d'objets, sous-routines, objets, classes, interfaces, tableaux, exceptions et types primitifs). L'article présente non seulement le système de types mais aussi un algorithme de vérification fondé sur la définition d'un graphe de flot de données. Une approche originale pour la certification du Bytecode est celle de Yelland [Yel99]. Dans ces travaux, les instructions Bytecode sont modélisées par des fonctions Haskell et il est ainsi possible d'utiliser le *type checker* d'Haskell comme *Bytecode verifier*.

Dans ce type de formalismes, l'*environnement d'exécution* est généralement modélisé par une pile d'activation (souvent appelée *frame stack*). Plus précisément, l'environnement d'exécution d'un programme est composé de deux éléments : une pile de *frames* (composées chacune d'un compteur de programme, d'une pile d'opérandes et d'un ensemble de variables locales) et d'un tas (qui contient tous les objets et tableaux alloués dynamiquement par le programme). L'*environnement de programmation* associe aux noms présents dans un programme (c'est à dire les noms des classes, noms d'interface et références de méthodes) leur définition.

Il faut d'une part vérifier que l'environnement de programmation est bien formé et d'autre part, que le code du corps de chaque méthode est bien typé. La notion d'environnement de programmation bien formé correspond à différentes vérifications telles que l'absence de cycles dans la hiérarchie de classes, la bonne implémentation de ses interfaces par une classe.

Parmi les versions allégées de Java permettant de réaliser facilement de la vérification de types, on peut citer le langage *Featherweight Java* [IPW99] qui a fait l'objet de nombreuses publications décrivant des extensions de Java et leur impact sur le système de type [IP02, Ald04]. En plus d'omettre les *threads* et la réflexivité, ce sous-ensemble ne prend pas en compte les interfaces, ni même l'assignation de variables et est souvent décrit comme le lambda calcul de Java. En effet, il décrit l'essentiel des mécanismes de calcul de Java, en particulier le concept de classes, méthodes, attributs, héritage et enfin la liaison dynamique. Ce sous-ensemble permet donc de décrire de manière compacte les règles de typage et la sémantique opérationnelle de Java.

6.1.2 Interprétation abstraite

L'interprétation abstraite [CC77] est une théorie d'approximation de la sémantique de programmes informatiques fondée sur les fonctions monotones pour ensembles ordonnés,

en particulier les treillis. Elle peut être définie comme une exécution partielle d'un programme pour obtenir des informations sur sa sémantique (par exemple, sa structure de contrôle, son flot de données) sans avoir à en faire le traitement complet. Dans le cas de l'analyse statique, c'est une puissante théorie permettant d'approximer l'information. Il faut ensuite utiliser d'autres techniques pour traiter cette information approximée et en déduire des propriétés sur le programme.

L'analyse du Bytecode peut être réalisée par interprétation abstraite de la JVM (par exemple en abstrayant les valeurs par leur type). Les premiers à avoir proposé un algorithme de certification du Bytecode sont Gosling et Yellin chez Sun. La plupart des *Bytecode Verifiers* sont fondés sur cet algorithme qui consiste à appliquer une analyse de flots de données sur l'interprétation abstraite de la machine virtuelle. Dans cet algorithme, les valeurs sont abstraites par leur type. Comme un point du programme a plusieurs prédécesseurs et successeurs (instructions de branchements et exceptions), l'analyse d'une instruction nécessite de fusionner les informations des différents prédécesseurs (cela correspond à la notion de type le plus général : chaque paire de types a un plus petit super type commun). Il existe trois types particuliers dans ce treillis. La borne supérieure du treillis correspond aux types des registres non initialisés (n'importe quelle valeur), la borne inférieure correspond à l'absence de valeurs (pour les instructions inaccessibles) et enfin le type `null` pour la référence `null`.

Julia [Spo05] est un outil générique d'analyse statique de Bytecode proposé par Fausto Spoto qui se fonde sur l'interprétation abstraite pour calculer des points-fixes à certains points du programmes appelés *watchpoints*. L'analyse est donc localisée. Il permet autant de réaliser des optimisations que des vérifications. Il est générique dans le sens où l'on peut définir les domaines abstraits et ainsi déterminer le comportement de l'analyseur. Le logiciel est structuré en deux modules principaux :

- *Romeo*, le préprocesseur de code qui utilise un module externe de domaine pour abstraire le Bytecode,
- *Juliette*, le calculateur de point-fixe.

6.1.3 Model checking

Le *model checking* est une technique de vérification automatique des systèmes informatiques (logiciels, circuits logiques, protocoles de communication). Il s'agit de tester algorithmiquement si un modèle donné, le système lui-même ou une abstraction du système, satisfait une spécification logique, généralement formulée en termes de logique temporelle. La première étape du *model checking* consiste à exprimer le modèle considéré au moyen d'un graphe orienté, formé de nœuds et de transitions. Chaque nœud représente un état du système, chaque transition représente une évolution possible du système d'un état donné vers un autre état.

Bandera [CDH$^+$00] est un ensemble d'outils de *model checking* de programmes Java. Il réalise de l'analyse statique à partir du code source pour détecter des propriétés par abstraction et *model checking*. Pour cela, l'utilisateur peut annoter son code et ainsi vérifier des propriétés supplémentaires. Cependant, en l'absence d'annotations, Bandera peut vérifier quelques propriétés comme l'absence de *deadlock* par exemple. Bandera

peut utiliser divers *model checkers* dont Spin et Java Path Finder.

6.1.4 Annotations et assistants de preuve

JML (Java *Modelling Language*) [LRR+00] est un langage d'annotations pour programmes Java, dans le style de JavaDoc, permettant de spécifier, par des assertions logiques, les propriétés du programme. Il est possible de compiler Java et JML de manière à tester ces assertions lors de l'exécution du programme ou alors de manière statique comme c'est le cas dans l'outil ESC/Java 2.

ESC/Java 2 (*Extending Static Checking*) [FLL+02] est un outil développé à l'origine par Compaq et fondé sur l'assistant à la preuve Simplify. Les assistants à la preuve sont des outils informatiques permettant de prouver de manière interactive ou automatique des théorèmes mathématiques. Suivant la logique sous-jacente, les preuves peuvent être plus ou moins automatisées. Simplify fait partie de la famille des outils automatiques. Il est possible d'annoter le code Java avec des pré- et post- conditions à l'aide du langage JML. A partir de ces annotations, l'outil génère des obligations de preuve à faire vérifier par un assistant à la preuve. D'autres projets fondés sur l'annotation JML et les assistants à la preuve automatiques sont LOOP(*Logic of Object-Oriented Programming*) [BJ01], JACK [BRL03] et Krakatoa [MPMU04].

Un projet plus récent basé sur le langage C # est Spec # [Lei06], est une extension du langage C # permettant d'annoter le code avec des contrats *à la* Eiffel [Mey92], en particulier des invariants et des pre- et post- conditions. Comme pour ESC/Java, le projet propose un outil d'analyse statique fondé sur un assistant à la preuve automatique permettant de vérifier ces contrats directement à la compilation dans l'IDE Visual Studio.

6.1.5 Analyse par réécriture

Je participe depuis deux ans à l'ANR `Ravaj` (Rewriting and Approximations for Java Applications Verification [AR07]). Le but de ce projet est d'implémenter un outil de vérification de propriétés de sécurité en se fondant sur l'atteignabilité dans les systèmes de réécriture. Cet outil permettra à terme de vérifier des propriétés de sûreté sur des programmes Java embarqués sur des téléphones portables (Midlets). Dans le cadre de ce projet, l'équipe `LANDE` de Rennes a développé un outil permettant à partir d'un programme Bytecode de générer un système de règles qui encode une exécution symbolique du programme [BGJL07]. Ma première contribution dans ce projet a été de développer une plateforme permettant de tester la conformité de ces systèmes de règles et l'exécution effective sur une JVM en vérifiant la concordance des traces. Une seconde contribution a été de développer conjointement avec Yohan Boichut une implémentation efficace de la complétion de systèmes de réécriture fondée sur les automates d'arbre présentée dans [BBGM08].

6.1.6 Analyse par détection de motifs et extraction de métriques

Le domaine de l'analyse statique ne se limite pas à la vérification de propriétés de correction sur le code. En effet, de nombreux outils fondés sur des techniques d'analyse

statique permettent d'extraire des informations concernant le code afin d'améliorer la qualité des programmes.

Par exemple, FindBugs [HP04] est un outil *ad hoc* d'analyse statique de Bytecode qui permet de détecter des *bugs* fonctionnels mais aussi de conception. Cet outil fonctionne à partir de règles définies en Java à l'aide de visiteurs. Il est donc possible pour l'utilisateur de définir ses propres analyses. FindBugs propose plus d'une centaine d'analyses fondées sur diverses méthodes :

- structure des classes et hiérarchie d'héritage,
- analyse linéaire du code,
- graphe de flot de contrôle,
- analyse de flot de données.

Un autre outil proche de FindBugs est PMD [PMD] dont l'une des originalités est d'utiliser XPath pour décrire des *bug patterns*. La détection de ces motifs est ensuite traduite par des visiteurs comme dans FindBugs. PMD diffère de FindBugs en travaillant sur le code source.

Afin d'améliorer la qualité du code, de nombreux outils permettent d'extraire des métriques à partir du code source. Le calcul de ces métriques nécessite souvent l'appel à des fonctions d'analyse statique. Il existe de nombreux outils de métriques pour Java. La plupart sont intégrés aux IDEs sous forme de *plugins* ou de tâches ant ou Maven. Les outils *open source* les plus populaires sont Checkstyle, Hammurapi et JLint.

6.1.7 Langages dédiés à l'analyse et la transformation de programmes

La plupart des outils d'analyse statique ont été écrits en utilisant des langages généralistes. Or beaucoup de ces analyses et de ces transformations peuvent s'exprimer de manière déclarative. Nous présentons donc ici les langages dédiés à l'expression d'analyses et de transformations de programmes.

Les langages dédiés à la construction de compilateurs extensibles

De nombreux langages dédiés à la construction de compilateurs ont été proposés dans la littérature. Par exemple, le langage Zephir [DR98] est un langage permettant de définir à l'aide d'une grammaire non-contextuelle la structure d'ASTs et à l'aide d'un langage d'expression régulières d'encoder des analyses relativement simples. Le langage ASTLog [Cre97] se fonde sur un sous-ensemble du langage Prolog dédié à la définition d'analyses d'ASTs, ce qui permet de profiter du filtrage implicite et du *backtracking* d'un langage logique. Ces deux langages ont été cependant principalement utilisés pour définir des analyses syntaxiques sur l'AST.

Plus récemment, avec la prolifération des langages dédiés et des extensions de langages généralistes, de nouveaux types d'outils permettant de définir des compilateurs extensibles sont apparus. En particulier, un des outils les plus utilisés est Polyglot [NCM03] qui permet de définir facilement des extensions du langage Java. Il a été utilisé par exemple pour implémenter le compilateur abc [ACH+05] du langage AspectJ. Polyglot n'est pas un langage dédié mais son approche consiste à proposer des *design patterns* adaptés à

l'écriture de compilateurs extensibles et à offrir un ensemble de bibliothèques et d'outils facilitant la mise en pratique de cette approche. On peut citer d'autres projets comme SUIF [AALL93] développé à Stanford et spécialisé dans la construction de compilateurs parallélisant ou encore JastAdd [EH07a] fondé sur les grammaires attribuées et qui sera présenté plus en détail par la suite.

En se fondant sur ce concept de compilateur extensible, on peut ainsi offrir directement des langages généralistes assez expressifs pour définir directement le DSL dans le langage hôte. Ce type de travaux se fonde en général sur des extensions de systèmes de macros. On peut citer entre autres le projet xtc [Gri05] qui étend le système de macros de C pour définir des DSLs. Cette extension nécessite cependant que le compilateur du langage généraliste soit lui aussi extensible.

Les langages à base de règles et les langages fonctionnels

L'ensemble des langages à base de règles présentés dans le chapitre 4 offrent des primitives intéressantes pour la transformation et l'analyse de programmes. En particulier, les langages ASF+SDF et Stratego ont été conçus avec comme vocation principale la transformation de programmes. Cependant, ces langages sont plus adaptés à la transformation qu'à l'analyse. Dans ce contexte, le but de cette thèse était de proposer des extensions à ces langages permettant d'offrir un meilleur support pour l'analyse.

Les langages fonctionnels offrent naturellement des primitives adaptées à la construction de compilateurs (le filtrage par exemple) et donc de manière plus générale au développement d'analyses et de transformations de programmes. Par exemple, le compilateur GHC (*Glasgow Haskell Compiler*) a été développé en utilisant de manière intensive le concept de transformations d'arbres [Jon96]. De plus, les travaux réalisés par Joost Visser et Ralf Lämmel sur Strafunski [LV02] ont montré que les langages de stratégies proposés par les langages à base de règles pouvaient s'encoder facilement en utilisant le concept de *type classes* du langage Haskell. Cependant, pour l'analyse de flot, à notre connaissance, aucun des langages fonctionnels connus n'offre de primitives permettant de manipuler des structures de graphes.

Les grammaires attribuées

Les grammaires attribuées ont été introduites par Donald Knuth en 1968 afin de définir la sémantique des langages de programmation. Une grammaire attribuée permet d'associer des valeurs aux nœuds de l'AST et il est donc très naturel de spécifier des analyses de programmes sous forme d'attributs. Le langage JastAdd [EH07a] propose une implémentation des grammaires attribuées permettant d'embarquer du code Java pour définir un attribut. Ce langage permet de définir facilement un compilateur extensible et est donc très adapté à l'analyse de programmes. En particulier, le langage JastAdd a été utilisé pour définir un compilateur Java extensible [EH07b]. Plusieurs extensions de Java ont ainsi été définies comme par exemple AspectJ [KHH+01] ou encore les génériques de Java 5. Pour conclure, concernant la définition d'analyses, cette approche est très déclarative et le langage JastAdd en mixant ses constructions avec Java offre une grande

expressivité. En particulier, les attributs référencés (attributs dont la valeur est un nœud de l'AST) permettent de définir naturellement au sein de l'AST des structures de graphes tels que les graphes de flot. On peut juste noter que sa principale limite est de ne pas offrir un très bon support pour la transformation d'AST.

Requêtes *à la* SQL

L'équipe *Programming Tools* dirigée par Oege De Moor s'intéresse beaucoup au problème de l'expression d'analyses et de transformations de programmes.

Cette équipe développe en particulier l'outil Semmle [MSV+07] qui se présente comme un plugin Eclipse d'analyse de code fondé sur un langage de requêtes *à la* SQL. Ce langage appelé `.QL` est un langage de requêtes orienté objet et qui permet d'exprimer de manière concise des requêtes sur du code. L'utilisateur peut donc définir facilement ses propres requêtes. Prenons l'exemple de la cohérence entre les méthodes `compareTo` et `equals` : la méthode `compareTo` doit retourner 0 si `equals` retourne `true`. En particulier, si la méthode `compareTo` est redéfinie mais pas la méthode `equals`, on souhaite indiquer par un avertissement une erreur potentielle à l'utilisateur. Cette vérification s'exprime par la requête `.QL` suivante :

```
from Method m
where m.hasName("compareTo") and
not( m.getDeclaringType().declaresMethod("equals") )
select m, "Is␣compareTo␣consistent␣with␣equals␣?"
```

Le langage `.QL` [MSV+07] utilisé dans Semmle est donc uniquement dédié à l'analyse et permet principalement de détecter des problèmes liés à la conception et de calculer des métriques sur le code. Un des points forts de cet outil est d'offrir de nombreux modes de visualisation des résultats d'une requête (sous forme de graphes ou de tableaux par exemple).

Parmi les travaux de l'équipe *Programming Tools* dans le domaine des langages dédiés, on peut aussi citer le langage JunGL [VPM06] dédié au *refactoring* qui a été principalement développé par Mathieu Verbaere durant sa thèse. Ce langage hybride (mélange d'un langage *à la* Datalog et d'un langage *à la* ML) permet d'encoder de manière déclarative des relations complexes entre les nœuds de l'AST (comme par exemple le graphe de flot de contrôle) et offre une évaluation de ces arrêtes à la demande.

6.2 Analyse de code source en Tom

6.2.1 Problématique du *refactoring* de code

Le *refactoring* est une technique permettant d'améliorer le code existant en réalisant de manière incrémentale des transformations source à source préservant la sémantique du programme. La plupart des IDEs proposent aux programmeurs quelques fonctions de *refactoring* comme par exemple :

- encapsuler un attribut en forçant le code à accéder à cet attribut au travers de son *getter/setter*,
- extraire une classe en déplaçant une partie d'une classe dans une nouvelle,
- déplacer une méthode ou un champ dans une classe plus appropriée,
- renommer une méthode ou un champ,
- déplacer une méthode ou un champ dans la hiérarchie d'héritage (remonter dans la super-classe ou descendre dans une sous-classe, *pull up* ou *push down*).

Prenons par exemple le programme Java suivant :

```
class A {
  static class B {
    static int x = 42;
  }
}

public class D {
  static class C extends A {
    static int x = 23;
    static int m() { return C.x; }
  }
  public static void main(String[] args) {
    System.out.println(C.m());
  }
}
```

Ce programme affiche la valeur `23` car le nom `C.x` fait référence à la variable `x` de la classe `D.C`. Supposons que nous utilisions la fonction de *refactoring* de l'IDE Eclipse permettant de renommer une classe. On renomme alors la classe `B` contenue dans `A` en `C`. La fonction de *refactoring* d'Eclipse ne détecte pas que la référence `C.x` dans la méthode `C.m` fait maintenant référence à `A.C.x` au lieu de `D.C.x`. Le programme compile mais le résultat obtenu est différent (le programme affiche maintenant `42`). Pour préserver les références, il aurait fallu renommer `C.x` en `D.C.x`. Ce petit exemple montre à quel point il est difficile d'implémenter de manière correcte ce type de fonctions et il existe de nombreuses erreurs sémantiques dans les implémentations proposées par les IDEs n'entraînant pas nécessairement d'erreurs de compilation et donc difficilement détectables par les utilisateurs.

L'implémentation de ces fonctions est donc très complexe car la sémantique du programme Java doit être préservée. En particulier, le renommage d'une variable doit préserver la résolution des noms qui consiste à lier un nom symbolique à sa déclaration. Comme les règles de visibilité du langage Java sont très complexes, il est difficile d'assurer cette préservation dans le cas général.

Traditionnellement, la résolution de noms est réalisée à la compilation par la construction d'une table des symboles en parcourant l'arbre abstrait. Le langage JastAdd présenté dans le section 6.1 propose une approche différente en associant directement ce types d'informations dans l'arbre abstrait au travers d'attributs. La définition déclarative de

ces attributs offre une plus grande modularité en isolant le traitement de chacune des constructions du langage et donc une meilleure extensibilité (pour ajouter des nouvelles règles en fonction de l'évolution du langage). Dans le cadre du *refactoring* [SEM08], cette approche a permis de prouver la correction de transformations telles que le renommage de variables.

Contrairement à la construction d'une table des symboles qui parcourt l'arbre abstrait de la racine jusqu'aux feuilles, la définition de l'attribut chargé de la résolution de noms est beaucoup plus locale. Elle consiste à rechercher depuis n'importe quel nœud de l'AST la déclaration correspondant à un nom. Supposons que ce nom soit celui d'une variable. Selon les règles de visibilité de Java, le traitement à réaliser consiste à rechercher la déclaration dans la portée courante et en cas d'échec dans la portée englobante. Une portée correspond soit à un bloc de code (qui peut être entre autres une déclaration de méthode ou de constructeur), soit à une déclaration de classe. En fonction du type de la portée, le traitement est différent :

- si c'est un bloc de code, rechercher dans la liste des instructions qui précèdent l'utilisation du nom et dans la liste des variables déclarées par le bloc (pour le bloc *for* par exemple). De plus, si c'est une déclaration de méthode ou de constructeur, rechercher dans la liste des paramètres.
- si c'est une déclaration de classe, rechercher dans les déclarations du corps de la classe. Si la déclaration du nom ne peut pas être résolue dans la classe courante, rechercher dans les déclarations de la super classe jusqu'à être à la racine de la hiérarchie d'héritage (classe `Object`). Cette analyse dépend en plus des niveaux d'accès associés qui peuvent restreindre la visibilité des déclarations.

En cas d'échec dans la portée courante, l'analyse se poursuit dans la portée englobante. Or comme en Java, une classe peut être déclarée à l'intérieur d'une autre (notion de *member classes*), la portée englobante de la déclaration d'une *member class* peut être soit un bloc, soit une autre déclaration de classe. Dans ce cas, en plus de la hiérarchie d'héritage, l'analyse peut se poursuivre dans la déclaration de la classe englobante de la *member class* et parcourir ainsi la hiérarchie d'imbrication des *member classes*.

L'analyse de noms nécessite donc de parcourir ces deux types de hiérarchies. La figure 6.1 illustre l'ordre de parcours de ces deux hiérarchies lors de l'analyse de noms dans la classe `A3`. Supposons que l'on essaie de trouver la déclaration d'une variable `v` à laquelle la classe `A3` fait référence. La résolution de noms parcourt d'abord la hiérarchie d'héritage de `A3` puis si la déclaration de `v` n'a pas été trouvée, la résolution se poursuit dans la classe englobante `A2` et ainsi de suite jusqu'à atteindre une classe qui n'est pas imbriquée (dans cet exemple, la classe `A1`).

Dans le cas de JastAdd, ce mécanisme de résolution est défini par un ensemble d'attributs comme par exemple l'attribut `lookupVariable(String name)` paramétré par `name` le nom à résoudre et de type `VariableDecl` (où `VariableDecl` est un type de nœud de l'arbre abstrait correspondant à la déclaration d'une variable).

Nous nous somme donc inspirés de cette approche pour proposer une définition de la résolution de noms à base de stratégies.

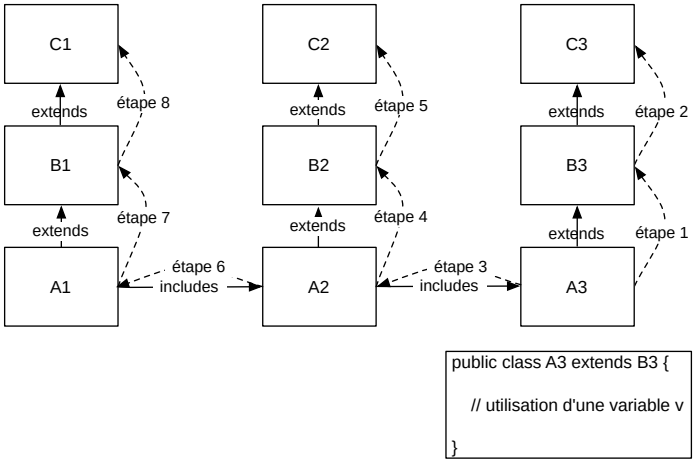


FIG. 6.1: Ordre de traversée des hiérarchies d'héritage et d'imbrication des classes

6.2.2 Résolution de noms par stratégies

On considère un sous-ensemble de Java représenté par la signature suivante :

```
Prog = Prog(Package*)

Package = Package(package:Name,classes:ClassDeclList)

ClassDeclList = ConcClassDecl(ClassDecl*)

ClassDecl = ClassDecl(name:Name,super:Name,body:BodyDeclList)

BodyDeclList = ConcBodyDecl(BodyDecl*)

BodyDecl = FieldDecl(FieldType:Name,name:Name,expr:Name)
         | MemberClassDecl(innerClass:ClassDecl)
         | Constructor(body:Stmt)

Stmt = Block(Stmt*)
     | LocalVariableDecl(VarType:Name,name:Name,expr:Name)

Name = Name(name:String)
     | Dot(Name*)
```

Cette signature permet de définir un ensemble de paquetages de classes. Une classe peut contenir des déclarations d'attributs, des classes internes et un constructeur sans argument. Le code du constructeur est formé de blocs de déclarations de variables locales. Ce sous-ensemble illustre les principales difficultés d'implémentation de la résolution de

noms. Pour simplifier la présentation, on ne prend pas en compte les mécanismes de contrôle d'accès tels que `public`, `private`,....

À partir de cette signature algébrique, la spécification de la résolution de noms appelée communément *lookup* peut être définie au travers d'une quinzaine de stratégies composées. Voici un extrait simplifié de ces définitions :

```
%op Strategy LookupVariable(pos:Position,name:Name) {
  Mu(MuVar("x"), Choice(
         LookupLocal(pos,name),
         ApplyAtEnclosingScope(MuVar("x"))))
}

%op Strategy LookupLocal(pos:Position,name:Name) {
  Mu(MuVar("x"),
      Choice(
        WhenStmt(ApplyAtPredecessors(FindName(pos,name))),
        _ClassDecl(Identity(),Identity(),
          Choice(_ConcBodyDecl(FindName(pos,name)),
                 ApplyAtSuperClass(LookupVariable(pos,name))
          )
        )
      )
    )
}
```

La stratégie `LookupVariable` permet de réaliser la résolution d'un nom à partir de n'importe quel nœud de l'arbre abstrait. Elle est paramétrée par le nom à résoudre et une position qui contiendra le résultat de la requête en identifiant le sous-terme correspondant à la déclaration. En effet, à cause du partage maximal des termes, un sous-terme ne peut pas être identifié par un pointeur Java. Cette position correspond donc au résultat de la résolution de noms. La stratégie `LookupVariable` est une stratégie récursive qui recherche la déclaration d'abord dans la portée locale puis dans la portée englobante (stratégie `ApplyAtEnclosingScope(MuVar("x"))` fondée sur le combinateur `Up`). Pour discriminer les types de portée, cette stratégie utilise les opérateurs de congruence. La stratégie `WhenStmt` permet de discriminer les nœuds de type `Smt`. La stratégie `FindName` est alors appliquée sur toutes les instructions du bloc qui précèdent le nœud courant grâce à l'opérateur `ApplyAtPredecessors`. Pour discriminer la déclaration d'une classe, `LookupLocal` utilise l'opérateur `_ClassDecl` et applique sur chacune des déclarations de la classe la stratégie `FindName` grâce au combinateur `_ConcBodyDecl` qui est un cas particulier de la fonction `map`. Si la déclaration n'a pas été trouvée, elle réapplique `LookupVariable` sur la super classe (`ApplyAtSuperClass`). Enfin, la stratégie `FindName` permet de tester si un sous-terme est de type `LocalVariableDecl` ou `FieldDecl` et si le nom correspond.

Certaines caractéristiques du langage de stratégies SL s'avèrent très utiles à l'implémentation d'une telle analyse :
- l'opérateur `Up` permet de *remonter* dans les hiérarchies,
- les opérateurs de congruence permettent de discriminer les nœuds directement au niveau du terme de stratégie,
- les positions permettent d'identifier de manière unique des sous-arbres dans des structures avec partage maximal.

Cette analyse a été réalisée dans le cadre d'une visite au sein de l'équipe *Programming tools* de l'Université d'Oxford dirigée par le Professeur Oege De Moor. Le but de cette visite était de réaliser un générateur aléatoire de programmes écrits dans un sous-ensemble de Java. Ce générateur devait permettre de tester les outils de *refactoring* développés dans cette équipe, en particulier les fonctions de renommage comme décrites dans [SEM08]. Ces travaux ont montré que le langage de stratégie était assez expressif pour exprimer de manière déclarative des analyses complexes. Cependant, contrairement aux grammaires attribuées, la définition spécifie de manière exhaustive le parcours de l'arbre. Dans les grammaires attribuées, le concept d'attributs hérités permet de propager de manière automatique de l'information dans l'arbre abstrait.

6.3 Analyse et transformation de Bytecode en Tom

Une des applications des constructions Tom présentées dans les chapitres précédents est l'analyse et la transformation de Bytecode Java. Nous avons tout d'abord développé une syntaxe abstraite pour le Bytecode permettant de manipuler un programme Bytecode comme un terme Tom. Cela nous permet ainsi de filtrer des programmes écrits en Bytecode, de les transformer puis de regénérer le Bytecode. Nous présentons dans cette section deux exemples d'application de cette bibliothèque. Une première application concerne la réalisation d'un compilateur à la volée de stratégies Tom. Une seconde application a été la vérification du respect de politiques de sécurité en générant des chargeurs de classes Java défensifs.

6.3.1 Fonctionnement de la machine virtuelle Java

Un programme Java est compilé en Bytecode. Ce dernier est destiné à être exécuté sur une machine virtuelle le rendant ainsi portable quelque soit l'architecture. Le principe de fonctionnement de la machine virtuelle Java (JVM) est celui d'une machine à pile [LY99]. A titre d'exemple, l'appel d'une méthode se fait en empilant la référence vers l'objet cible puis les différents paramètres. La méthode est ensuite invoquée, et peut accéder à ses opérandes qui sont situées sur la pile de son environnement. Les différents traitements sont alors réalisés. La valeur de retour vient remplacer la référence vers l'objet cible ainsi que les paramètres lorsque la fonction se termine.

Les types de données

La JVM manipule deux types de données :

– les types primitifs qui correspondent aux types primitifs de Java (le type `int` par exemple),
– les types référencés (correspondant aux pointeurs).

La JVM ne se préoccupe pas de la vérification de type. On considère que cette vérification a été réalisée lors de la phase de compilation. Les opérandes des instructions sont donc de type déterminé. Par exemple pour l'allocation des registres, les instructions sont typées :

Instructions	*Type associé*
`ILoad/IStore`	`int`
`LLoad/LStore`	`long`
`FLoad/FStore`	`float`
`DLoad/DStore`	`double`
`ALoad/AStore`	référence

Les types d'instruction Bytecode

Le Bytecode, composé d'environ deux cents instructions, est une sorte de code assembleur haut-niveau puisque certaines instructions permettent d'invoquer directement une méthode Java en réalisant la liaison dynamique ou encore de gérer la synchronisation de processus.

Le langage de Bytecode est composé de différentes catégories d'instructions :
– gestion des registres (instructions de type *store/load*),
– instructions arithmétiques,
– conversion de type,
– manipulation et création d'objets,
– gestion de la pile,
– transfert de contrôle,
– retour et invocation de méthodes (`invokestatic`,`invokevirtual`,...),
– propagation des exceptions,
– synchronisation.

Le Bytecode fournit par exemple quatre instructions permettant d'appeler une méthode Java :

Instruction	*Type d'appel*
`InvokeVirtual`	méthode d'instance publique avec liaison dynamique
`InvokeInterface`	méthode définie dans une interface
`InvokeStatic`	méthode statique
`InvokeSpecial`	constructeur, méthode privée, méthode de la super-classe

Prenons par exemple la méthode Java suivante :

```
void setColor(Color color) {
  Parameters p = getParameters();
  p.setColor(color);
}
```

Elle est compilée en le Bytecode suivant :

```
0:    aload_0 //empile this
1:    invokevirtual #2 //invocation de getParameters
4:    astore_2 //dépile le résultat dans le registre 2
5:    aload_2 // empile le registre 2
6:    aload_1 // empile l'argument color de setColor
7:    invokevirtual #3 //invocation de setColor
10:   return
```

Notons que les instructions `invokevirtual` sont paramétrées par deux octets permettant d'identifier une représentation de la méthode dans la *constant pool* qui correspond informellement à une table des symboles construite à l'exécution au moment du chargement des classes.

Librairies de manipulation du Bytecode

Il existe de nombreuses bibliothèques permettant de manipuler le Bytecode Java. Les plus connues sont BCEL et ASM.

La plus ancienne bibliothèque pour manipuler du Bytecode est BCEL (*Jakarta Byte Code Engineering Library*) [BCE]. Elle est encore beaucoup utilisée comme par exemple dans *AspectJ*. Cette bibliothèque souffre d'un inconvénient majeur. La structure permettant les manipulations est constituée d'un nombre important d'objets, engendrant une consommation mémoire importante et un temps d'exécution élevé.

Une bibliothèque plus récente est *ASM Bytecode Manipulation Framework* [BLC02] développée dans le cadre du consortium *ObjectWeb* et qui a tenté de palier ce problème de performance en proposant une autre approche fondée sur le *design pattern* des Visiteurs. ASM ne construit pas complètement l'AST du Bytecode mais se sert des visiteurs pour construire à la demande les sous-arbres nécessaires. Il est donc ainsi possible de définir ses propres visiteurs pour recevoir certains appels de fonction, et donc réaliser ses propres traitements. Nous nous sommes fondés sur cette bibliothèque pour construire notre propre structure de données.

6.3.2 Manipulation de Bytecode dans Tom

Nous avons opté très tôt pour une signature algébrique plutôt que pour des ancrages pour plusieurs raisons. D'une part, le générateur de structures de données de Tom nous fournit directement les ancrages nécessaires et d'autre part, notre structure de données profite du partage maximal obtenu par *hash-consing* nous évitant ainsi les problèmes de mémoire de BCEL. Afin de générer un terme Tom à partir d'un programme Bytecode, nous avons choisi d'utiliser la bibliothèque ASM car le système fondé sur les visiteurs est bien moins gourmand en mémoire et en temps d'exécution que la méthode utilisée par BCEL.

Voici un extrait de la signature des ASTs Bytecode manipulés par Tom :

```
module Bytecode
```

```
imports int long float double String
abstract syntax
TClass = Class(info:TClassInfo, fields:TFieldList,
               methods:TMethodList)
...
TMethodList = MethodList(TMethod*)
TMethod = Method(info:TMethodInfo, code:TMethodCode)
TMethodCode = MethodCode(instructions:TInstructionList,
                         localVariables:TLocalVariableList,
                         tryCatchBlocks:TTryCatchBlockList)
...
TInstructionList = InstructionList(TInstruction*)
TInstruction = Nop()
             | Iload(var:int)
             | Ifeq(label:TLabel)
             | Invokevirtual(owner:String, name:String,
                             methodDesc:TMethodDescriptor)
 ...
```

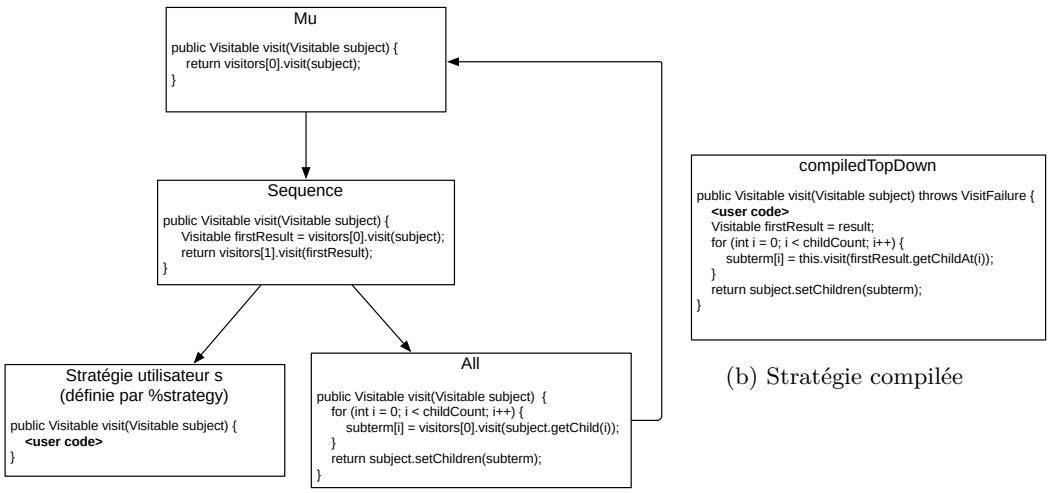
La signature complète contient un peu plus de 250 constructeurs. Cet extrait donne une idée du format de représentation. Une classe est représentée par un constructeur `Class` prenant comme arguments un ensemble d'informations telles que le nom de la classe et le paquetage. Les arguments principaux sont la liste des attributs ainsi que la liste des méthodes. Ces deux listes sont représentées par des opérateurs variadiques, ce qui permet d'utiliser le filtrage associatif pour rechercher un élément. Une méthode est définie à l'aide du constructeur `Method` dont l'argument principal est la liste des instructions Bytecode. Chaque instruction est représentée par un constructeur tels que `Nop` ou `Iload`.

6.3.3 Transformation de Bytecode : compilation à la volée des stratégies

Comme cela a été présenté dans le chapitre 4, la bibliothèque de stratégie SL est fondée sur une adaptation du patron de conception des visiteurs. Lorsque les stratégies sont composées de plusieurs dizaines de combinateurs comme c'est le cas par exemple dans l'optimiseur de Tom, l'implémentation par visiteurs engendre un nombre très important d'appels à la méthode `visit`. La majorité du temps d'exécution est alors passée à créer les différentes *frames* de ces appels. Pour améliorer les performances à l'exécution, nous proposons de compiler à la volée les termes de stratégies pour composer directement au niveau du Bytecode les différents appels à la méthode `visit`. Le principe du compilateur de stratégies est donc de générer une nouvelle stratégie spécialisée ayant le même comportement qu'une stratégie composée mais où les appels aux autres visiteurs ont directement été composés dans le Bytecode de sa méthode `visit`.

La figure 6.2 présente le code obtenu lors de la compilation de la stratégie composée `TopDown`. La *compilation* des stratégies consiste donc à fusionner au sein d'une seule

stratégie le code des différentes méthodes `visit` des combinateurs rencontrés. En réalisant une telle fusion, un grand nombre d'appels de méthodes `visit` entre les différents visiteurs disparaît. Nous avons choisi de réaliser cet *inlining* à la volée pendant l'exécution du programme plutôt qu'au moment de la compilation de Tom. En effet, cette optimisation nécessite d'analyser le code or le compilateur de Tom est conçu de manière à ne pas réaliser d'analyse sur le code Java environnant. De plus, réaliser ce type d'analyse au niveau du Bytecode ne présente aucune difficulté puisque les instructions sont relativement de haut-niveau.



(a) Stratégie composée d'origine

(b) Stratégie compilée

FIG. 6.2: Inlining de la stratégie Top-Down

La compilation des stratégies est composée des étapes suivantes :

1. charger le Bytecode de la classe de la racine du graphe de stratégies,
2. rechercher dans le Bytecode la définition de la méthode `visit`,
3. parcourir les instructions afin de détecter un appel à un autre `visit` avec une stratégie fille `s`. Si `s` est un combinateur (déterminé par réflexivité) alors l'algorithme est réappliqué sur `s`. Sinon, la stratégie `s` est simplement ajoutée comme argument à la stratégie compilée et l'appel est ajouté dans le code de la méthode `visit` de la stratégie compilée.

Après avoir construit l'AST de l'instance racine du graphe de stratégies, la première étape consiste à trouver la méthode `visit` par inspection de la liste des méthodes de l'AST correspondant :

```
Class[method=MethodList(_*,Method(MethodInfo[name="visit"],code),_*)]
```

Maintenant que nous avons récupéré dans la variable `code` le corps de la méthode de la racine, la seconde étape consiste à filtrer les appels à la méthode `visit` dans

ce code. Chaque combinateur de stratégies étend la classe abstraite AbstractCombinator et accède donc à ses paramètres via l'attribut `visitors[]`. Par exemple, dans la classe `Sequence`, `visitors[0]` correspond à la première stratégie à appliquer et `visitors[1]` à la seconde. Le code de la méthode `visit` de la classe `Sequence` se termine par `return visitors[1].visit(firstResult)`. Le compilateur de stratégies doit donc détecter l'appel à la méthode `visit` sur les éléments de ce tableau.

```
%strategy InlineVisitCode() extends Identity() {
  visit TMethodCode {
    /* Match the bytecode of visitors[index].visit(_)'*/
    Code( Aload(0),Getfield[name="visitors"],indexInst,Aaload[],
          Invokeinterface[name="visit"],tail*) -> {
      /* retrouver le code correspondant */
      newCode = ...
      return `Code(newCode, tail*);
    }
  }
}
```

Dans cet exemple, le motif Java `visitors[index].visit(_)` est identifié par cinq instructions Bytecode en séquence : `Aload(0)` met le contenu du registre `0` sur la pile (correspond à `this`), `Getfield` permet de récupérer l'attribut `visitors`. La variable `indexInst` peut être instanciée par n'importe quelle instruction Bytecode car il existe différentes manières d'empiler l'index du tableau sur la pile (à l'aide d'un attribut ou d'un registre par exemple). `Aaload` correspond à l'accès à `visitors[index]`. Enfin, `Invokeinterface` appelle la méthode `visit`. L'instruction `newCode = ...` détermine le type dynamique de ce nœud du graphe de stratégies, charge la classe correspondante, et charge la représentation algébrique du code inliné de la méthode `visit` dans la variable locale `newCode`. L'inlining est réalisé en remplaçant les cinq instructions Bytecode par le terme `newCode` dans `` `Code(...) ``.

Grâce aux capacités réflexives de Java, le type dynamique de `visitors[index]` peut être déterminé et si ce type correspond à celui d'un combinateur de la bibliothèque, l'implémentation de `visit` est inlinée. Dans l'exemple de la figure 6.2, le type dynamique de `visitors[1]` est `All` et l'appel à `visit` peut être remplacé par l'itération Java sur les enfants du sujet. Les stratégies utilisateurs ne sont donc pas inlinées ce qui permet de modifier leurs arguments. En effet, il est fréquent d'utiliser une stratégie au sein d'une boucle ayant un paramètre qui est modifié au cours de l'itération. En laissant les stratégies utilisateurs comme fils du bloc compilé, il est possible de réaffecter l'un d'entre eux avec une nouvelle stratégie. De ce fait, l'utilisation d'une stratégie compilée dans une boucle reste possible comme le montre l'exemple suivant :

```
Strategy compiledStrategy =
  `BottomUp(Sequence(S(i), Identity())).compile();
for (int i =0; i<5; i++) {
  compiledStrategy.setChildAt(0, `S(i));
```

```
  compiledStrategy.visit(t);
}
```

La gestion de la récursion via l'opérateur `Mu` est un peu plus délicate. La méthode adoptée consiste à créer une nouvelle méthode dans la stratégie compilée. Les appels à la méthode `visit` du `Mu` sont alors réécrits pour réaliser un appel à la nouvelle fonction. Le `visit` du `MuVar` est remplacée par un simple appel à cette nouvelle fonction. La gestion de la récursion (sauvegarde des registres...) se fait ainsi automatiquement par la JVM.

Pour que la stratégie `InlineVisitCode` génère du code correct, il ne doit pas y avoir de capture de variable. Dans notre cas, un simple renommage des variables avec des noms frais suffit. Avant chaque inlining, une phase de renommage est donc appliquée pour modifier tous les accès aux registres (changement du numéro) et aux champs de classe (renommage) ou pour transformer les étiquettes utilisées dans les instructions de contrôle du flot telles que `Goto` et `Label`. Cette phase peut être réalisée par une stratégie que nous appellerons `Rename`.

Ces deux stratégies sont donc réalisées à l'aide de la stratégie `TopDown` et combinées à l'aide du combinateur de séquence :

```
Strategy inliner =
 `Sequence(
   TopDown(Rename()),
   TopDown(InlineVisitCode())
);
newCode = (TMethodCode)inliner.visit(`code);
```

Pour réaliser l'ensemble des étapes de compilation, une méthode `compile` peut être ajoutée à la classe `AbstractCombinator` qui génère à partir de l'instance correspondant à la racine du graphe de stratégies une nouvelle classe qui correspond au terme de stratégie compilé et retourne une instance de cette classe. Au lieu de générer un fichier pour cette nouvelle classe, son code est directement chargé dans la JVM.

Résultats expérimentaux

La figure 6.3 présente un ensemble de résultats expérimentaux du compilateur sur différentes applications. Pour chacun des exemples, le premier temps correspond à l'exécution de la stratégie non-compilée et le deuxième temps à l'exécution de la même stratégie mais cette fois-ci compilée.

Le premier exemple correspond à la normalisation d'arbres binaires dont la taille varie de 2^2 à 2^{32} nœuds. Les stratégies utilisent l'échec, ce qui signifie qu'à chaque fois qu'une règle échoue, une exception est lancée et est ensuite capturée par l'un des combinateurs du graphe de stratégies. On obtient donc un facteur 2 car après inlining, le traitement des exception est réalisée de manière locale dans la même méthode au lieu d'être distribué dans plusieurs méthodes.

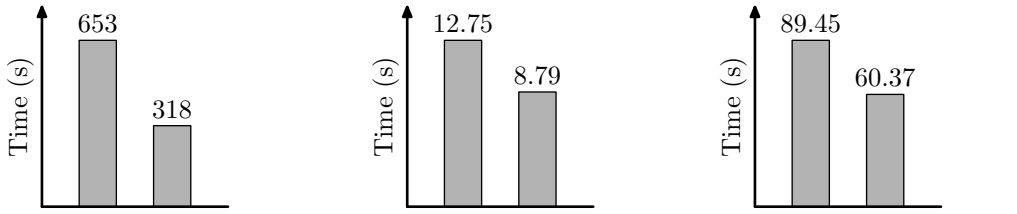


FIG. 6.3: Résultats expérimentaux de l'optimiseur de stratégies

Les deux autres exemples sont des applications réelles utilisant les stratégies de manière intensive. L'exemple du calcul des structures est un outil de preuve de théorèmes fondé sur le calcul des structures, comme présenté dans [KMR05a]. Ce système utilise l'*inférence profonde*, ce qui nécessite de traverser le terme de preuves en profondeur afin de détecter de nouveaux réduits où les règles de déduction peuvent s'appliquer. La dernière application est un outil permettant de trouver des quasi-interprétations de programmes fonctionnels [BMP06]. Cela implique de manipuler des expressions polynomiales en les traversant afin d'appliquer des transformations comme l'application d'une substitution par exemple.

6.3.4 Analyse de Bytecode : application de politiques de sécurité

Dans ce paragraphe, nous présentons une application de réécriture de Bytecode permettant de réaliser des chargements sûrs de classes, en redirigeant certains appels de méthodes forçant ainsi l'utilisation de bibliothèques d'accès aux fichiers. Au lieu de modifier directement la machine virtuelle, cette approche de chargement défensif de classes permet une plus grande modularité comme cela a été démontré dans [RW02]. Pour cela nous définissons une sous-classe de la classe `ClassLoader` spécialisée afin de détecter automatiquement si une classe chargée contient des parties de code non sûres. Dans notre exemple, nous allons rechercher des appels du type `new FileReader(nameFile).read()` et les remplacer par un appel vers une bibliothèque plus sûre.

Le cœur de notre programme correspond donc à la classe `SecureClassLoader`. Cette classe contient en particulier une stratégie élémentaire permettant de détecter les motifs de code non sûr et de les réécrire :

```
%strategy FindFileAccess() extends Identity() {
  visit TInstructionList {
    /* match new FileReader(nameFile).read() */
    InstructionList(before*,
      New("java/io/FileReader"),
      Dup(),
      Aload(number),
      Invokespecial[owner="java/io/FileReader",name="<init>"],
      Invokevirtual[owner="java/io/FileReader",name="read"],
```

```
      Pop(),
      after*) ->
      /* replace it with new SecureAccess(nameFile).sread() */
        `InstructionList(before*,
            New("SecureAccess"),
            Dup(),
            Aload(number),
            Invokespecial("SecureAccess","<init>"),
            Invokevirtual("SecureAccess","sread"),
            Pop(),
            after*);
  }
}
```

Les variables `before*` et `after*` sont instanciées par une liste d'instructions. Les termes entre les variables `before*` et `after*` (`New(...)`, `Dup()`,...) sont les termes Tom représentant le Bytecode du code Java `new FileReader(nameFile).read()`. Nous les remplaçons donc par les instructions correspondant à l'invocation de la méthode `sread` de la classe `SecureAccess`.

Cette règle ne capture pas toutes les séquences d'instructions Bytecode sémantiquement équivalentes. En effet, dans un programme réel, l'instanciation de `FileRead` et l'appel à la méthode `read` peuvent être éloignés dans le code. De telles situations sont gérées par la recherche d'une instruction `read` en suivant le graphe de flot de contrôle et en utilisant la stratégie correspondant à l'opérateur temporel EF. Nous verrons dans les paragraphes suivants comment représenter le CFG.

En utilisant le filtrage associatif de Tom, nous appliquons cette stratégie sur toutes les méthodes de la classe :

```
public TClass fileAccessVerify(TClass aclass) {
  TMethodList methods = aclass.getmethods();
  TMethodList secureMethods = `MethodList();
    %match(methods) {
      MethodList(_*, x, _*) -> {
        TInstructionList ins = `x.getcode().getinstructions();
        TInstructionList secureInstList =
          `TopDown(FindFileAccess()).apply(ins);
        TMethodCode secureCode =
          `x.getcode().setinstructions(secureInstList);
        TMethod secureMethod = `x.setcode(secureCode);
        secureMethods = `MethodList(secureMethods*,secureMethod);
      }
    }
    return aclass.setmethods(secureMethods);
}
```

Les méthodes `getmethods` et `setmethods` sont définies dans la bibliothèque Tom et permettent respectivement de récupérer et modifier le sous-terme correspondant à la liste des méthodes associées à un terme du type `TClass`. De la même manière, `getcode` et `setcode` appliquées à un terme de la sorte `TMethod` permettent respectivement de récupérer et modifier le sous-terme correspondant au code tandis que `getinstruction` et `setinstruction`, appliquées à un terme de type `TMethodCode` agissent sur le sous-terme correspondant à la liste des instructions.

On peut maintenant définir la méthode `transform` qui à partir d'un nom de fichier retourne un tableau d'octets correspondant à la classe modifiée :

```
public byte[] transform(String file) {
    BytecodeReader br = new BytecodeReader(file);
    TClass c = br.getTClass();
    c = fileAccessVerify(c);
    BytecodeGenerator bg = new BytecodeGenerator();
    return bg.toBytecode(c);
}
```

Après avoir récupéré le terme Tom de type `TClass` à partir du nom de fichier donné en paramètre, nous appliquons la transformation défensive du code à l'aide la méthode `fileAccessVerify` qui retourne un nouveau terme `c`. La classe `BytecodeGenerator` permet de générer un tableau d'octets correspondant à la classe transformée.

Dans notre chargeur de classe défensif appelé `SecureClassLoader`, la seule méthode à redéfinir est `loadClass`. Notre transformation est alors réalisée sur le Bytecode juste avant le chargement réalisé à l'aide des méthodes `defineClass` (génère une instance de la classe `Class` à partir d'un tableau d'octets) et `loadClass` (charge la classe dans la JVM).

En utilisant ce chargeur de classes défensif, nous pouvons assurer qu'aucune classe utilisant les méthodes d'accès aux fichiers non protégés ne pourra être chargée en redirigeant au moment du chargement de la classe chaque appel de méthode correspondant à un accès de fichier vers une classe intermédiaire plus sûre. Afin de simplifier la présentation, la recherche des appels n'est réalisée que par la détection du motif `new FileReader(nameFile).read()` dans le Bytecode. Dans une application réelle, une analyse du flot de contrôle serait nécessaire afin de détecter la création d'une instance de `FileReader` suivi dans la suite du CFG par un appel à la méthode `read()`.

Le chargeur généré assure par analyse et transformation du Bytecode les redirections d'appels de méthodes et c'est la classe `SecureAccess` qui assure la vérification d'une politique de sécurité donnée en paramètre spécifiée en XACML (format XML standardisé pour spécifier une politique de sécurité).

6.3.5 Analyse de flot de contrôle

La plupart des analyses nécessitent la manipulation de graphes de flot tels que le graphe de flot de contrôle ou encore le graphe de flot de données. Nous présentons ici deux approches possibles. L'approche la plus naturelle est d'utiliser l'extension du

langage Tom pour les termes-graphes afin de représenter explicitement ces nouvelles structures. Une deuxième approche proposée est de simuler ces structures directement sur l'AST en utilisant le langage de stratégies de Tom. Nous allons dans les paragraphes suivants présenter ces deux approches en construisant le CFG d'un programme Bytecode.

Analyse de flot par termes-graphes

Une première solution consiste à construire directement le terme-graphe correspondant au graphe de flot d'un programme et de réaliser l'analyse sur cette nouvelle structure. Cette transformation est réalisée grâce à l'extension du langage Tom pour les termes-graphes. On peut spécifier par un système de règles la traduction de l'AST vers le CFG en utilisant les mêmes étiquettes que dans l'AST. On enrichit la signature algébrique à l'aide d'un nouveau constructeur `Node(i :TInstruction,next :TInstructionList)` qui constitue un nœud du CFG. Son premier argument `i` correspond à l'instruction Bytecode de ce nœud et son deuxième argument est une liste de pointeurs vers ses suivants (cette liste est construite en utilisant le nouveau constructeur `Next` de domaine `TInstruction` et de codomaine `TInstructionList`). On définit alors la stratégie `AstToCfg` qui traduit l'AST Bytecode en son CFG :

```
%strategy AstToCfg() extends Identity() {
  visit TInstruction {
      Anchor(l) -> LabTInstruction(l,Node(Anchor(l),Next(nextinst)))
      Goto(l) -> Node(Goto(l),Next(RefTInstruction(l)))
      i@(Ifeq| ... |Jsr|Ifnull(l)) ->
        Node(i,Next(RefTInstruction(l),nextinst))
      Return() -> Node(Return(),Next())
      ...
      x -> Node(x,Next(nextinst))
  }
}
```

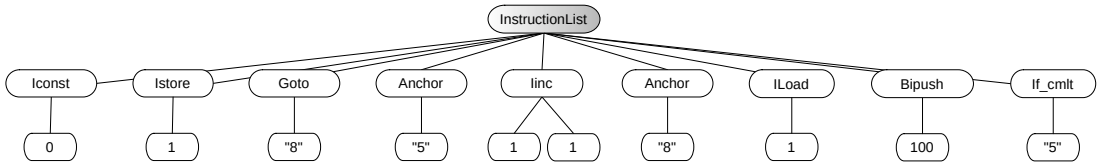
Notons que la variable `refinst` représente un pointeur vers l'instruction suivante dans la liste des instructions de l'AST.

Prenons l'exemple de code Java suivant :

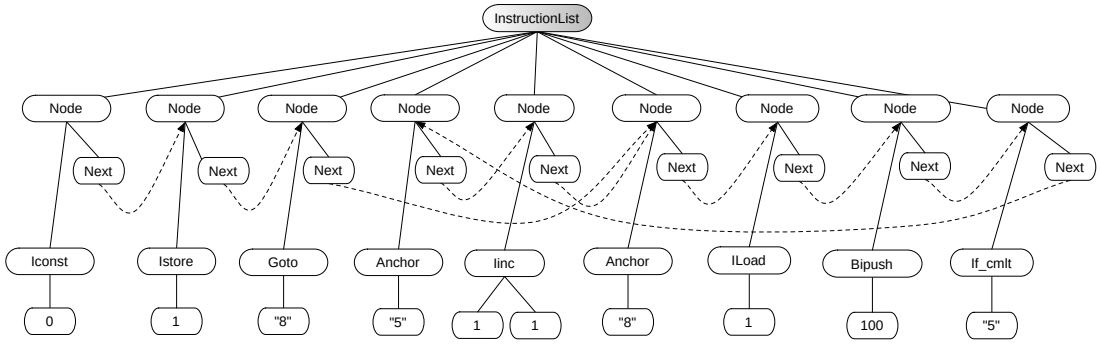
```
int i = 0;
while (i < 100) {
  i++;
}
```

Le Bytecode correspondant est :

```
0  Iconst_0()
1  Istore_1()
2  Goto(8)
5  Iinc(1,1)
```



(a) AST



(b) CFG

FIG. 6.4: AST et CFG d'une boucle Java

```
8  Iload_1()
9  Bipush(100)
11 If_icmplt(5)
```

La figure 6.4 montre l'AST pour cette liste d'instructions et le CFG obtenu par application de la stratégie `TopDown(AstToCfg())`.

Analyse de flot par stratégies

Nous allons montrer dans ce paragraphe comment étendre le langage de stratégies SL de manière à ce que la traversée des programmes Bytecode soit réalisée comme si on avait représenté le graphe de flot de contrôle.

Pour cela, nous prenons un exemple de transformation qui consiste à éliminer les instructions de type `store` inutiles. Prenons, par exemple, le programme Bytecode suivant :

```
Istore(3);Istore(2);Iload(3);Istore(2);Iload(2)
```

On souhaite détecter dans le graphe de flot de programme les instructions `store` inutiles, autrement dit, celles pour lesquelles il n'y a pas d'instruction `load` avant la prochaine instruction `store` ayant le même index. Dans cet exemple, la première instruction `IStore(2)` peut être supprimée. La vérification qu'un `store` est inutile peut s'exprimer par la formule CTL suivante :

$$IsStore(i) \wedge AX(AU(\neg IsLoad(i), IsStore(i)))$$

Où *IsLoad*(i) (respectivement *IsStore*(i))) est vrai si le nœud courant correspond à l'instruction `Load(i)` (respectivement `Store(i)`).

Cette recherche n'a de sens que si elle est réalisée non pas sur l'arbre de syntaxe abstraite (AST) mais sur le graphe de flot de contrôle (CFG). Une première solution pourrait être d'utiliser les termes-graphes pour représenter le CFG mais cela nécessiterait de traduire l'AST. Une autre solution est d'utiliser les stratégies comme un moyen de simuler le CFG sans le construire explicitement. Pour cela, il suffit de spécialiser les opérateurs de traversée (`All`, `One` et `Up`) de manière à ce qu'ils considèrent comme enfants d'un nœud les instructions suivant ce nœud dans le CFG. Par exemple, si l'on considère le code Java précédent constitué d'une boucle, la figure 6.5 présente le comportement attendu par le nouvel opérateur `All`.

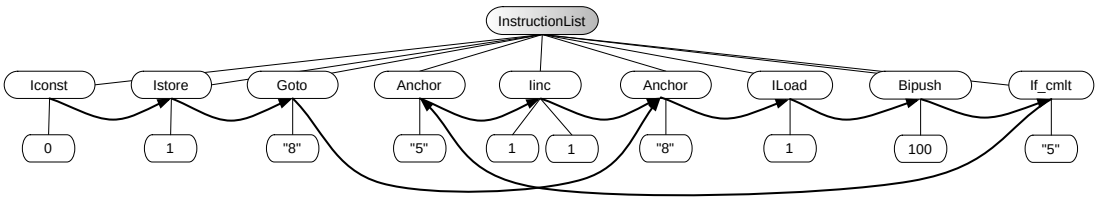


FIG. 6.5: Exemple de comportement attendu par l'opérateur `All` spécialisé

Les flèches représentent la relation de sous-termes au niveau du CFG. La stratégie `All` peut donc être adaptée afin de parcourir le CFG d'un programme Bytecode. Pour cela, il est nécessaire de la paramétrer par une table associant à chaque étiquette du programme Bytecode la position correspondante. Par exemple, l'instruction `Goto` n'a qu'un seul enfant qui correspond au sous-terme étiqueté et une instruction de type `If_XX` a deux enfants suivant la valeur de l'expression. Cette stratégie appelée `AllCfg` peut être définie pour le `Goto` comme suit :

```
%strategy AllCfg(s:Strategy, etiquettes:Map) extends Identity() {
  visit TInstruction {

    Goto(l) -> {
      Position p = etiquettes.get(‘l);
      return p.getOmega(s).visit(getEnvironment());
    }

    ...

  }
}
```

L'application de la stratégie `AllCfg` sur un nœud de type `Goto(l)` a pour effet d'appliquer la stratégie `s` donnée en argument au nœud étiqueté par `l`, contrairement à la

stratégie `All` qui aurait appliqué `s` sur le sous-terme `l`.

L'intérêt de l'approche par stratégies est qu'il n'est pas nécessaire de construire une structure supplémentaire. L'inconvénient est qu'en cas de modification de l'AST, la table des étiquettes doit être mise à jour. Dans le cas de l'approche par termes-graphes, la réécriture de termes-graphes comme présentée dans le chapitre 5 assure la mise à jour automatique des pointeurs du CFG.

6.4 Synthèse

Dans ce chapitre, nous avons vu comment les constructions de langage proposées dans cette thèse peuvent être utilisées pour l'analyse et la transformation de code source Java et de Bytecode. Par rapport aux autres projets présentés au début du chapitre, le point fort de cette approche est principalement la constante connexion entre les constructions du langage dédié et le code écrit dans le langage hôte. Par exemple, le principe des ancrages offre une très grande souplesse dans la manière de s'intégrer à un outil d'analyse existant. Suivant le contexte d'utilisation, l'utilisateur peut choisir de conserver ses propres structures ou profiter du générateur offert par le langage Tom. Par rapport aux langages dédiés à la transformation comme par exemple les langages à base de règle, notre langage se distingue par son langage de stratégies qui permet d'exprimer des parcours plus complexes et par sa capacité à manipuler des structures cycliques telles que les termes-graphes. Les applications présentées dans ce chapitre démontrent son pouvoir d'expressivité.

En comparaison avec des langages dédiés uniquement à l'analyse telle que les grammaires attribuées, cette approche est cependant moins élégante. En effet, les stratégies étant à la base un moyen de contrôler l'application des règles, elles ne permettent pas dans leur état actuel de renvoyer par exemple un objet d'un type différent de celui sur lequel elles s'appliquent. Le résultat doit pour l'instant être conservé sous forme d'un paramètre Java de la stratégie. L'approche par attributs offre donc beaucoup plus de flexibilité dans la manière de combiner différents types de résultats. En ce qui concerne l'aspect transformation, l'approche par attributs est néanmoins relativement inadaptée. Les travaux de cette thèse sont donc un premier pas vers des langages dédiés à la fois à l'analyse et la transformation. En particulier, ces premières expériences d'analyse de code source et de Bytecode nous ont amené à réfléchir à des améliorations pour faciliter encore plus l'expression d'analyses qui seront discutées dans la conclusion de cette thèse.

Conclusion

L'analyse statique permet d'accroître la qualité et la sûreté des programmes en garantissant certaines propriétés ou en détectant, avant exécution, des erreurs éventuelles. En se fondant sur des techniques de réécriture et de programmation par stratégies, la finalité de cette thèse était de concevoir un langage dédié à l'analyse et la transformation de programmes. Le langage Tom développé dans l'équipe a été le support de ces travaux.

Contributions

Un cadre formel pour les langages dédiés embarqués

Un des points forts du langage proposé dans cette thèse est son intégration dans un langage généraliste. Or lorsque le langage dédié est fortement lié au langage hôte (en partageant une partie de son environnement d'exécution par exemple), il est très difficile de raisonner formellement sur l'exécution du programme global. La conception d'un langage dédié embarqué comme Tom nous a amené à proposer un cadre formel pour exprimer la sémantique de combinaison de langages. Ce cadre définit cette combinaison d'un point de vue syntaxique et sémantique sous forme d'un ancrage, ce qui permet de déterminer quelles propriétés de l'ancrage sont nécessaires à la préservation de la sémantique du langage embarqué. Actuellement, la prolifération des langages dédiés démontre l'intérêt de ce formalisme appliqué à la certification de leur développement.

Un langage de stratégies adapté à l'analyse de programmes

Durant cette thèse, nous avons proposé un langage de stratégies adapté à l'expression de propriétés sur un programme. Associé au filtrage, ce langage permet d'exprimer de manière élégante des analyses et des transformations sur des arbres ou des graphes. De plus, un des ses points forts est son intégration dans Java permettant de tirer avantage au mieux des deux paradigmes. Une autre contribution a été de formaliser la sémantique opérationnelle de ce langage et de proposer un encodage des opérateurs de logique CTL. Ainsi, il est possible d'exprimer de manière élégante des optimisations classiques telles que la propagation de constantes ou encore l'*inlining* sous forme de règles de réécriture conditionnelles où les conditions s'expriment par une formule CTL.

Un encodage de la réécriture de termes-graphes dans Java

La représentation de graphes de flot nous a conduit à réfléchir à des constructions de langage inspirées de la réécriture de termes-graphes. Une première contribution a été de représenter les termes-graphes par des termes et d'utiliser la notion de chemin

pour représenter les cycles et les sous-termes partagés, ce qui permet de conserver des structures avec partage maximal. Un deuxième apport a été la définition d'un algorithme original permettant d'exprimer le pas de réécriture de termes-graphes par des opérations de redirection de pointeurs. L'ensemble de ces travaux a été implémenté et intégré dans le langage Tom et représente la seule implémentation connue de réécriture de termes-graphes dans le langage Java.

Validation de ce langage à l'analyse de Java

Durant cette thèse, nous avons développé une bibliothèque permettant de manipuler un programme Bytecode comme un terme algébrique. Le principal avantage de cette approche est de permettre aux utilisateurs de décrire leurs transformations sous forme de règles et ainsi d'améliorer la confiance en leurs programmes. Nous avons donc proposé plusieurs prototypes d'application de ces constructions à l'analyse du langage Java. D'une part, nous avons présenté comment encoder à l'aide de stratégies une analyse telle que la résolution de noms dans le code source Java. D'autre part, nous avons utilisé la bibliothèque permettant de manipuler du Bytecode au travers de deux applications : la compilation à la volée des termes de stratégies et la construction de chargeurs défensifs de classes. Nous avons vu aussi comment les structures de termes-graphes pouvaient être utilisées pour représenter le graphe de flot de contrôle d'un programme.

Perspectives

Les perspectives à court-terme de ce travail sont principalement liées à l'amélioration de l'expressivité du langage proposé dans cette thèse. À long-terme, nous voulons nous donner les moyens d'analyser des langages îlots comme Tom. De part sa nature déclarative, un langage dédié est généralement plus adapté à l'analyse et il serait donc intéressant de développer des techniques d'analyse spécifiques à ce type de langages.

Stratégies et logiques temporelles

Étant donnée l'expressivité du langage de stratégies de Tom, une perspective à court terme serait d'encoder des logiques plus riches comme le μ-calcul ou la logique hybride. Dans le cas de la logique hybride, la réification des positions pourrait permettre de référencer des états afin de pouvoir encoder l'opérateur $@_i$. Une formule $@_i\ p$ est vraie si et seulement si p est vraie à l'unique état identifié par l'étiquette i. Si on associe une position à chacune des étiquettes, cette formule pourrait être vérifiée en testant la position du sous-terme courant.

Nous avons montré dans le chapitre 4 comment encoder les opérateurs de la logique temporelle arborescente sous forme de stratégies. Cet encodage permet d'exprimer de manière plus concise des propriétés sur des arbres. Dans le cas de véritables langages de programmation, il serait intéressant de pouvoir utiliser cet encodage sur des graphes de flot de contrôle. Or tel qu'il est défini, tout cycle entraîne la non-terminaison de la stratégie. Une extension pourrait être de gérer les structures de graphes grâce à la

notion d'environnement du langage SL. L'environnement servirait d'une part à gérer les points-fixes et d'autre part à éviter des calculs inutiles (par *memoization*).

Langage îlot pour les termes-graphes

Dans cette thèse, nous avons proposé plusieurs extensions permettant de réaliser de la réécriture de termes-graphes. L'intégration de ce formalisme n'est pas encore finalisée. Dans l'état actuel, il est possible d'exprimer au même niveau que la signature algébrique un ensemble de règles de réécriture de termes-graphes qui peut par la suite être utilisé dans un programme Tom comme une stratégie élémentaire. La construction *backquote* du langage Tom n'a pour l'instant pas été étendue pour offrir une syntaxe à base d'étiquettes. En effet, une telle intégration nécessite de réfléchir de manière plus globale à l'intégration des termes-graphes au niveau des ancrages du langage Tom. En proposant un nouveau type d'ancrage dédié aux termes-graphes, il serait possible à l'utilisateur de décrire des transformations de termes-graphes directement sur ses propres structures de données. L'extension des signatures algébriques proposée dans cette thèse ne serait alors qu'une instance possible des termes-graphes au même titre que le générateur de structures de termes actuel. Le générateur de termes-graphes offrirait directement à l'utilisateur les ancrages nécessaires à son utilisation dans le langage Tom.

Stratégies dédiées à l'analyse

Nous avons vu dans le chapitre 6 que lorsqu'il est utilisé pour définir des analyses, le langage SL est pénalisé par son manque de souplesse au niveau des types de retour. Si on considère une stratégie comme un moyen de contrôler l'application des règles, il est indispensable que les stratégies préservent le type du terme visité. Cependant, le langage de stratégie s'avère aussi être adapté à la collecte d'informations. Dans ce cas, l'utilisateur n'est pas intéressé par la transformation d'arbres. On pourrait donc imaginer généraliser un peu le langage afin que les combinateurs puissent être utilisés aussi sur des fonctions Java pouvant retourner n'importe quel type de donnée. En s'inspirant des grammaires attribuées, ce type de stratégies associerait des informations aux nœuds de l'arbre abstrait d'un programme et utiliserait le filtrage offert par Tom pour discriminer les sous-arbres tout en préservant la structure de l'arbre abstrait.

Au niveau de l'intégration du langage de stratégies dans le langage Tom, nous avons vu que les stratégies étaient considérées comme des termes. Une stratégie complexe est donc construite comme n'importe quel terme, ce qui ne reflète pas sa nature fonctionnelle. Une amélioration possible pourrait être d'offrir au niveau du langage Tom une syntaxe concrète pour les stratégies basée sur un langage fonctionnel simple. Il serait ainsi possible de déclarer des variables globales partagées par l'ensemble des constructeurs de la stratégie avec une syntaxe à la ML.

Intégration dans le langage Java

Le point fort de ce langage est son intégration dans Java. Or comme Tom a été conçu initialement comme un langage dédié indépendant de son langage hôte, le code envi-

ronnant n'est pas analysé. Les conséquences sont multiples. D'une part, la compilation d'un programme Tom est réalisée en deux phases (la traduction vers Java puis la compilation en Bytecode), ce qui entraîne une gestion des erreurs à deux niveaux. D'autre part, comme certaines constructions du langage Tom sont entrelacées avec du code Java, le système de typage de Tom n'est pas complet puisqu'il suppose que les parties hôtes sont correctement typées. En se fondant sur le compilateur Java extensible proposé par JastAdd, nous avons commencé à travailler sur une extension du système de type de Java prenant en compte les ancrages définis en Tom. Cette extension permet de prendre en compte les types introduits par Tom. Cette information pourra à terme être exploitée par le typeur de Tom. De plus, la gestion des erreurs Tom et Java pourra être réalisée simultanément.

Certification et analyse des îlots

Le concept des îlots formels permet de caractériser formellement les conditions nécessaires à la certification de langages dédiés. Les travaux de thèse d'Antoine Reilles [Rei06b] ont permis de certifier la compilation du filtrage syntaxique. L'approche choisie est celle d'un compilateur certifiant (le code produit est prouvé correct à chaque compilation). L'intérêt est de pouvoir préserver cette certification malgré les évolutions du compilateur et d'autoriser toutes sortes d'optimisations. Durant cette thèse, nous avons poursuivi ses travaux en commençant à certifier la compilation des anti-motifs proposés dans la thèse de Radu Kopetz [Kop08]. Ce prototype se fonde sur l'outil Why [FM07] développé par Jean-Christophe Filliâtre. Why est un générateur d'obligations de preuve. L'une de ses caractéristiques est de s'intégrer à de nombreux prouveurs (assistants de preuve ou outils automatiques). L'utilisation de cet outil facilite grandement la certification d'un langage îlot comme Tom. En effet, le langage intermédiaire de Tom correspond à un sous-ensemble de celui proposé par l'outil Why. Une des perspectives de cette thèse est de continuer le développement du compilateur certifiant de Tom en intégrant de nouvelles constructions (comme le filtrage modulo une théorie ou les stratégies).

Un des intérêts d'un langage dédié est que sa nature déclarative le rend généralement plus adapté à l'analyse. Dans le cas des langages îlots, une perspective serait de proposer un langage d'annotations adaptées. En s'inspirant des annotations proposées dans les langages Jml ou Spec #, l'utilisateur pourrait annoter ses programmes îlots. Par exemple, pour le langage Tom, la logique CTL semble adaptée à la caractérisation des pré- et post-conditions de l'application d'une stratégie sur un terme. Une des caractéristiques d'un langage îlot est la communication entre les environnements d'évaluation du langage hôte et des îlots. Ce type d'annotations devrait donc pouvoir prendre en compte les ancrages pour caractériser par exemple des restrictions sur l'environnement d'évaluation de l'îlot.

Composition de langages dédiés

Dans le cadre de l'analyse de programmes, il existe différentes approches pour exprimer de manière déclarative des analyses ou des transformations. Chacune de ces approches s'avère être plus ou moins adaptée suivant le problème à résoudre. Nous pensons que ce

type de langages est plus intéressant lorsqu'il est embarqué dans un langage généraliste que lorsqu'il est isolé et qu'il pourrait être intéressant de pouvoir combiner plusieurs approches plutôt que de vouloir tout réaliser avec le même langage. Une perspective à plus long terme est de proposer un formalisme permettant d'intégrer et de prototyper facilement des extensions de langage dédiées à l'analyse, en se fondant sur le concept d'ancrages introduits dans cette thèse et largement mis en pratique dans le langage Tom. Ce mécanisme de combinaisons pourrait être étudié dans un cadre tel que les îlots formels ou les sémantiques multi-niveaux.

D'un point de vue pratique, il serait profitable pour le langage Tom (qui définit des structures de données spécifiées par une signature algébrique) d'être combiné avec le langage JastAdd permettant de spécifier des grammaires attribuées. Une autre combinaison intéressante serait l'ajout de constructions de requêtes (*à la* SQL). Par exemple, LINQ est une extension du langage C # qui permet de réaliser des requêtes sur n'importe quelle collection d'objets. Cette extension se fonde pour l'instant sur une interface que les objets visités doivent implémenter. Une perspective pourrait être de proposer le même type d'extension pour Java en se fondant sur les ancrages des îlots formels pour par exemple appliquer des requêtes sur des arbres abstraits Java ou des graphes de flot de contrôle Bytecode.

Bibliographie

[AALL93] Saman P. Amarasinghe, Jennifer M. Anderson, Monica S. Lam et Amy W. Lim – « An overview of a compiler for scalable parallel machines », *LCPC'93 : Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Springer-Verlag, 1993, p. 253–272. 144

[ACH+05] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Oege De Moor, Damien Sereni, Ganesh Sittampalam et Julian Tibble – « abc : An extensible AspectJ compiler », *AOSD'05 : Proceedings of the 4th international conference on Aspect-oriented software development*, ACM Press, 2005, p. 87–98. 143

[AG93] Andrew W. Appel et Marcelo J. R. Gonçalves – « Hash-consing garbage collection », Tech. report, Princeton, 1993. 28

[AK96] Zena M. Ariola et Jan W. Klop – « Equational term graph rewriting », *Fundamenta Informaticae* **26** (1996), no. 3-4, p. 207–240. 7, 107, 116, 126

[AL91] Andrea Asperti et Giuseppe Longo – *Categories, types, and structures : an introduction to category theory for the working computer scientist*, MIT Press, Cambridge, MA, USA, 1991. 105

[Ald04] Jonathan Aldrich – « Open modules : A proposal for modular reasoning in aspect-oriented programming », *FOAL'04 : Workshop on foundations of aspect-oriented languages*, 2004, p. 7–18. 140

[All78] John Allen – *Anatomy of LISP*, McGraw-Hill, Inc., New York, NY, USA, 1978. 82

[AML03] João Cachopo António Menezes Leitão – « re-ProCLessing : Embedding Lisp within Java », *ILC'03 : International Lisp Conference*, 2003. 48

[AR07] Site web du projet ANR RAVAJ – 2007, `http://www.irisa.fr/lande/genet/RAVAJ`. 142

[Ass95] Uwe Assmann – « On Edge Addition Rewrite Systems and Their Relevance to Program Analysis », *5th Workshop on Graph Grammars and Their Application To Computer Science* (Williamsburg, Virginia) (J. Cuny, éd.), Lecture Notes in Computer Science, vol. 1073, Springer-Verlag, November 1994 1995. 112

[Ass00] Uwe Assmann – « Graph rewrite systems for program optimization », *TOPLAS'00 : ACM Transactions on Programming Languages and Systems* **22** (2000), no. 4, p. 583–637. 112

[ASVO05] Tiago Alves, Paulo Silva, Joost Visser et José Nuno Oliveira – « Strategic Term Rewriting And Its Application To a VDM-SL to SQL Conversion », *FM'05 : Formal Methods*, Lecture Notes in Computer Science, vol. 3582, 2005, p. 399–414. 40

[Bar84] Henk P. Barendregt – *The lambda calculus : its syntax and semantics*, North-Holland, Amsterdam, 1984, Second Edition. 6, 15

[BB07] Emilie Balland et Paul Brauner – « Term-graph rewriting in Tom using relative positions », *TERMGRAPH'07 : 4th International Workshop on Computing with Terms and Graphs* (Braga/Portugal), vol. 203, Electronic Notes in Theoretical Computer Science, no. 1, 2007, p. 3–17. 2, 7, 103

[BBGM08] Emilie Balland, Yohan Boichut, Thomas Genet et Pierre-Etienne Moreau – « Efficient implementation of tree automata completion », *AMAST'08 : 13th International Conference of Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science, vol. 5140, Springer-Verlag, 2008, p. 67–82. 142

[BBK+07] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau et Antoine Reilles – « Tom : Piggybacking rewriting on java », *RTA'07 : 18th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science, vol. 4533, Springer-Verlag, 2007, p. 36–47. 1, 5, 16, 18, 20

[BBK+08] Émilie Balland, Paul Brauner, Radu Kopetz, Pierre-Étienne Moreau et Antoine Reilles – « The Tom manual », 2008, `http://tom.loria.fr/soft/release-2.6/manual-2.6/index.html`. 16, 20

[BCE] « Site officiel de BCEL, Byte Code Engineering Library » – `http://jakarta.apache.org/bcel/`. 152

[BEG+87] H. P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer et M.R. Sleep – « Term graph rewriting », *PARLE'87 : Parallel Architectures and Languages Europe* (Eindhoven), Lecture Notes in Computer Science, vol. 259, Springer-Verlag, 1987, p. 141–158. 7, 107

[BGJL07] Y. Boichut, T. Genet, T. Jensen et L. Leroux – « Rewriting Approximations for Fast Prototyping of Static Analyzers », *RTA'07 : Proceedings of the 18th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science, vol. 4533, Springer-Verlag, 2007, p. 48–62. 142

[BHK07] Paul Brauner, Clément Houtmann et Claude Kirchner – « Principles of superdeduction », *LICS'07 : Proceedings of the 22nd Annual IEEE Symposium on Logic In Computer Science* (L. Ong, éd.), IEEE, 2007, p. 41–50. 102

[BHKO02] Mark van den Brand, Jan Heering, Paul Klint et Peter Olivier – « Compiling language definitions : the ASF+SDF compiler », *TOPLAS'02 :*

ACM Transactions on Programming Languages and Systems **24** (2002), no. 4, p. 334–368. 64, 81

[BJ01] Joachim VAN DEN BERG et Bart JACOBS – « The loop compiler for java and jml », *TACAS'01 : Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (London, UK), Springer-Verlag, 2001, p. 299–312. 142

[BJ06] Jean BÉZIVIN et Frédéric JOUAULT – « Applying a model transformation taxonomy to graph transformation technology », *Electronic Notes in Theoretical Computer Science* **152** (2006), p. 69–81. 113

[BJKO00] M.G.J. VAN DEN BRAND, H.A. DE JONG, P. KLINT et P.A. OLIVIER – « Efficient annotated terms », *Software, Practice and Experience* **30** (2000), no. 3, p. 259–291. 66

[BKK+98] Peter BOROVANSKÝ, Claude KIRCHNER, Hélène KIRCHNER, Pierre-Étienne MOREAU et Christophe RINGEISSEN – « An overview of ELAN », *WRLA'98 : Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications* (Pont-à-Mousson, France) (C. KIRCHNER et H. KIRCHNER, éds.), vol. 15, Electronic Notes in Theoretical Computer Science, 1998. 1, 5, 61

[BKM06] Emilie BALLAND, Claude KIRCHNER et Pierre-Etienne MOREAU – « Formal Islands. », *AMAST'06 : 11th International Conference of Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science, vol. 4019, 2006, p. 51–65. 2, 6, 33

[BKV03a] M. BEZEM, J. KLOP et R. DE VRIJER (éds.) – *Term Rewriting Systems.*, Cambridge Tracts in Theoretical Computer Science, vol. I, Cambridge University Press, 2003. 6, 11, 15, 16

[BKV03b] Mark VAN DEN BRAND, Paul KLINT et Jurgen VINJU – « Term rewriting with traversal functions », *TOPLAS'03 : ACM Transactions on Programming Languages and Systems* **12** (2003), no. 2, p. 152–190. 64

[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)*, 2002. 152

[BM06] Emilie BALLAND et Pierre-Etienne MOREAU – « Optimizing pattern matching compilation by program transformation », *SeTra'06 : 3rd Workshop on Software Evolution through Transformations*, vol. 3, Electronic Communications of EASST, 2006. 84

[BM08] Emilie BALLAND et Pierre-Etienne MOREAU – « Term-graph rewriting via explicit paths », *RTA'08 : 19th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science, vol. 5117, Springer-Verlag, 2008, p. 32–47. 2, 7, 54, 103

[BMP06] Guillaume Bonfante, Jean-Yves Marion et Romain Péchoux – « A characterization of alternating log time by first order functional programs. », *LPAR'06 : Proceedings of the 6th International Conference on Logic Programming and Automated Reasoning*, 2006, p. 90–104. 157

[BMR07a] Emilie Balland, Pierre-Etienne Moreau et Antoine Reilles – « Rewriting strategies in Java », *RULE'07 : 8th International Workshop on Rule-Based Programming* (Paris/France), Electronic Notes in Theoretical Computer Science, vol. 219, 2007, p. 97–111. 3, 8, 139

[BMR07b] Emilie Balland, Pierre-Etienne Moreau et Antoine Reilles – « Bytecode rewriting in Tom », *BYTECODE'07 : Second Workshop on Bytecode Semantics, Verification, Analysis and Transformation* (Braga/Portugal), vol. 190, Electronic Notes in Theoretical Computer Science, no. 1, 2007, p. 19–33. 3, 8, 139

[BN98] Franz Baader et Tobias Nipkow – *Term rewriting and all that*, Cambridge University Press, 1998. 11

[Bor98] Peter Borovanský – « Le contrôle de la réécriture : étude et implantation d'un formalisme de stratégies », Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, 1998. 62

[BRL03] L. Burdy, A. Requet et J.-L. Lanet – « Java applet correctness : A developer-oriented approach », *FME'03 : Formal Methods : International Symposium of Formal Methods Europe* (K. Araki, S. Gnesi et D. Mandrioli, éds.), Lecture Notes in Computer Science, vol. 2805, Springer-Verlag, 2003, p. 422–439. 142

[BS02] Claus Brabrand et Michael I. Schwartzbach – « Growing languages with metamorphic syntax macros », *PEPM'02 : Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation* (New York, NY, USA), ACM Press, 2002, p. 31–40. 48

[Bur07] Emir Burak – « Object-oriented pattern matching », Thèse, École Polytechnique Fédérale de Lausanne, 2007. 23

[CC77] Patrick Cousot et Radhia Cousot – « Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints », *POPL'77 : Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (New York, NY, USA), ACM Press, 1977, p. 238–252. 140

[CD97] A. Corradini et F. Drewes – « (Cyclic) term graph rewriting is adequate for rational parallel term rewriting », Tech. Report TR-97-14, Dipartimento di Informatica, Pisa, 1997. 7, 107

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby et Hongjun Zheng – « Bandera : extracting finite-state models from java source code », *ICSE'00 : Proceedings of the 22nd international conference on Software engineering* (New York, NY, USA), ACM Press, 2000, p. 439–448. 141

[CEL99] Manuel Clavel, Steven Eker et Patrick Lincoln – « Maude : specification and programming in rewriting logic », *Theoretical Computer Science* **285** (1999), p. 2002. 1, 5, 63

[CES86] E. M. Clarke, E. A. Emerson et A. P. Sistla – « Automatic verification of finite-state concurrent systems using temporal logic specifications », *TOPLAS'86 : ACM Transactions on Programming Languages and Systems* **8** (1986), no. 2, p. 244–263. 7, 87

[CG95] Andrea Corradini et Fabio Gadducci – « Cpo models for infinite term rewriting », *AMAST'95 : Proceedings of the 4th International Conference in Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science, vol. 936, 1995, p. 368–384. 118

[CHM+02] György Csertan, Gabor Huszerl, Istvan Majzik, Zsigmond Pap, Andréas Pataricza et Daniel Varro – « Viatra : Visual automated transformations for formal verification and validation of uml models », *ASE'02 : Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, IEEE, 2002, p. 267–270. 113

[CM00] Manuel Clavel et José Meseguer – « Reflection and strategies in rewriting logic », *Electronic Notes in Theoretical Computer Science* (J. Meseguer, éd.), vol. 4, Elsevier Science Publishers, 2000. 63

[Cre97] Roger F. Crew – « Astlog : a language for examining abstract syntax trees », *DSL'97 : Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages* (Berkeley, CA, USA), USENIX Association, 1997, p. 18–18. 143

[CRPE08] Kirchner Claude, Kopetz Radu et Moreau Pierre-Etienne – « Anti-pattern matching modulo », *LATA'08 : Proceedings of the 2nd International Conference on Language and Automata Theory and Applications*, Lecture Notes in Computer Science, vol. 5196, Springer-Verlag, 2008, p. 275–286. 25

[DKH97] F. Drewes, H.-J. Kreowski et A. Habel – « Hyperedge replacement graph grammars », p. 95–162, World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997. 104

[DKV00] Arie van Deursen, Paul Klint et Joost Visser – « Domain-specific languages : An annotated bibliography », *SIGPLAN Notices* **35** (2000), no. 6, p. 26–36. 34

[DLL06] Daniel J. Dougherty, Pierre Lescanne et Luigi Liquori – « Addressed term rewriting systems : Application to a typed object calculus », *Mathematical Structures in Computer Science* **16** (2006), p. 667–709. 130

[DR98] Jack Davidson et Norman Ramsey – « The zephyr compiler infrastructure », Internal Report, 1998. 143

[EEKR99] H. Ehrig, G. Engels, H. Kreowski et G. Rozenberg (éds.) – *Handbook on Graph Grammars and Computing by Graph Transformation : Applications, Languages, and Tools*, vol. 2, World Scientific, Singapur, 1999. 113

[EFM00] Conal Elliott, Sigbjorn Finne et Oege de Moor – « Compiling embedded languages », *SAIG'00 :Semantics, Applications, and Implementation of Program Generation*, 2000, p. 9–27. 36

[EH07a] Torbjörn Ekman et Görel Hedin – « The jastadd extensible java compiler », *OOPSLA'07 : Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM Press, 2007, p. 1–18. 24, 58, 86, 144

[EH07b] Torbjörn Ekman et Görel Hedin – « The jastadd extensible java compiler », *SIGPLAN Notices* **42** (2007), no. 10, p. 1–18. 144

[EM98] Andrew Eisenberg et Jim Melton – « Sqlj part 0, now known as sql/olb (object-language bindings) », *SIGMOD Record* **27** (1998), no. 4, p. 94–100. 49

[EP06] Rachid Echahed et Nicolas Peltier – « Narrowing data-structures with pointers. », *ICGT'06 : Proceedings of the Third International Conference on Graph Transformations*, Lecture Notes in Computer Science, vol. 4178, 2006, p. 92–106. 109, 134

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe et Raymie Stata – « Extended static checking for java », *PLDI'02 : Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (New York, NY, USA), ACM Press, 2002, p. 234–245. 142

[FM99] Stephen N. Freund et John C. Mitchell – « A formal framework for the Java bytecode language and verifier », *SIGPLAN Notices* **34** (1999), no. 10, p. 147–166. 140

[FM07] Jean-Christophe Filliâtre et Claude Marché – « The why/krakatoa/caduceus platform for deductive program verification », *CAV'07 : Proceedings of the 19th International Conference on Computer Aided Verification*, vol. 4590, 2007, p. 173–177. 58, 168

[Fow05] Martin Fowler – 2005, `http://martinfowler.com/bliki/FluentInterface.html`. 35

[FUJ] « Fujaba official web site » – `http://www.fujaba.de`. 113

[GBG+06] Rubino Geiss, Gernot Veit Batz, Daniel Grund, Sebastian Hack et Adam Szalkowski – « Grgen : A fast spo based graph rewriting tool », *ICGT'06 : Third International Conference on Graph Transformation*, Lecture Notes in Computer Science, Springer-Verlag, 2006, p. 383–397. 111

[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

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides – *Design patterns : elements of reusable object-oriented software*, Addison-Wesley Professional, 1995. 34

[Gri05] Robert Grimm – « Systems need languages need systems ! », *PLOS'05 :2nd ECOOP Workshop on Programming Languages and Operating Systems*, 2005, unpublished. 144

[GST01] Steven E. Ganz, Amr Sabry et Walid Taha – « Macros as multi-stage computations : Type-safe, generative, binding macros in macroml », *ICFP'01 : International Conference on Functional Programming*, 2001, p. 74–85. 53

[GWDD06] Kris Gybels, Roel Wuyts, Stéphane Ducasse et Maja D'Hondt – « Inter-language reflection : A conceptual model and its implementation », *Computer Languages, Systems & Structures* **32** (2006), no. 2-3, p. 109–124. 6, 54

[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. 60

[Hoa69] C. A. R. Hoare – « An axiomatic basis for computer programming », *Commununications of the ACM* **12** (1969), no. 10, p. 576–580. 52

[HP91] Berthold Hoffmann et Detlef Plump – « Implementing term rewriting by jungle evaluation », *RAIRO'91 : Theoretical Informatics and Applications* **25** (1991), p. 445–472. 7, 107

[HP04] David Hovemeyer et William Pugh – « Finding bugs is easy », *SIGPLAN Notices* **39** (2004), no. 12, p. 92–106. 143

[IP02] Atsushi Igarashi et Benjamin C. Pierce – « On inner classes », *Information and Computation* **177** (2002), no. 1, p. 56–89. 140

[IPW99] Atsushi Igarashi, Benjamin Pierce et Philip Wadler – « Featherweight java : A minimal core calculus for java and gj », *TOPLAS'99 : ACM Transactions on Programming Languages and Systems*, 1999, p. 132–146. 140

[Jon96] Simon L Peyton Jones – « Compiling haskell by program transformation : A report from the trenches », *ESOP'96 : Proceedings of the European Symposium on Programming*, Springer-Verlag, 1996, p. 18–44. 144

[Kac08] Matthieu Kaczmarek – « Des fondements de la virologie informatique vers une immunologie formelle », Thèse de Doctorat d'Université, Institut National Polytechnique de Lorraine, 2008. 137

[Kah87] Gilles Kahn – « Natural semantics », *STACS'87 : 4th Annual Symposium on Theoretical Aspects of Computer Sciences* (London, UK), Springer-Verlag, 1987, p. 22–39. 37

[Kah99] Wolfram Kahl – « The term graph programming system HOPS », *Tool Support for System Specification, Development and Verification* (R. Berghammer et Y. Lakhnech, éds.), Advances in Computing Science, Springer-Verlag, 1999, p. 136–149. 111

[KG89] Sonya E. Keene et Dan Gerson – *Object-oriented programming in Common LISP : a programmer's guide to CLOS*, Addison-Wesley Professional, 1989. 48

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm et William G. Griswold – « An overview of AspectJ », *ECOOP '01 : Proceedings of the 15th European Conference on Object-Oriented Programming*, Springer-Verlag, 2001, p. 327–353. 35, 144

[Kie01] Richard B. Kieburtz – « A logic for rewriting strategies. », *Electronic Notes in Theoretical Computer Science* **58** (2001), no. 2, p. 138–154. 91

[KK99] Claude Kirchner et Hélène Kirchner – « Rewriting, solving, proving », A preliminary version of a book is available at `www.loria.fr/~ckirchne/=rsp/rsp.pdf`, 1999. 123

[KKK08] Claude Kirchner, Florent Kirchner et Hélène Kirchner – « Strategic computations and deductions », To appear in : Festschrift for Peter Andrews, 2008. 6, 15, 16

[KKM07] Claude Kirchner, Radu Kopetz et Pierre-Etienne Moreau – « Anti-pattern matching », *ESOP'07 : Proceedings of European Symposium on Programming*, Lecture Notes in Computer Science, vol. 4421, Springer-Verlag, 2007, p. 110–124. 25

[KKV93] Claude Kirchner, Hélène Kirchner et Marian Vittek – « Implementing computational systems with constraints », *CP'93 : Proceedings First Workshop on Principles and Practice of Constraint Programming* (Brown University, Providence, RI, USA) (P. Kanellakis, J.-L. Lassez et V. Saraswat, éds.), 1993, p. 166–175. 6, 16, 61

[Klo90] J. W. Klop – « Term Rewriting Systems », Handbook of Logic in Computer Science (S. Abramsky, D. Gabbay et T. Maibaum, éds.), vol. 1, Oxford University Press, 1990. 15

[KM01] Hélène Kirchner et Pierre-Etienne Moreau – « Promoting rewriting to a programming language : A compiler for non-deterministic rewrite programs in associative-commutative theories », *Journal of Functional Programming* **11** (2001), no. 2, p. 207–251. 6, 16

[KM08] Radu Kopetz et Pierre-Etienne Moreau – « Software quality improvement via pattern matching », *FASE'08 : Proceedings of the 11th Conference on Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, vol. to appear, Springer-Verlag, 2008. 57, 86

[KMR05a] Ozan Kahramanoğullari, Pierre-Etienne Moreau et Antoine Reilles – « Implementing deep inference in TOM », *SD'05 : Structures and Deduction* (Lisbon, Portugal) (P. Bruscoli, F. Lamarche et C. Stewart, éds.), Technische Universität Dresden, 2005, ISSN 1430-211X, p. 158–172. 157

[KMR05b] Claude Kirchner, Pierre-Etienne Moreau et Antoine Reilles – « Formal validation of pattern matching code », *PPDP'05 : Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of Declarative Programming* (P. Barahone et A. Felty, éds.), ACM Press, 2005, p. 187–197. 6, 57, 58

[Kop08] Radu KOPETZ – « Contraintes d'anti-filtrage et programmation par réécriture », Thèse de Doctorat d'Université, Institut National Polytechnique de Lorraine, 2008. 25, 58, 168

[KOR07] Jan Willem KLOP, Vincent VAN OOSTROM et Femke VAN RAAMSDONK – « Reduction strategies and acyclicity », *Rewriting, Computation and Proof* (H. COMON-LUNDH, C. KIRCHNER et H. KIRCHNER, éds.), vol. 4600, Springer-Verlag, jun 2007. 15

[KV00] Tobias KUIPERS et Joost VISSER – « Object-oriented tree traversal with jjforester », Tech. report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 2000. 81

[KV06] Karl Trygve KALLEBERG et Eelco VISSER – « Strategic graph rewriting : Transforming and traversing terms with references », *WRS'06 : Proceedings of the 6th International Workshop on Reduction Strategies in Rewriting and Programming* (Seattle, Washington), August 2006, Online publication. 110

[Lac03] David LACEY – « Program transformation using temporal logic specifications », Thèse, Oxford University Computing Laboratory, 2003. 59, 83, 92, 102

[Lei06] K. Rustan M. LEINO – « Specifying and verifying programs in spec# », *Ershov Memorial Conference*, Lecture Notes in Computer Science, vol. 4378, 2006, p. 20. 142

[LM01] David LACEY et Oege DE MOOR – « Imperative program transformation by rewriting », *CC'01 : Proceedings of the 10th International Conference on Compiler Construction* (London, UK), Springer-Verlag, 2001, p. 52–68. 7, 87

[Löw93] M. LÖWE – « Algebraic approach to single-pushout graph transformation », *Theoretical Computer Science* **109** (1993), no. 1–2, p. 181–224. 105

[LRR+00] Gary T. LEAVENS, Clyde RUBY, K. RUSTAN, M. LEINO, Erik POLL et Bart JACOBS – « Jml (poster session) : notations and tools supporting detailed design in java », *OOPSLA'00 : Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)* (New York, NY, USA), ACM Press, 2000, p. 105–106. 142

[LV02] Ralf LÄMMEL et Joost VISSER – « Typed Combinators for Generic Traversal », *PADL'02 : Proc. Practical Aspects of Declarative Programming*, Lecture Notes in Computer Science, vol. 2257, Springer-Verlag, 2002, p. 137–154. 62, 66, 144

[LY99] Tim LINDHOLM et Frank YELLIN – *The java(tm) virtual machine specification (2nd edition)*, Prentice Hall PTR, 1999. 150

[ME00] Jim MELTON et Andrew EISENBERG – *Understanding SQL and Java together : A guide to SQLJ, JDBC, and related technologies*, MORGAN-KAUFMANN, 2000. 49

[Mey92] Bertrand Meyer – *Eiffel : the language*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. 142

[MF07] Jacob Matthews et Robert Bruce Findler – « Operational semantics for multi-language programs », *POPL'07 : Proceedings of the 34th Symposium on Principles of Programming Languages*, ACM Press, 2007, p. 3–10. 6, 53

[MHS05] Marjan Mernik, Jan Heering et Anthony M. Sloane – « When and how to develop domain-specific languages », *ACM Computing Surveys* **37** (2005), no. 4, p. 316–344. 35, 36

[MOMV05] Narciso Martí-Oliet, José Meseguer et Alberto Verdejo – « Towards a strategy language for Maude », *WRLA'04 : Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications, Barcelona, Spain* (N. Martí-Oliet, éd.), Electronic Notes in Theoretical Computer Science, vol. 117, Elsevier Science Publishers, 2005, p. 417–441. 6

[MPMU04] C. Marche, C. Paulin-Mohring et X. Urbain – « The krakatoa tool for certification of java/javacard programs annotated in jml », *Journal of Logic and Algebraic Programming* (2004), p. 89–106. 142

[MRV03] Pierre-Etienne Moreau, Christophe Ringeissen et Marian Vittek – « A Pattern Matching Compiler for Multiple Target Languages », *CC'03 : 12th Conference on Compiler Construction, Warsaw, Poland* (G. Hedin, éd.), Lecture Notes in Computer Science, vol. 2622, Springer-Verlag, 2003, p. 61–76. 19, 20

[MSV+07] Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco et Julian Tibble – « .ql : Object-oriented queries made easy », *GTTSE'07 : Generative and Transformational Techniques in Software Engineering II, International Summer School, Braga, Portugal, July 2-7, 2007. Revised Papers*, Lecture Notes in Computer Science, vol. 5235, 2007, p. 78–133. 145

[NCM03] Nathaniel Nystrom, Michael R. Clarkson et Andrew C. Myers – « Polyglot : An extensible compiler framework for java », *CC'03 : In 12th International Conference on Compiler Construction*, Springer-Verlag, 2003, p. 138–152. 143

[Ngu02] Quang-Huy Nguyen – « Rewriting calculus and automation of proofs in proof assistants », Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, 2002. 80

[NO98] Tobias Nipkow et David Von Oheimb – « JavaLight is type-safe — definitely », *POPL'98 : In Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, ACM Press, 1998, p. 161–170. 35

[OAC+06] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman et Matthias Zenger – « An Overview of the Scala Programming Language (2. edition) », Technical report, École Polythecnique Fédérale de Lausanne, 2006. 23

[OW97] M. Odersky et P. Wadler – « Pizza into Java : Translating theory into practice », *POPL'97 : Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, ACM Press, 1997, p. 146–159. 23

[Pic07] Robert Pickering – *Foundations of f#*, Apress, Berkely, CA, USA, 2007. 23

[Plu99] D. Plump – « Handbook of graph grammars and computing by graph transformation », vol. 2, ch. Term graph rewriting, p. 3–61, World Scientific Publishing, 1999. 7, 107

[PMD] « Site officiel de l'outil pmd » – `http://pmd.sourceforge.net`. 143

[Pnu77] Amir Pnueli – « The temporal logic of programs », *FSCS'77 : 18th Annual Symposium on Foundations of Computer Science*, IEEE, 1977, p. 46–57. 87

[RE98] Willem-Paul de Roever et Kai Engelhardt – *Data refinement : Model-oriented proof methods and their comparison*, Cambridge Tracts in Theoretical Computer Science, no. 47, Cambridge University Press, Cambridge, UK, 1998. 40

[Rei06a] Antoine Reilles – « Canonical abstract syntax trees », *WRLA'06 : Proceedings of the 6th International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science, 2006. 28, 77, 81

[Rei06b] Antoine Reilles – « Réécriture et compilation de confiance », Thèse de Doctorat d'Université, Institut National Polytechnique de Lorraine, 2006. 168

[Ren03] Arend Rensink – « GROOVE : A graph transformation tool set for the simulation and analysis of graph grammars », Available at `http ://www.cs.utwente.nl/~groove`, 2003. 112

[Roz97] G. Rozenberg (éd.) – « Handbook of graph grammars and computing by graph transformation », ch. Algebraic Approach to Graph Transformation Part II : Single Pushout Approach and Comparison with Double Pushout Approach, p. 247–312, World Scientific Publishing, 1997. 105

[RW02] A. Rudys et D. Wallach – « Enforcing java run-time properties using bytecode rewriting », *In Proceedings of the International Symposium on Software Security. Tokyo, Japan*, Lecture Notes in Computer Science, 2002. 157

[SA98] Raymie Stata et Martín Abadi – « A type system for Java bytecode subroutines », *POPL'98 : 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California* (New York, NY), 1998, p. 149–160. 140

[SBP99] Tim Sheard, Zine el-abidine Benaissa et Emir Pasalic – « Dsl implementation using staging and monads », *PLAN '99 : Proceedings of the 2nd conference on Domain-specific languages* (New York, NY, USA), ACM Press, 1999, p. 81–94. 36

[SEM08] Max Schäfer, Torbjörn Ekman et Oege de Moor – « Sound and Extensible Renaming for Java », *OOPSLA'08 : 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (G. Kiczales, éd.), ACM Press, 2008. 8, 147, 150

[SEP73] H. J. Schneider, H. Ehrig et M. Pfender – « Graph-grammars - an algebraic approach », *Proceedings of the Fourteenth Annual Symposium on Switching and Automata Theory* (IEEE, éd.), Iowa, 1973, p. 167–180. 105

[SNM07] Don Syme, Gregory Neverov et James Margetson – « Extensible pattern matching via a lightweight language extension », *SIGPLAN Notices* **42** (2007), no. 9, p. 29–40. 23

[Spi01] Diomidis Spinellis – « Notable design patterns for domain-specific languages », *Journal of Systems and Software* **56** (2001), no. 1, p. 91–99. 34, 35, 44

[Spo05] F. Spoto – « Julia : A generic static analyser for the java bytecode », *FTfJP'05 : Proc. of the 7th Workshop on Formal Techniques for Java-like Programs* (Glasgow, Scotland), 2005, Available at `www.sci.univr.it/~spoto/papers.html`. 141

[SWZ95] A. Schürr, A. Winter et A. Zündorf – « Visual programming with graph rewriting systems », *VL'95 : In 11th IEEE Symp. on Visual Languages*, IEEE, 1995, p. 326–335. 111

[Tae99] Gabriele Taentzer – « Agg : A tool environment for algebraic graph transformation », *AGTIVE'99 : International Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance*, Lecture Notes in Computer Science, Springer-Verlag, 1999, p. 481–488. 111

[Tah99] Walid Taha – « Multi-stage programming : Its theory and applications », Thèse, Oregon Graduate Institute of Science and Technology, 1999. 6, 53

[VBT98] Eelco Visser, Zine-el-Abidine Benaissa et Andrew Tolmach – « Building program optimizers with rewriting strategies », *ICFP'98 : Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, ACM Press, Septembre 1998, p. 13–26. 1, 5, 6, 16, 62, 63

[Vis01] Joost Visser – « Visitor combination and traversal control », *OOPSLA'01 : Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (New York, NY, USA), ACM Press, 2001, p. 270–282. 66

[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. 61

[VPM06] Mathieu Verbaere, Arnaud Payement et Oege de Moor – « Scripting refactorings with *jungl* », *OOPSLA'06 : Companion to the 21th ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications* (New York, NY, USA), ACM Press, 2006, p. 651–652. 145

[Wad87] Philip Wadler – « Views : a way for pattern matching to cohabit with data abstraction », *POPL'87 : Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (New York, NY, USA), ACM Press, 1987, p. 307–313. 23

[Wad90] Philip Wadler – « Comprehending monads », *LFP '90 : Proceedings of the 1990 ACM conference on LISP and functional programming* (New York, NY, USA), ACM Press, 1990, p. 61–78. 36

[Yel99] Phillip M. Yelland – « A compositional account of the java virtual machine », *POPL'99 : Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA), ACM Press, 1999, p. 57–69. 140

Table des figures

Résumé

Développer des analyseurs statiques nécessite une manipulation intensive de structures d'arbres et de graphes représentant le programme. Même si la plupart des langages généralistes tels que Java ou C++ disposent de bibliothèques dédiées à la manipulation de telles structures, l'absence d'instruction spécialisée rend le code complexe et difficile à maintenir. La finalité de cette thèse est de proposer des constructions de langage dédiées au prototypage d'outils d'analyse et de transformation de programmes et inspirées de la réécriture de termes et de termes-graphes. L'originalité de notre approche est d'embarquer ces nouvelles constructions dans les langages généralistes sous la forme d'un langage dédié embarqué. Les travaux de cette thèse se fondent sur le langage Tom qui propose d'embarquer des constructions de réécriture dans des langages généralistes comme Java.

La première contribution de cette thèse a été de formaliser les langages embarqués sous le concept de langage îlot. Ce formalisme a ainsi permis de certifier la compilation du langage Tom. Nos travaux sur l'analyse de Bytecode nous ont ensuite conduit à réfléchir à la représentation et la manipulation de graphes de flot de programmes et nous avons alors proposé des constructions de langage inspirées de la réécriture de termes-graphes. Une autre contribution de cette thèse est la conception d'un langage de stratégies adapté à l'expression de propriétés sur un programme. Associé au filtrage, ce langage permet d'exprimer de manière déclarative des analyses et des transformations sur des arbres ou des graphes. Enfin, l'ensemble des propositions de cette thèse a été intégré au langage Tom sous la forme de nouvelles constructions syntaxiques ou d'améliorations de constructions existantes et a ainsi pu être appliqué à l'analyse du langage Java.

Mots clefs : langages de programmation, langages embarqués, analyse statique, transformation de programmes, réécriture, stratégies, termes-graphes

Abstract

Developing static analyzers requires an intensive handling of tree and graph structures representing the program. Even if generalist languages such as Java or C++ have libraries dedicated to the manipulation of such structures, the absence of specialized statements makes the code complex and difficult to maintain. The purpose of this thesis is to provide dedicated language constructs to prototype tools for analysis and program transformation inspired by the term and term-graph rewriting. The originality of our approach is to embed these new statements in generalist languages. This is motivated by the development of the Tom language that offers rewriting constructs for generalist languages like Java.

The first contribution of this thesis is to formalize embedded languages in the concept of island languages. This formalism enables the certification of the Tom compiler. Our work on Bytecode analysis leads us to propose a dedicated language for the representation and manipulation of program flow graphs. Thus we propose language constructs based on the term-graph rewriting. A further contribution of this thesis is to design a strategy language adapted to the expression of properties on a program. Associated with matching capabilities, this language allows to express in a declarative way analysis and transformations on trees or graphs. Finally, all the proposals of this thesis have been integrated into the Tom language in the form of new statements or improvements of existing ones. This language proposal has been applied to the analysis of Java programs.

Keywords: Programming languages, embedded languages, static analysis, program transformation, rewriting, strategies, term-graphs