Contraintes d'anti-filtrage et programmation par réécriture

Radu Kopetz

To cite this version:

Radu Kopetz. Contraintes d'anti-filtrage et programmation par réécriture. Génie logiciel [cs.SE]. Institut National Polytechnique de Lorraine - INPL, 2008. Français. NNT : 2008INPL045N . tel-01748690v2

HAL Id: tel-01748690
https://theses.hal.science/tel-01748690v2

Submitted on 7 Nov 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Institut National
Polytechnique de Lorraine

Département de formation doctorale en informatique

École doctorale IAEM Lorraine

Contraintes d'anti-filtrage et programmation par réécriture

THÈSE

présentée et soutenue publiquement le 15 octobre 2008

pour l'obtention du

Doctorat de l'Institut National Polytechnique de Lorraine
(spécialité informatique)

par

Radu Kopetz

Composition du jury

Rapporteurs :	Mark van den Brand	Professeur, Eindhoven University of Technology, Pays-Bas
	Denis Lugiez	Professeur, Université de Provence, Marseille
Examinateurs :	Yann Le Biannic	Architecte Principal, Business Objects, an SAP company, Paris
	Claire Gardent	Directrice de Recherche, CNRS, Nancy
	Claude Kirchner	Directeur de Recherche, INRIA, Bordeaux
	Michel Leconte	Architecte Principal, ILOG SA, Paris
	Jean-Yves Marion	Professeur, École des Mines de Nancy, INPL
	Pierre-Etienne Moreau	Chargé de Recherche, INRIA, Nancy

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

Mis en page avec LaTeX

Table des matières

Introduction **1**

1. Contexte et motivations **7**

1.1. Le langage TOM . . . 7
- 1.1.1. Signatures algébriques . . . 7
- 1.1.2. Filtrage dans JAVA . . . 8
- 1.1.3. Ancrage formel . . . 11
- 1.1.4. Fonctionnement global du système . . . 12
- 1.1.5. Stratégies : notion de contrôle . . . 13
- 1.1.6. Extensions du langage . . . 14

1.2. Les améliorations traitées dans cette thèse . . . 16
- 1.2.1. Des membres gauches plus expressifs . . . 17
- 1.2.2. Compilation modulaire . . . 17
- 1.2.3. Filtrage sur des structures complexes JAVA . . . 18

2. Notions et concepts élémentaires **19**

2.1. Réécriture . . . 19
- 2.1.1. Termes . . . 19
- 2.1.2. Substitutions et sémantiques closes . . . 20
- 2.1.3. Règles et systèmes de réécriture . . . 21
- 2.1.4. Terminaison et confluence . . . 22
- 2.1.5. Réécriture conditionnelle . . . 23

2.2. Théories équationnelles . . . 23
- 2.2.1. Exemples de théories équationnelles . . . 24
- 2.2.2. Filtrage et réécriture modulo une théorie . . . 25

3. Anti-patterns **27**

3.1. Motivation . . . 27

3.2. Anti-patterns in practice . . . 28
- 3.2.1. Enhancing pattern-matching capabilities of TOM . . . 29
- 3.2.2. Negative conditions in security policies . . . 32
- 3.2.3. Anti-patterns in firewall rules . . . 32

3.3. Anti-terms and their semantics . . . 33
- 3.3.1. Syntax and substitutions . . . 33
- 3.3.2. Semantics: syntactic anti-terms . . . 34
- 3.3.3. Semantics: anti-terms modulo . . . 35
- 3.3.4. Matching syntactic anti-patterns . . . 37

3.3.5. Matching anti-patterns modulo 37
3.4. Solving syntactic and $\mathcal{AU}$ pattern matching 39
3.4.1. Syntactic matching 39
3.4.2. Associative matching 39
3.5. Solving equational anti-pattern matching 45
3.5.1. From anti-pattern matching to equational problems 46
3.5.2. Solving syntactic anti-pattern matching via disunification 48
3.5.3. More tailored approaches 52
3.6. Compiling anti-patterns 56
3.7. Related work 59
3.7.1. Syntactic anti-patterns 59
3.7.2. $\mathcal{AU}$ anti-patterns 61
3.8. Conclusion 61
3.8.1. Synthesis 61
3.8.2. Future work 62

4. Contraintes non-atomiques comme membre gauche des règles 63
4.1. Motivation 63
4.1.1. Le filtrage classique 63
4.1.2. Limitations de l'approche actuelle 64
4.1.3. Les règles de production 65
4.1.4. Un filtrage plus expressif 66
4.2. Nouveau filtrage de Tom 66
4.2.1. Syntaxe 66
4.2.2. Les restrictions 67
4.2.3. L'exécution des actions 69
4.3. Synthèse 70
4.3.1. Plus d'expressivité et d'efficacité 70
4.3.2. Les difficultés 73
4.3.3. Perspectives 73

5. Compiling constraints with pluggable rewrite systems 75
5.1. Several approaches for compiling pattern matching 75
5.1.1. One-to-one pattern matching 75
5.1.2. Decision trees 76
5.1.3. Backtracking automata 76
5.1.4. Tom: one-to-one matching + optimization 77
5.2. Presentation of the Tom system 78
5.2.1. Global view of the system 78
5.2.2. Existing compiler: a brief description 80
5.3. An approach based on rewriting and constraints 85
5.3.1. General architecture of the compiler 86
5.3.2. Input constraints 88
5.3.3. Decomposition and propagation 88

5.3.4. Scheduling 92
5.3.5. Pre-treatment of conditions 93
5.3.6. Code generation 94
5.4. Evaluation and future work 96
5.4.1. Flexibility et legibility 96
5.4.2. Performance analysis 96
5.4.3. Future work 98

6. Filtrage sur des structures complexes Java **99**
6.1. Motivation 99
6.2. Approches possibles 100
6.3. Extraction automatique des informations structurelles d'une hiérarchie JAVA 101
6.3.1. La correspondance entre les classes JAVA et les *mappings* 102
6.3.2. La génération des *mappings* par réflexivité 106
6.4. Bilan et perspectives 108

Conclusion **111**
Contributions 111
Perspectives 112

Bibliographie **115**

Introduction

L'objectif principal de cette thèse est l'étude et la formalisation de nouvelles constructions permettant d'augmenter l'expressivité du filtrage et des langages à base de règles en général. Ceci est motivé par le développement de TOM*, un système qui enrichit les langages impératifs comme* JAVA *et* C *avec des constructions de haut niveau comme le filtrage et les stratégies.*

Une première extension que l'on propose est la notion d'anti-patterns, i.e. des motifs qui peuvent contenir des symboles de complément. La négation est intrinsèque au raisonnement habituel, et la plupart du temps quand on cherche quelque chose, on base nos motifs sur des conditions positives et négatives. Cela doit naturellement se retrouver dans les logiciels permettant les recherches à base de motifs. Par exemple, les anti-patterns permettent de spécifier qu'on cherche des voitures blanches qui ne sont pas des monospaces, ou qu'on cherche une liste d'objets qui ne contient pas deux éléments identiques. Nous définissons alors de manière formelle la sémantique des anti-patterns dans le cas syntaxique, i.e. quand les symboles n'ont aucune théorie associée, et aussi modulo une théorie équationnelle arbitraire. Puis nous étendons la notion classique de filtrage entre les motifs et les termes clos au filtrage entre les anti-patterns et les termes clos (anti-filtrage).

S'inspirant de l'expressivité des règles de production, nous proposons plusieurs extensions aux constructions de filtrage fournies par TOM*. Par conséquent, la condition pour l'application d'une règle n'est plus une simple contrainte de filtrage, mais une combinaison (conjonction ou disjonction) de contraintes de filtrage et d'anti-filtrage ainsi que d'autres types de conditions.*

Les techniques classiques de compilation du filtrage ne sont pas bien adaptées à ces conditions complexes. Ceci a motivé l'étude d'une nouvelle méthode de compilation basée sur des systèmes de réécriture. L'application de ces systèmes est contrôlée par des stratégies, permettant la mise en place d'extensions futures (comme la prise en compte de nouvelles théories de filtrage) de manière simple et naturelle, sans interférer avec le code existant. Nous avons complètement réécrit le compilateur de TOM *en utilisant cette technique.*

Une fois tous ces éléments rassemblés, on obtient un environnement pour décrire et implémenter des transformations de manière élégante et concise. Pour promouvoir son utilisation dans des projets complexes du milieu industriel, nous développons une technique pour extraire de manière automatique des informations structurelles à partir d'une hiérarchie arbitraire de classes JAVA*. Cela permet l'intégration du filtrage offert par* TOM *dans n'importe quelle application* JAVA*, nouvelle ou déjà existante.*

Contexte

La notion de réécriture est omniprésente en informatique et en logique mathématique. En effet, le concept de réécriture apparaît dès les fondements théoriques jusqu'aux réalisations logicielles. La réécriture est utilisée pour définir la sémantique opérationnelle de langages de programmation [Kah87] aussi bien que pour décrire la transformation de programmes [vdBvDK+96]. La réécriture est utilisée pour calculer [Der85], implicitement ou explicitement comme dans Mathematica [Wol99] ou OBJ [GKK+87], mais également lorsqu'on décrit par des règles d'inférence un prouveur de théorèmes [JK86a] ou un solveur de contraintes [JK91]. La réécriture est naturellement très importante dans les systèmes où la notion de règle est un objet explicite du premier ordre, comme les systèmes experts ou de règles de production (JRules [ILO08], Drools [JBo08], Jess [Lab08]).

Le filtrage est une des notions centrales de la réécriture. Il permet, étant donné un terme (appelé *sujet*), de sélectionner parmi un ensemble de règles celles qui peuvent être appliquées. En plus de son utilisation dans les systèmes basés sur la réécriture, comme par exemple ASF+SDF [vdBHdJ+01], ELAN [Vit94, Mor99], Maude [CDE+03, CDE+07], le filtrage est aussi un citoyen de première classe pour les langages fonctionnels comme F# [SGC08], Haskell [has08], OCaml [Ler08]. Plus récemment, il a été considéré comme un ajout pour les langages orientés objet [OW97, Bur07], permettant l'inspection et la décomposition de données de manière concise et élégante.

Dans ce contexte, le but principal de cette thèse est d'étudier des nouvelles constructions et formalismes permettant d'augmenter l'expressivité du filtrage et des langages à base de règles en général. Elle a été principalement motivée par le développement du système Tom [MRV03, BBK+07].

Le langage Tom

Tom est une extension des langages impératifs existants, comme Java ou C. Son but est d'enrichir ces langages en ajoutant des constructions de filtrage inspirées par la réécriture et la programmation fonctionnelle. En plus du filtrage, Tom permet aussi la définition de signatures algébriques. Par exemple, la définition de l'addition sur les entiers de Peano peut se définir de la manière suivante :

```
Nat = zero()
    | suc( s:Nat )
```

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

Dans cet exemple, étant donnés deux termes `t1` et `t2`, représentant des entiers de Peano, l'évaluation de la fonction `plus` retourne la somme de `t1` et `t2`. Ce calcul est implémenté par filtrage : la variable `x` filtre `t1` et les motifs `zero()` et `suc(y)` filtrent éventuellement `t2`. Lorsque le motif `zero()` filtre `t2`, le résultat de l'évaluation de la

fonction `plus` est `x`, qui est instancié par `t1` par filtrage. Lorsque le motif `suc(y)` filtre `t2`, cela signifie que le terme `t2` a `suc` comme symbole de tête ; le sous-terme `y` est alors ajouté à `x`, et le successeur de ce terme est retourné. L'expression de cette fonction `plus` est donnée d'une manière fonctionnelle, mais définit une fonction JAVA, qui peut alors être utilisée de manière classique.

Une caractéristique de TOM est de pouvoir s'intégrer dans plusieurs langages hôte, grâce aux *mappings*. Les *mappings* sont de petits morceaux de code qui décrivent à TOM quelle est la correspondance entre les termes algébriques et leur implémentation au niveau du langage hôte. Cela permet au compilateur de traduire les constructions de filtrage qui sont exprimées sur un type de données algébrique dans des manipulations du type de données concret qui représente ces termes algébriques. Ainsi, à la fin du processus de compilation, toutes les constructions TOM sont transformées dans des instructions du langage hôte.

La construction **%match** permet d'analyser la structure des termes mais aussi d'encoder des règles de transformation. Pour contrôler l'application de ces règles, TOM fournit un langage de stratégies flexible et puissant, inspiré par ELAN [BKK+98], Stratego [VBT98], et JJTraveler [Vis01]. Dans ce langage, les stratégies de haut niveau sont définies en combinant des primitives de bas niveau. Ainsi, le contrôle de l'application des règles est dissocié de règles elles-mêmes, permettant de raisonner séparément sur les règles de transformation et la façon d'effectuer ces transformations.

Nous présentons plus en détail le langage TOM dans le chapitre 1.

Contributions

Vers plus d'expressivité

Une première extension que l'on propose est la notion d'anti-patterns, *i.e.* des motifs qui peuvent contenir des symboles de complément. La négation est intrinsèque au raisonnement habituel, et la plupart de temps quand on cherche quelque chose, on base nos requêtes sur des conditions positives et négatives. Cela doit naturellement se retrouver dans les logiciels permettant les recherches à base de motifs. Par exemple, les anti-patterns permettent de spécifier qu'on cherche des voitures blanches qui ne sont pas de monospaces, ou qu'on cherche une liste d'objets qui ne contient pas deux éléments identiques.

Nous définissons alors de manière formelle la sémantique des anti-patterns dans le cas syntaxique, *i.e.* quand les symboles n'ont aucune théorie associée, et aussi modulo une théorie équationnelle arbitraire. Par la suite, nous étendons la notion classique de filtrage entre les motifs et les termes clos pour permettre le filtrage entre les anti-patterns et les termes clos. Pour résoudre ce type de problèmes, nous proposons plusieurs algorithmes à base de règles.

Les anti-patterns sont des notions générales qui peuvent être utilisées dans la plupart des langages qui intègrent la notion de filtrage, comme par exemple ASF+SDF, ELAN, F#, HASKELL, MAUDE, OCAML, SCALA, *etc.* Nous les avons complètement intégrés dans

TOM, aussi bien dans le cas syntaxique que pour des théories de filtrage plus complexes comme l'associativité avec l'élément neutre.

Les travaux sur les anti-patterns, présentés dans le chapitre 3, ont fait le sujet de publications [KKM07a, KKM07b, KKM08].

Le filtrage de TOM, tel qu'il a été présenté dans la section précédente, permet d'encoder des motifs non-linéaires, mais n'est pas assez général pour encoder tout type de motif avec garde ou des conditions plus complexes combinant conjonctions et disjonctions de conditions de filtrage. L'utilisateur est souvent obligé de recourir à des constructions du langage hôte, ou d'introduire des **%match** imbriqués, qui rendent le code difficile à lire et à maintenir. Pour contourner ces problèmes, nous proposons dans le chapitre 4 des extensions inspirées principalement par les règles de production, permettant de combiner des conditions de filtrage et des prédicats.

Les résultats de ces travaux sont d'ordre pratique. Ils permettent de mettre toutes les conditions pour l'application d'une règle dans la partie gauche, ce qui donne un code plus clair et plus concis. La sémantique du code, son intention, ne sont plus cachés par le mélange entre les instructions JAVA du type *if-then-else* et les constructions **%match** imbriquées.

Programmation par réécriture

Le filtrage de TOM, bien que proche de celui habituellement utilisé par des langages fonctionnels ou ceux basés sur la réécriture, a plusieurs particularités. D'une part il est assez expressif (filtrage syntaxique, filtrage associatif avec élément neutre, des négations illimitées dans les motifs, des conditions non-atomiques dans les membres gauches de règles, *etc*) et d'autre part il est intégré dans un langage hôte sur lequel TOM n'a aucune information — aucune supposition ne peut être faite au sujet de la disponibilité des instructions spécifiques (comme le *switch* ou le *jump*) et en plus les structures de données sont définies par l'utilisateur par l'intermédiaire des *mappings*. Tous ces aspects rendent peu convenables les techniques classiques de compilation du filtrage.

Nous proposons dans le chapitre 5 une nouvelle méthode de compilation pour les combinaisons complexes de conditions de filtrage. C'est une méthode basée sur des systèmes de réécriture, dont l'application est contrôlé par des stratégies. Chaque système de réécriture est un *plug-in* qui s'intègre à une plate-forme. De cette façon, on obtient un environnement de compilation modulaire et extensible, où chaque système de réécriture est en charge du traitement d'un seul type de contrainte (contrainte de filtrage syntaxique, contrainte de filtrage associatif, *etc*). Un autre avantage important de cette approche est la lisibilité du code décrivant le compilateur. Le travail d'un compilateur est essentiellement un travail de transformation, et l'implémenter explicitement avec des règles de transformation est par conséquent la manière la plus naturelle de le faire. Le résultat est un code très proche de sa spécification et implicitement facile à comprendre et à maintenir.

Filtrage ubiquitaire

Une fois tous ces éléments rassemblés, on obtient un environnement pour décrire et implémenter des transformations de manière élégante et concise. Pour promouvoir son utilisation dans des projets complexes du milieu industriel, nous proposons dans le chapitre 6 une technique permettant d'extraire de manière automatique des informations structurelles à partir d'une hiérarchie arbitraire de classes Java. Cela permettra l'intégration du filtrage offert par Tom dans n'importe quelle application Java, nouvelle ou déjà existante, avec peu d'effort de la part de l'utilisateur.

Le résultat est la possibilité de simplifier le code en remplaçant par filtrage la majorité des constructions difficiles à lire et à maintenir habituellement utilisées, comme par exemple les instructions *if-then-else* imbriquées [KM08].

Plan de la thèse

On présente tout d'abord le contexte et les motivations de ce travail dans le chapitre 1. Le chapitre 2 rappelle quelques notions préliminaires sur la réécriture et les théories équationnelles. Les anti-patterns sont présentés dans le chapitre 3, ainsi que leur mise en œuvre au sein du langage Tom. Le chapitre 4 propose plusieurs extensions au filtrage de Tom, permettant de combiner des conditions de filtrage, d'anti-filtrage et des prédicats. Pour pouvoir intégrer toutes ces nouvelles extensions dans un langage de programmation, nous proposons dans le chapitre 5 une nouvelle méthode de compilation pour les combinaisons complexes de conditions de filtrage. Tom permet de décrire et implémenter des transformations de manière élégante et concise, et nous présentons dans le chapitre 6 une technique facilitant son intégration dans des projets complexes du milieu industriel. Enfin, la conclusion dresse un bilan et donne des directions dans lesquelles ce travail devrait être poursuivi.

1. Contexte et motivations

Cette thèse se place dans le cadre du développement du langage Tom. Ainsi, ce langage constitue non seulement le contexte des travaux réalisées, mais aussi le point de départ pour les motivations des contributions présentées.

Nous présentons dans la première partie de ce chapitre le langage Tom de manière informelle, au travers d'exemples. Nous allons nous concentrer plutôt sur les aspects qui seront utiles pour la suite de ce manuscrit. Une présentation complète du langage se trouve dans le manuel de référence [BBK+08]. Les idées de base du langage sont décrites par les auteurs du tout premier compilateur Tom dans [MRV03].

Une fois le contexte établi, nous présentons par la suite les différentes améliorations proposées et traitées dans cette thèse.

1.1. Le langage Tom

Un DSL (*domain-specific language*) fournit une notation adaptée à un domaine d'application en se basant sur les concepts et les caractéristiques de ce domaine. Lorsque les abstractions et les notations sont correctement choisies, un programme écrit dans un DSL est automatiquement plus concis et plus lisible qu'un programme équivalent dans un langage généraliste. Cela permet de minimiser le temps de développement et de maintenance.

Le langage Tom est un DSL qui permet l'encodage de systèmes de réécriture ainsi que la description de transformations sur des arbres. Il est un DSL embarqué dans des langages généralistes, comme Java ou C. Il enrichit ces langages en ajoutant des constructions de filtrage inspirées de la réécriture et de la programmation fonctionnelle. Ce concept est appelé « îlot formel » (*formal island*) [BKM06].

Le principal avantage de l'approche « îlot formel » par rapport à un langage à part entière (*standalone*) est de pouvoir profiter des implémentations de structures de données, des entrées/sorties, des bibliothèques graphiques, des interfaces natives, *etc*, sans effort. Utiliser un langage existant comme langage hôte facilite aussi l'intégration avec d'autres projets développés dans un contexte industriel ou dans le milieu académique. De plus, les programmeurs disposent de nouvelles constructions dans leur environnement de travail courant et bénéficient des preuves et des vérifications pouvant être appliquées sur ces constructions plus abstraites.

1.1.1. Signatures algébriques

Comme la plupart des langages à base de règles ou des langages fonctionnels, Tom fournit à l'utilisateur la possibilité de définir des signatures algébriques multi-sortées,

ceci par le biais du langage Gom [Rei06a].

Gom est un langage permettant de décrire de manière concise la structure d'arbre de syntaxe abstraite, et de générer pour ceux-ci une implémentation dans le langage Java. Cette implémentation offre du typage fort, garantissant que les objets créés sont conformes à la signature multi-sortée, ainsi que du partage maximal, rendant ces structures très efficaces en temps (tests d'égalité en temps constant) et en espace.

Exemple 1. Considérons comme exemple la définition d'une signature Gom pour les entiers Peano :

```
module Peano
  Nat = zero()
      | suc( s:Nat )
```

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

Le *backquote* « ‘ »

La construction « ‘ » (*backquote*) permet de construire un terme algébrique. Ainsi, pour le type algébrique `Nat` défini auparavant, le *backquote* permet de construire des objets représentant des termes. L'instruction

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

déclare une variable `trois` dont le type est `Nat`, ayant pour valeur le représentant du terme algébrique $suc(suc(suc(zero())))$.

Une expression construite avec « ‘ » peut évidemment contenir des variables du langage hôte, ainsi que des appels de fonctions. La portée du « ‘ » correspond à un terme bien formé. Par exemple, le terme `quatre` peut être construit de la façon suivante :

```
Nat quatre = ‘suc(trois);
```

1.1.2. Filtrage dans Java

Le filtrage est une notion centrale dans les langages fonctionnels ainsi que dans les langages basés sur la réécriture. Il permet 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.

Les langages impératifs habituels, comme Java ou C par exemple, ne comportent pas de notion de filtrage. Cette fonctionnalité est habituellement obtenue en utilisant différentes combinaisons d'instructions *switch/case*, *if*-s imbriqués, boucles, *etc*, qui rendent le code illisible et difficile à maintenir.

Le langage Tom permet d'apporter le filtrage dans les langages Java et C avec la construction **%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 sur laquelle on filtre.

Un **%match** est paramétré par une liste de sujets (*i.e.* expressions évaluées en des termes clos) et contient une liste de règles. Les côtés gauches des règles sont des motifs construits à partir de constructeurs et de variables fraîches, sans aucune restriction de linéarité. Les côtés droits ne sont pas des termes, mais des instructions Java qui sont exécutées quand les motifs correspondants filtrent les sujets. Comme dans le cas de la construction standard *switch/case*, les motifs sont évalués de haut en bas. Contrairement au *match* fonctionnel, plusieurs actions (*i.e.* côtés droits) peuvent être exécutées pour un sujet donné tant qu'aucun `return` ou `break` n'est exécuté.

Exemple 2. Considérant la signature algébrique présenté dans l'exemple 1 page ci-contre, l'addition sur les entiers de Peano peut se définir en Tom et Java de la façon suivante :

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

Dans cet exemple, étant donnés deux termes `t1` et `t2`, représentant des entiers de Peano, l'évaluation de la fonction `plus` retourne la somme de `t1` et `t2`. Ce calcul est implémenté par filtrage : la variable `x` filtre `t1` et les motifs `zero()` et `suc(y)` filtrent éventuellement `t2`. Lorsque le motif `zero()` filtre `t2`, le résultat de l'évaluation de la fonction `plus` est `x`, qui est instancié par `t1` par filtrage. Lorsque le motif `suc(y)` filtre `t2`, cela signifie que le terme `t2` a `suc` comme symbole de tête ; le sous-terme `y` est alors ajouté à `x`, et le successeur de ce terme est retourné. L'expression de cette fonction `plus` est donnée d'une manière fonctionnelle, mais définit une fonction Java, qui peut alors être utilisée de manière classique.

Cet exemple montre comment « ` » peut être utilisé pour construire les termes de manière algébrique. Notons que les variables `x` et `y` instanciées par le filtrage peuvent être utilisées. Il est également possible d'utiliser la fonction `plus` comme un constructeur algébrique de la sorte `Nat`, effectuant ainsi l'appel récursif.

Filtrage associatif

En plus des constructeurs libres (qui n'ont aucune théorie associée), des opérateurs représentant des listes peuvent être également déclarés. Ils sont une variante des opérateurs associatifs avec élément neutre :

```
module Peano
  Nat = zero()
```

```
    | suc( s:Nat )

NatList = conc( Nat* )
```

La notation `Nat*` signifie que `conc` est un opérateur variadique (*i.e.* qui prend un nombre arbitraire d'arguments) dont chaque sous-terme est de type `Nat`. Il peut être vu comme un opérateur de concaténation sur des listes de `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 (`zero()`, `zero()` et `suc(zero())`). La liste vide est considérée comme l'élément neutre des listes.

Les opérateurs de liste peuvent être utilisés dans la partie gauche d'une règle, ce qui est appelé *filtrage de liste*, ou *filtrage associatif avec élément neutre*. Ce type de filtrage n'est pas unitaire, et permet d'exprimer simplement des algorithmes manipulant des listes. Le système Tom introduit une distinction syntaxique entre les variables de filtrage représentant des éléments, comme la variable `x` de l'exemple 2 page précédente et les variables représentant une sous-liste d'une liste existante, qui sont suivies de « `*` ».

Exemple 3. Le filtrage de liste permet d'itérer sur les éléments d'une liste lorsque l'action associée au motif utilisant ce filtrage de liste n'interrompt pas le flot de contrôle. De cette façon, on peut définir l'affichage (dans un ordre à priori arbitraire) de tous les éléments d'une liste par :

```
public void printElements(NatList l) {
  %match(l) {
    conc(X1*,x,X2*) -> {
      System.out.println("Element : " + `x);
    }
  }
}
```

L'action associée au motif défini dans cette construction **%match** est exécutée pour chaque filtre trouvé, en instanciant les variables de liste `X1*` et `X2*` par toutes les sous-listes préfixes et suffixes de la liste `l`. Notons que `X1*` correspond à un objet Java de type `NatList`.

Ainsi, cette fonction exécutée sur l'entrée `` l = `conc(zero(), suc(zero()), zero(), suc(suc(zero()))) `` produit en sortie[1] :

```
Element : zero()
Element : suc(zero())
Element : zero()
Element : suc(suc(zero()))
```

Comme dans le cas des motifs syntaxiques, aucune restriction n'est imposée sur la linéarité des motifs associatifs. Aussi bien les variables correspondant à des éléments que

[1] en pratique l'implémentation énumère les éléments de la gauche vers la droite

les variables correspondant à des sous listes peuvent apparaître plusieurs fois dans le motif. Ainsi, le motif `conc(X*,X*)` filtre une liste pouvant être coupée en deux parties identiques, comme `conc(zero(),suc(zero()),zero(), suc(zero()))` — en instanciant `X*` avec `conc(zero(),suc(zero()))`. En revanche, la liste `conc(zero(),suc(zero()), zero(),zero())` n'est pas filtrée par `conc(X*,X*)`.

Exemple 4. Une fonction éliminant les doublons dans une liste est exprimée de manière naturelle par filtrage de liste non-linéaire :

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

1.1.3. Ancrage formel

Jusqu'à présent, nous avons utilisé Gom pour définir la signature des termes utilisés dans les constructions **%match**. Il faut savoir que Tom peut filtrer sur n'importe quel objet du langage hôte, à condition que certaines informations sur la structure de cet objet soient fournies. De manière plus générale, le filtrage est compilé indépendamment de l'implémentation concrète des termes. Ainsi, les constructions de filtrage sont exprimées sur un type de données algébrique, et le compilateur les traduit dans des manipulations du type de données concret qui représente ces termes algébriques. C'est à l'utilisateur du langage de spécifier le type algébrique des termes manipulés, le type de données concret qui le représente, ainsi que comment ces objets concrets représentent les termes algébriques, par l'intermédiaire d'un ancrage. Cet ancrage est appelé *ancrage formel* ou *mapping*. Lorsque Gom est utilisé pour définir les signature algébriques, tous les *mappings* nécessaires sont générés automatiquement.

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

Nous allons illustrer ces constructions par la définition du *mapping* entre le type algébrique `Nat` et son implémentation par les classes Java `NatJ`, `ZeroJ` et `SucJ` décrites ci-dessous :

```
class NatJ { }
class ZeroJ extends NatJ { }
class SucJ extends NatJ {
  NatJ s;
  SucJ( NatJ s ) {
```

```
    this.s = s;
  }
}
```

Tout d'abord, nous devons définir la sorte algébrique Nat, ainsi que spécifier sa correspondance avec les objets concrets qui représentent les termes de cette sorte (*i.e.* NatJ). Ceci est fait par la construction **%typeterm**, qui spécifie en plus comment le test d'égalité entre deux termes de la sorte doit être effectué, lorsque l'on a deux représentants concrets.

```
%typeterm Nat {
  implement       { NatJ }
  equals(t1, t2)  { t1.equals(t2) }
}
```

Tom n'impose pas de restriction sur les expressions qu'on met dans la partie droite des *mappings*. Dans cet exemple, on a utilisé la fonction equals de la classe Object pour tester si les deux termes sont égaux, mais un autre *mapping* pourrait très bien utiliser une autre fonction de comparaison définie par l'utilisateur.

Ensuite, pour pouvoir filtrer vers des objets vus comme des termes de sorte Nat, on doit définir des *opérateurs* pour cette sorte à l'aide de la construction **%op**.

```
%op Nat zero() {
  is_fsym(t)  { t instanceof ZeroJ }
  make        { new ZeroJ() }
}

%op Nat suc(s:Nat) {
  is_fsym(t)      { t instanceof SucJ }
  get_slot(s,t)   { t.s }
  make(t0)        { new SucJ(t0) }
}
```

La première ligne de chaque construction **%op** définit la signature de l'opérateur algébrique, ainsi que les noms des éventuels sous-termes. La définition d'un opérateur doit aussi spécifier la manière de tester si un objet donné représente un terme dont le symbole de tête est l'opérateur (**%is_fsym**), et la manière d'extraire les différents sous-termes en fonction de leur nom (**%get_slot**). Également, la définition de **%op** spécifie comment construire un objet représentant le terme algébrique.

1.1.4. Fonctionnement global du système

Dans la figure suivante, nous décrivons le fonctionnement global du système Tom. Le scénario d'utilisation le plus commun est le suivant : l'utilisateur écrit une structure de données abstraites (ADT), contenant des sortes et des opérateurs, et utilise le compilateur Gom pour générer d'une part l'implantation en Java de ces données et d'autre

part les *mappings* nécessaires. L'utilisateur écrit un programme en Tom et Java, où sont utilisés les sortes et les opérateurs définis. Le compilateur traduit les constructions Tom et produit le code Java correspondant, en utilisant les *mappings*.

Un deuxième scénario possible consiste à ne pas utiliser Gom, et écrire les *mappings* manuellement.

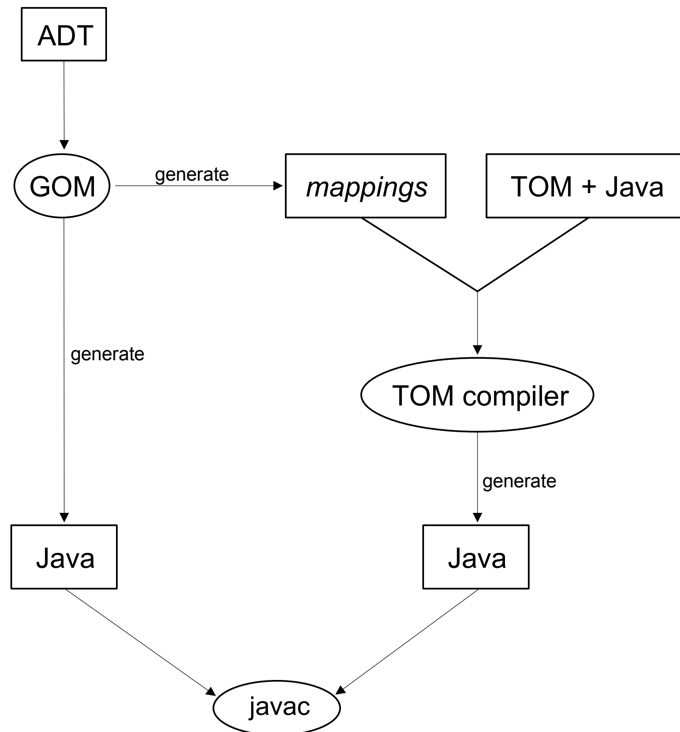


Fig. 1.1.: Fonctionnement global du système Tom

1.1.5. Stratégies : notion de contrôle

La construction **%match** permet d'analyser la structure des termes mais aussi d'encoder des règles de transformation. Pour contrôler l'application de ces règles, une solution est d'utiliser Java. Ainsi, pour effectuer une recherche dans un terme par exemple, une fonction récursive contentant un **%match** peut être employée. Le problème de cette approche, malgré sa souplesse, est la manque d'abstraction qui rend difficile le raisonnement sur les transformations ainsi exprimées. De plus, il est important de dissocier les notions de règles des notions de contrôle, afin de permettre de raisonner séparément sur les règles de transformation et la façon d'effectuer ces transformations.

Pour contourner ces problèmes, Tom fournit un langage de stratégies flexible et puissant, inspiré par ELAN [BKK+98], Stratego [VBT98], et JJTraveler [Vis01]. Dans ce langage, les stratégies de haut niveau sont définies en combinant des primitives élémentaires. De cette façon, Tom fournit à l'utilisateur une variété de stratégies allant des

stratégies comme Identity (ne fait rien), Fail (échoue toujours) ou un ensemble de *règles de réécriture* (qui exécutent un pas élémentaire de réécriture à la racine), aux stratégies plus complexes comme TopDown (applique un système de réécriture de haut en bas sur un terme) ou Innermost (normalise un terme par rapport à un système de réécriture en appliquant les règles sur les expressions réductibles -*redexes*- les plus profondes).

1.1.6. Extensions du langage

Nous avons présenté jusque là les constructions principales de Tom, permettant 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 et la maintenance des programmes complexes, le langage comporte un certain nombre d'extensions. Nous présentons ici les extensions du langage les plus importantes, car elles seront utiles par la suite.

Les variables anonymes

Tom permet l'utilisation d'une notation générique pour les variables dont le nom n'a pas d'importance (par exemple celles qui ne sont pas utilisées dans la partie droite des règles). Cette notation est disponible pour tous les motifs, syntaxique ou associatifs, permettant de simplifier leur écriture et d'améliorer leur lisibilité.

On utilise _ pour dénoter une variable anonyme représentant un terme et _* pour une variable de liste. Par exemple, avec cette notation nous pouvons écrire la fonction d'affichage de l'exemple 3 page 10 de cette façon :

```
public void printElements(NatList l) {
  %match(l) {
    conc(_*,x,_*) -> {
      System.out.println("Element : " + `x);
    }
  }
}
```

La notation implicite

Les signatures algébriques définies pour les opérateurs ont la particularité de donner un nom à chaque argument. Il est alors possible d'utiliser ce nom dans un motif pour désigner un sous-terme particulier ainsi qu'omettre de traiter certains sous-termes, qui sont alors ignorés lors du filtrage, en utilisant la notation [ ] dans les motifs.

Exemple 5. Considérons la signature Gom suivante, permettant d'avoir des termes qui représentent des voitures :

```
module Cars
imports String
```

```
Car = car(interiorColor:String, exteriorColor:String, type:String)
```

Les fonctions suivantes sont équivalentes :

```
public void printType(Car c) {
  %match(c) {
    car(_,_,x) -> {
      System.out.println(‘x);
    }
  }
}
public void printTypeCrochets(Car c) {
  %match(c) {
    car[type=x] -> {
      System.out.println(‘x);
    }
  }
}
```

La seconde écriture a certains avantages par rapport à la première :
- le fait que l'on s'intéresse au *type* dans la fonction est explicite au niveau du motif, ce qui rend la lecture du code plus facile,
- elle est plus robuste au changement : si la signature est changée, par exemple pour ajouter un champ *brand* aux voitures, la seconde fonction ne devra pas être modifiée, alors que la première devra contenir un `_` de plus,
- elle est plus concise dans le cas où les opérateurs contiennent beaucoup d'arguments auxquels on ne s'intéresse pas dans un contexte donné.

Les alias dans les motifs

En utilisant la notation « `@` » dans les motifs de Tom, nous pouvons attribuer un nom aux sous-termes obtenus par filtrage. L'utilisation d'un alias pour un sous-terme donne à une variable une valeur égale à ce sous-terme si le motif filtre, variable qui pourra par la suite être utilisée dans la partie droite de règles.

Exemple 6. Étant donnée une liste d'entiers naturels, la fonction suivante affiche les nombres qui sont supérieurs ou égaux à 2 :

```
public void printElements(NatList l) {
  %match(l) {
    conc(_*,x@suc(suc(_)),_*) -> {
      System.out.println("Element␣>=␣2:␣" + ‘x);
    }
  }
}
```

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

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

1.2. Les améliorations traitées dans cette thèse

Le langage Tom est particulièrement adapté pour encoder des transformations sur des données ayant une structure d'arbre. Il fournit des constructions expressives de filtrage, facilitant la représentation des règles et des systèmes de réécriture. Il offre aussi un langage puissant de stratégies pour contrôler l'application des règles.

Étant donné que Tom est intégré dans des langages généralistes, comme Java et C, il n'y a pas de limite à l'utilisation de Tom. Cependant, l'idéal serait d'avoir la possibilité d'écrire la plupart des parties concernant le filtrage et les transformations en utilisant les constructions de Tom et non celles du langage hôte, ce qui n'est pas valable pour certaines conditions complexes de filtrage. De cette façon, nous pouvons obtenir plus de concision et d'expressivité, et aussi plus de sûreté.

Afin de faire évoluer le langage Tom dans cette direction, nous proposons dans cette thèse une série d'améliorations. Les composants du système concernées par nos travaux sont illustrées dans la figure suivante par les pointillés.

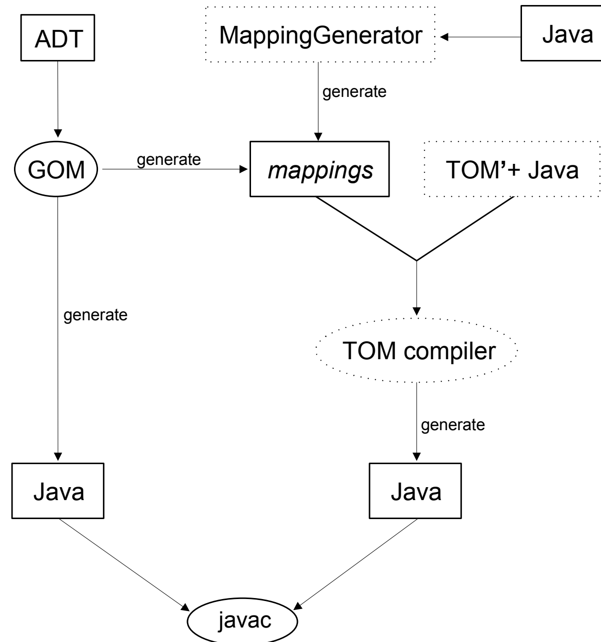


Fig. 1.2.: Aperçu des contributions

1.2.1. Des membres gauches plus expressifs

L'objectif principal de cette thèse est d'augmenter l'expressivité du filtrage et des langages à base de règles. Une première direction étudiée est l'extension des parties gauches des règles afin qu'elles puissent encoder tout type de condition nécessaire à l'application d'une règle.

Dans d'autres langages à base de règles, comme par exemple les systèmes de règles de production (*business rule management systems*), les règles sont exprimées d'une manière beaucoup plus proche du langage naturel. Elle sont généralement de la forme `if` ***`condition`*** `then` ***`action`***, où *condition* peut encoder différentes conditions de filtrage. En s'inspirant de ce formalisme, nous allons chercher les moyens d'intégrer dans TOM les mêmes types de règles. Le but n'est pas d'obtenir un système de règles de production, mais d'utiliser certains de leur concepts dans le monde de la réécriture. Nous introduirons la notion de contrainte de filtrage entre un motif `p` et un terme clos `t`, notée `p << t`, et aussi les connecteurs booléens $\wedge$ et $\vee$ pour encoder respectivement les conjonctions et les disjonctions. De cette façon, des combinaisons complexes de conditions de filtrage peuvent être exprimées de manière concise et élégante.

Le filtrage permet de décomposer les objets et d'analyser leur structure. Une des questions les plus naturelles que l'on se pose dans ce type d'analyses, est comment fait-on pour exprimer ce que l'on ne veut pas ? Autrement dit, y-aurait-il un moyen d'exprimer des conditions négatives sans avoir à les vérifier dans le langage hôte ?

Une première idée serait d'utiliser simplement la négation du filtrage par le biais d'une construction du type `p !<< t`, intuitivement évaluée à vrai si `p << t` est évalué à faux, et à faux dans le cas contraire. Même si utile dans certains cas, une telle construction n'est pas suffisante, car on veut souvent filtrer vers des termes qui *n'ont pas* certains sous-termes, c'est à dire avoir la négation aussi au niveau de sous-termes et pas forcement à la racine. Nous allons donc étudier un moyen d'exprimer les conditions négatives de filtrage en utilisant des symboles de complément ancrés dans les termes, sans aucune restriction sur la position de ces symboles dans les termes ou sur le nombre d'imbrications.

1.2.2. Compilation modulaire

Une fois établies les extensions qui rendraient le filtrage plus expressif, il faut les intégrer au langage. Le problème est que l'architecture initiale du compilateur n'a pas été conçue pour pouvoir être facilement étendue avec des nouvelles constructions. Au moment de sa création, une telle évolution n'a pas été prévue, et par conséquent les choix techniques ont été principalement ciblés vers une implémentation simple et efficace.

L'idéal dans notre vision serait d'avoir un compilateur qui soit une plate-forme basée sur des *plug-ins*, où chaque *plug-in* correspondrait à une fonctionnalité du langage (par exemple un *plug-in* en charge du filtrage syntaxique, un autre pour l'associatif, *etc*). Cette approche modulaire a beaucoup d'avantages, et notamment une grande flexibilité, permettant l'ajout d'une nouvelle extension juste en écrivant un nouveau *plug-in*, sans avoir à modifier le code déjà existant.

1.2.3. Filtrage sur des structures complexes Java

Avec ces extensions, TOM deviendrait encore plus puissant en tant qu'environnement de programmation permettant la définition des transformations de manière claire et concise. Mais son adoption dans les projets du milieu industriel n'est pas très facile, principalement parce que les structures de données utilisées dans ces projets ne peuvent pas, pour la plus part du temps, être définies avec GOM. Cela pour plusieurs raisons : elles existent déjà et un encodage en GOM n'est pas faisable, le modèle offert par GOM ne convient pas pour les définir (par exemple GOM ne supporte pas plusieurs niveaux d'héritage), l'utilisateur a ses propres outils pour leur création (par exemple la génération de classes à partir de modèles UML), *etc.*

Le fait de ne pas pouvoir utiliser GOM implique qu'un *mapping* automatique n'est pas fourni. Une option serait d'écrire ces *mappings* manuellement. Mais ça ce n'est pas une option viable, étant donnée la taille considérable des structures de données dans les projets complexes.

Une variante serait alors de trouver un moyen d'inspecter les données utilisateur et de générer ces *mappings* de manière automatique. Cela permettrait l'intégration du filtrage offert par TOM dans n'importe quelle application, nouvelle ou déjà existante, avec le moindre effort de la part de l'utilisateur.

2. Notions et concepts élémentaires

Nous présentons dans ce chapitre les notions préliminaires utilisées par la suite dans ce document. Nous introduisons notamment la réécriture de termes, puis les théories équationnelles ainsi que la réécriture et le filtrage modulo une théorie équationnelle.

2.1. Réécriture

2.1.1. Termes

Dans cette section nous présentons les notions de base concernant les algèbres de termes du premier ordre. Les termes du premier ordre sont construits à partir de symboles de fonctions et de variables. Par exemple, si f est un symbole de fonction binaire et x, y sont des variables, alors $f(x, y)$ est un terme.

Définition 1 (Signature)**.** *Une signature $\mathcal{F}$ est un ensemble fini de symboles de fonction, dont chacun est associé à un entier naturel qui est appelé son arité. Le sous-ensemble de symboles d'arité n est noté $\mathcal{F}_n$. Les éléments appartenant à $\mathcal{F}_0$ sont appelés des constantes.*

On note d'habitude les constantes par $a, b, c, \ldots$ et les variables par $x, y, z, \ldots$ L'ensemble des variables est noté $\mathcal{X}$. Les symboles de fonction sont souvent appelés *opérateurs.*

Définition 2 (Termes)**.** *Étant donnés un ensemble dénombrable de variables $\mathcal{X}$ et une signature $\mathcal{F}$, on définit l'ensemble des termes $\mathcal{T}(\mathcal{F}, \mathcal{X})$ de manière inductive :*
- *$\mathcal{X} \subset \mathcal{T}(\mathcal{F}, \mathcal{X})$: toute variable est un terme ;*
- *pour tout $n \geq 0$, tous $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, tout opérateur $f \in \mathcal{F}_n$, on a $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.*

Les termes peuvent aussi être définis avec une syntaxe BNF :

Définition 3 (La syntaxe des termes)**.** *Étant donné $\mathcal{F}$ et $\mathcal{X}$, la syntaxe d'un terme est défini par :*

$$\mathcal{T} ::= x \mid f(\mathcal{T}, \ldots, \mathcal{T})$$

où $x \in \mathcal{X}$, $f \in \mathcal{F}$ et l'arité est respectée.

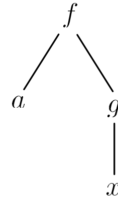
Définition 4 (Variables)**.** *L'ensemble $\mathcal{V}ar(t)$ des variables d'un terme t est défini inductivement par :*
- *$Var(t) = \emptyset$ si $t \in \mathcal{F}_0$,*
- *$Var(t) = \{t\}$ si $t \in \mathcal{X}$,*

– $Var(t) = \bigcup_{i=1}^{n} Var(t_i)$ *si* $f \in \mathcal{F}_n$ *et* $t = f(t_1, \ldots, t_n)$.

Un terme t est dit *linéaire* si toutes les variables apparaissent une seule fois dans t, et *non-linéaire* dans le cas contraire. Par exemple, $f(x, y)$ est un terme linéaire et $f(x, x)$ non-linéaire.

Définition 5 (Terme clos)**.** *Un terme t est clos si $\mathcal{V}ar(t) = \emptyset$. L'ensemble des termes clos est noté $\mathcal{T}(\mathcal{F})$.*

Les termes sont considérés d'habitude comme des arbres, dont les nœuds sont des opérateurs d'arité non nulle et les feuilles sont des variables ou des constantes. Ainsi, le terme $f(a, g(x))$ est représenté par l'arbre :



Définition 6 (Positions)**.** *Une position (ou occurrence) dans un terme t est représentée par une séquence d'entiers positifs ω, décrivant le chemin de la racine du terme jusqu'à la racine du sous-terme à cette position, noté $t_{|\omega}$. Un terme u a une occurrence dans t si $u = t_{|\omega}$ pour une position ω dans t.*

On note $t(\omega)$ le symbole de la racine du $t_{|\omega}$. Par $t[u]_\omega$ on dénote que le terme t contient u comme sous-terme à la position ω. La séquence vide, qui dénote le chemin vide vers la racine est notée Λ.
Les positions sont ordonnées de la manière suivante : $\omega_1 < \omega_2$ si ω_1 est un préfixe du ω_2.
L'ensemble des positions d'un terme t est noté $\mathcal{P}os(t)$.

2.1.2. Substitutions et sémantiques closes

Définition 7 (Substitution)**.** *Une substitution est une application $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F}, \mathcal{X})$, notée lorsque son domaine $\mathsf{Dom}(\sigma)$ est fini $\sigma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$. Cette fonction s'étend de manière unique en un endomorphisme $\sigma' : \mathcal{T}(\mathcal{F}, \mathcal{X}) \to \mathcal{T}(\mathcal{F}, \mathcal{X})$ sur l'algèbre des termes, défini inductivement par :*
– $\sigma'(x) = \sigma(x)$ *pour toute variable* $x \in \mathsf{Dom}(\sigma)$,
– $\sigma'(x) = x$ *pour toute variable* $x \notin \mathsf{Dom}(\sigma)$,
– $\sigma'(f(t_1, \ldots, t_n)) = f(\sigma'(t_1), \ldots, \sigma'(t_n))$ *pour tout symbole de fonction* $f \in \mathcal{F}_n$.

On note Σ l'ensemble de toutes les substitutions.

Définition 8 (Substitution close)**.** *Une substitution σ est appelée* close *si $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$.*

Pour simplifier les notations, on utilise souvent la notation σt à la place de $\sigma(t)$ et on ne fait pas la distinction entre σ et σ'.

Définition 9 (Composition des substitutions). *La composition de deux substitutions σ et θ, notée $\sigma\theta$, est aussi une substitution définie par $\sigma\theta(t) = \sigma(\theta(t))$.*

Définition 10 (Substitution fermante [1]). *Étant donné un terme $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, une substitution σ est appelée* fermante *pour t si $\sigma(t) \in \mathcal{T}(\mathcal{F})$. L'ensemble des substitutions fermantes est noté $\mathcal{G}S(t)$.*

Note A. Une substitution fermante pour un terme t est en général différente d'une substitution close, qui ne dépend pas de t.

Définition 11 (Sémantique close des termes). *La sémantique close d'un terme $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ est l'ensemble de toutes ses instances closes : $[\![t]\!]_g = \{\sigma(t) \mid \sigma \in \mathcal{G}S(t)\}$.*

Exemple 7.

1. $[\![a]\!]_g = \{a\}$, pour a une constante,
2. $[\![x]\!]_g = \mathcal{T}(\mathcal{F})$, pour toute variable x,
3. $[\![f(a, x)]\!]_g = \{f(a, \sigma(x)) \mid \sigma \in \mathcal{G}S(f(a, x))\} = f(a, a), f(a, b), f(a, c), \ldots$

2.1.3. Règles et systèmes de réécriture

L'idée centrale de la réécriture est d'imposer une direction dans l'utilisation des égalités. Une égalité orientée est appelée *règle de réécriture*.

Définition 12 (Règle de réécriture). *Une règle de réécriture pour $\mathcal{T}(\mathcal{F}, \mathcal{X})$ est une paire orientée de termes, notée $l \to r$, où l et r sont des termes de $\mathcal{T}(\mathcal{F}, \mathcal{X})$, appelés respectivement* membre gauche *et* membre droit *de la règle.*

Deux restrictions sont souvent imposées sur une règle de réécriture $l \to r$:
- $l \notin \mathcal{X}$ et
- $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$.

Autrement dit, les parties gauches ne sont pas des variables, et les règles n'introduisent pas des nouvelles variables.

Définition 13 (Système de réécriture). *Un système de réécriture $\mathcal{R}$ construit sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$ est un ensemble (fini où infini) de règles de réécriture pour $\mathcal{T}(\mathcal{F}, \mathcal{X})$.*

Définition 14 (Relation de réécriture). *Étant donné un système de réécriture $\mathcal{R}$ construit sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$, la relation de réécriture associée au système $\mathcal{R}$, notée $\to_{\mathcal{R}}$, est définie par : pour tous termes $t, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, le terme t se réécrit en t', noté $t \to_{\mathcal{R}} t'$, s'il existe :*
- *une règle de réécriture $l \to r$,*
- *une position ω dans t,*
- *une substitution σ,*

tel que $t_{|\omega} = \sigma l$ et $t' = t[\sigma r]_{\omega}$.

Pour une relation binaire $\to$, on note $\xrightarrow{*}$ sa fermeture transitive et réflexive. La fermeture transitive, réflexive et symétrique de $\to$ est noté $\xleftrightarrow{*}$.

[1] *grounding* substitution

Définition 15 (Forme normale)**.** *Soit* $\rightarrow$ *une relation binaire sur un ensemble* T*.*
- *un élément* $t \in T$ *est une* forme normale *si il n'existe pas* $t' \in T$ *tel que* $t \rightarrow t'$*,*
- $t \in T$ *a une forme normale si* $t \xrightarrow{*} t'$ *pour une forme normale* t'*, notée* $t \downarrow$*.*

2.1.4. Terminaison et confluence

La terminaison et la confluence sont les propriétés les plus importantes d'un système de réécriture. La question qu'on se pose est de savoir si pour deux termes t et t', $t \xleftrightarrow{*} t'$. L'idéal serait de calculer une forme normale de chacun et de tester si elles sont égales. Cela n'est possible que si d'une part une forme normale existe pour chaque élément, et si d'autre part elle est unique. Une forme normale existe quand $\rightarrow$ *termine*, et elle est unique quand $\rightarrow$ est de plus *confluente*.

Définition 16 (Terminaison)**.** *Une relation* $\rightarrow$ *sur un ensemble* T termine *si il n'existe pas de suite infinie* $(t_i)_{i \geq 1}$ *d'éléments de* T *telle que* $t_1 \rightarrow t_2 \rightarrow \cdots$*.*

Définition 17 (Confluence)**.** *Étant donnée une relation binaire* $\rightarrow$ *sur un ensemble* T*, la relation* $\rightarrow$ *est dite :*
- avoir la propriété de Church-Rosser *si*

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

- confluente *si*

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

- localement confluente *si*

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

- fortement confluente *si*

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

Si une relation est fortement confluente alors elle est confluente. Si une relation est confluente alors elle est localement confluente. Une relation est confluente si et seulement si elle satisfait la propriété de Church-Rosser.

La confluence est une propriété difficile à vérifier. En pratique, le test de confluence se fait localement grâce au théorème suivant :

Theorem 1 (Newman [New42])**.** *Si* $\rightarrow$ *termine, alors les propriétés suivantes sont équivalentes :*
- $\rightarrow$ *a la propriété de Church-Rosser,*
- $\rightarrow$ *est confluente,*
- $\rightarrow$ *est localement confluente,*
- $\forall t,\ t' \in T : t \xleftrightarrow{*} t' \ \Leftrightarrow\ t \downarrow = t' \downarrow$.

2.1.5. Réécriture conditionnelle

En ajoutant des conditions sur l'application des règles de réécriture, les systèmes de réécriture sont naturellement étendus à des systèmes de réécriture *conditionnels*. Plusieurs définitions des systèmes de réécriture conditionnels ont été proposées, avec comme différence principale l'interprétation des conditions.

Les systèmes de réécriture conditionnels *standards* (standard (join) conditional rewriting system) sont constitués de règles de réécriture de la forme

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

La règle $l \to r$ est appliquée dans le sens de la réécriture non-conditionnelle si avec la substitution σ obtenue pour l'application de la règle, on peut pour tout i entre 1 et n joindre les termes $\sigma(s_i)$ et $\sigma(t_i)$, *i.e.* $\sigma(s_i)$ peut être réduit (en utilisant zéro ou plusieurs réécritures) au même terme que $\sigma(t_i)$.

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

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

dans lesquelles $s \to^! t$ indique que t est une forme normale dérivable à partir de s.

Un système standard contenant des règles de la forme

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

peut être transformé dans un système normal en remplacent les règles par

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

et en ajoutant la règle $eq(x, x) \to true$ au système, ainsi que les symboles eq et $true$. Les réductions des termes ne contenant pas les symboles eq et $true$ sont similaires dans les deux systèmes.

2.2. Théories équationnelles

Une paire de termes (l, r) est appelé *égalité*, *axiome équationnel* ou *axiome égalitaire*, ou *équation* suivant le contexte, et notée $(l = r)$.

Définition 18 (Relation de réduction). *Étant donné un ensemble d'équations $\mathcal{E}$, la relation binaire $\to_{\mathcal{E}} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X}) \times \mathcal{T}(\mathcal{F}, \mathcal{X})$, appelée* relation de réduction*, est définie par $s \to_{\mathcal{E}} t$ si et seulement si il existe une équation $(l = r) \in \mathcal{E}$, une position $\omega \in \mathcal{P}os(s)$, et une substitution σ tels que $s_{|\omega} = \sigma l$ et $t = s[\sigma r]_\omega$.*

La clôture réflexive, symétrique et transitive de $\to_{\mathcal{E}}$, notée $=_{\mathcal{E}}$, est une théorie équationnelle dite *théorie équationnelle engendrée par $\mathcal{E}$* ou plus brièvement, la théorie équationnelle $\mathcal{E}$.

Étant donnée une théorie équationnelle $\mathcal{E}$ et deux ensembles de termes A et B, on a les définitions suivantes :

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

Définition 19 (Présentation d'une théorie équationnelle)**.** *Nous appelons* présentation *d'une théorie équationnelle $\mathcal{E}$ tout ensemble d'axiomes $\mathcal{E}$-égal à $\mathcal{E}$.*

Définition 20 (Théorie syntaxique [Kir86, KK90, Kla92])**.** *Étant donnée une théorie équationnelle $\mathcal{E}$, elle est appelée* syntaxique*, si et seulement si il existe une présentation finie A de la théorie $\mathcal{E}$ tel que toute théorème équationnelle peut être prouvée avec au plus une application d'un axiome de A à la racine.*

Définition 21 (Théorie régulière)**.** *Une théorie équationnelle $\mathcal{E}$ est* régulière *si $s =_{\mathcal{E}} t$ implique $\mathcal{V}ar(s) = \mathcal{V}ar(t)$ pour tous les termes $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.*

2.2.1. Exemples de théories équationnelles

En pratique, des théories équationnelles très utilisées sont l'*associativité* et la *commutativité.*

Définition 22 (Opérateur associatif et commutatif)**.** *Un opérateur binaire $f : \mathcal{T}(\mathcal{F}, \mathcal{X}) \times \mathcal{T}(\mathcal{F}, \mathcal{X}) \rightarrow \mathcal{T}(\mathcal{F}, \mathcal{X})$ est* associatif *si il satisfait*

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

et commutatif *si*

$$\forall x, y \in \mathcal{T}(\mathcal{F}, \mathcal{X}), f(x, y) = f(y, x).$$

Définition 23 (Neutre)**.** *La constante e_f, élément de $\mathcal{T}(\mathcal{F})$, est* neutre à gauche *pour l'opérateur binaire $f : \mathcal{T}(\mathcal{F}, \mathcal{X}) \times \mathcal{T}(\mathcal{F}, \mathcal{X}) \rightarrow \mathcal{T}(\mathcal{F}, \mathcal{X})$ si*

$$\forall x \in \mathcal{T}(\mathcal{F}, \mathcal{X}), f(e_f, x) = x$$

neutre à droite *si*

$$\forall x \in \mathcal{T}(\mathcal{F}, \mathcal{X}), f(x, e_f) = x$$

et élément neutre *pour l'opérateur binaire f si elle est neutre à gauche et à droite pour cet opérateur.*

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

L'associativité est une théorie syntaxique, car toute $\mathcal{A}$-égalité entre deux termes peut être prouvée avec au plus une application de l'axiome de l'associativité à la racine (on peut bien sûr l'appliquer plusieurs fois à des autres positions dans les termes) [Kir86, Kla92].

La forme aplatie des opérateurs associatifs

Un opérateur *variadique* est un symbole de fonction dont l'arité est variable. Il peut prendre un nombre arbitraire d'arguments.

Quand on utilise des opérateurs associatifs, il est courant d'*aplatir* les termes, en remplaçant les occurrences imbriquées de ces opérateurs par des opérateurs variadiques. Par exemple, le terme

$$f(a, f(f(g(a,b), g(b, g(c,d))), f(e,b))),$$

où f, g sont des symboles associatifs, et a, b, c, d, e des constantes, devient

$$f^*(a, g^*(a,b), g^*(b,c,d), e, b),$$

où f^* et g^* sont des opérateurs variadiques [Plo72].

2.2.2. Filtrage et réécriture modulo une théorie

Il existe des égalités que l'on ne peut pas orienter sous peine de perdre la terminaison de la réécriture standard. Un exemple typique est l'axiome de commutativité. Dans ce cas, le raisonnement équationnel nécessite une autre relation de réécriture dans laquelle on travaille sur les classes d'équivalence de termes modulo des égalités non-orientables.

Définition 24 (Réécriture modulo les classes d'équivalence)**.** *Étant donné un système de réécriture $\mathcal{R}$ et un ensemble d'équations $\mathcal{E}$, le terme t se réécrit dans le terme s par $\mathcal{R}$ modulo $\mathcal{E}$, noté $t \rightarrow_{\mathcal{R}/\mathcal{E}} s$, s'il existe une règle $l \rightarrow r \in \mathcal{R}$, un terme u, une position ω dans u et une substitution σ tels que $t =_{\mathcal{E}} u[\sigma l]_\omega$ et $s =_{\mathcal{E}} u[\sigma r]_\omega$.*

Du point de vue calculatoire, la relation $\rightarrow_{\mathcal{R}/\mathcal{E}}$ n'est pas satisfaisante, car pour réécrire un terme, il faut éventuellement parcourir toute sa classe d'équivalence modulo $\mathcal{E}$. Ce parcours est souvent inefficace et même impossible quand les classes d'équivalence modulo $\mathcal{E}$ sont infinies. Une relation plus faible est proposée par Peterson et Stickel [PS81] et plus tard généralisée par Jouannaud et Kirchner [JK86b]. Cette relation, noté $\rightarrow_{\mathcal{R},\mathcal{E}}$, est basée sur le filtrage modulo une théorie équationnelle et est appelée *la réécriture modulo une théorie équationnelle.*

Définition 25 (Réécriture modulo une théorie équationnelle)**.** *Le terme t se réécrit dans le terme s par un système de réécriture $\mathcal{R}$ modulo un ensemble d'axiomes $\mathcal{E}$ s'il existe une règle $l \rightarrow r \in \mathcal{R}$, une position ω dans t et une substitution σ tels que $t_{|\omega} =_{\mathcal{E}} \sigma l$ et $s = t[\sigma r]_\omega$.*

Définition 26 (Équation de filtrage)**.** *Étant donnée une théorie équationnelle $\mathcal{E}$ et deux termes $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, on note $p \ll_{\mathcal{E}} t$ l'équation de filtrage modulo $\mathcal{E}$, appelée aussi $\mathcal{E}$-équation de filtrage (*$\mathcal{E}$-matching equation*), entre ces deux termes.*

Le terme p est habituellement appelé *motif* (*pattern*). Quand $\mathcal{E}$ est la théorie vide, le filtrage entre p et t est appelé *filtrage syntaxique* et noté $p \ll t$.

Définition 27 (Solution d'une équation de filtrage)**.** *Étant donnée une théorie équationnelle $\mathcal{E}$ et deux termes $p, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, une substitution σ est une $\mathcal{E}$-solution pour l'équation de filtrage $p \ll_{\mathcal{E}} t$, appelée aussi un $\mathcal{E}$-*filtre *($\mathcal{E}$-*match*) de p vers t, si et seulement si $\sigma(p) =_{\mathcal{E}} t$.*

Quand il existe une substitution pour une équation de filtrage $p \ll_{\mathcal{E}} t$, on dit que *p filtre t*.

Définition 28 (Système d'équations de filtrage)**.** *Un système d'équations de filtrage ($\mathcal{E}$-*matching system*), noté S, est une conjonction, éventuellement quantifiée, d'équations de filtrage : $\exists\bar{x}, \forall\bar{y}(\wedge_i p_i \ll_{\mathcal{E}} t_i)$[2]. Une substitution σ est solution d'un système S si il existe une substitution ρ de domaine $\bar{x}$, tel que pour toutes les substitutions θ de domaine $\bar{y}$, σ est une solution pour toutes les équations de filtrage $\rho\theta(p_i) \ll_{\mathcal{E}} \rho\theta(t_i)$. L'ensemble de solution de S est noté $\mathcal{S}ol_{\mathcal{E}}(\mathsf{S})$.*

Définition 29 (Disjonction de systèmes d'équations)**.** *Une disjonction de systèmes d'équations de filtrage ($\mathcal{E}$-*matching disjonction*) est notée D. Ses solutions sont les substitutions qui sont des solutions pour au moins un de ses systèmes constituants. Ses variables libres $\mathcal{FV}ar(\mathsf{D})$ sont définies comme habituellement dans la logique des prédicats. La notation $\mathsf{D}[\mathsf{S}]$ dénote que le système S se trouve dans le contexte D.*

[2] $\bar{x}, \bar{y}$ sont des notations courtes pour $x_1, \ldots, x_n$, respectivement $y_1, \ldots, y_m$ avec $n, m \geq 0$.

3. Anti-patterns

When searching for something, we usually base our searches on both positive and negative conditions. Indeed, when stating "except a red car", it means that whatever the other characteristics are, a red car will not be accepted. This is a very common way of thinking, more natural than a series of disjunctions like "a white car or a blue one or a black one or. . . ".

As this is so natural in common reasoning, we fell that complements should be supported in their full generality by languages that offer pattern matching. To this end, we introduce here the notion of *anti-patterns*, as patterns that may contain complement symbols.

3.1. Motivation

Pattern matching is a widely spread concept both in the computer science community and in everyday life. Whenever we search for something, we build a structured object, a pattern, that specifies the features we are interested in. But we are often in the case where we want to exclude certain characteristics: for instance we would like to specify that we search for white cars that are *not* station wagons.

We introduce in this chapter the notion of *anti-patterns*, as patterns that may contain complement symbols, denoted by $\urcorner$. Consider the following situation: we have a search engine for watches, where watches are specified using three properties: case material (steel, gold, plastic, *etc.*), bracelet material (leather, steel, *etc.*) and brand (Rolex, Cartier, *etc.*). In this setting, the anti-pattern $watch(steel, \urcorner leather, Rolex)$ would denote all the Rolex watches that have a steel case and the bracelet is *not* made of leather, while $\urcorner watch(steel, leather, Rolex)$ would select the watches that are *not* Rolexes with steel case and leather bracelet. Things get even more interesting when using nested complement symbols: $\urcorner watch(steel, steel, \urcorner Rolex)$ expresses that we search for a watch that doesn't have both the case and the bracelet made of steel; but in the case the watch is a Rolex, we do not care about the other characteristics. Using disjunctions, this is equivalent with $\urcorner watch(steel, steel, _) \vee watch(_, _, Rolex)$, where the $_$ can be given any value. Although non-linearity inside a negative context is a difficult problem, rarely addressed in current pattern matching based languages, it can be very practical for searching objects that do not have similar sub-characteristics. For instance, the anti-pattern $\urcorner watch(x, x, _)$ would select the watches that do not have the same material for the case and the bracelet. If we also specify the type, $\urcorner watch(x, x, Rolex)$ should be read: no Rolex watch with the same case and bracelet material.

Syntactic anti-patterns (*i.e.* when operators have no particular property, like the ones above) are very useful, but the anti-patterns are even more valuable when associated with

equational theories, in particular with associativity, unit, and eventually with commutativity. For instance, consider the associative matching with neutral element as provided by Tom. The pattern $list(*, \urcorner a, *)$ denotes a list which contains at least one element different from the constant a, whereas $\urcorner list(*, a, *)$ denotes a list which does not contain any a (*list* is an associative operator having the empty list as its neutral element, and $*$ denotes any sublist). By using non-linearity we can express, in a single pattern, list constraints as *AllDiff* or *AllEqual*. Take for instance the pattern $list(*, x, *, x, *)$ that denotes a list with at least two equal elements (x is a variable). The complement of this, $\urcorner list(*, x, *, x, *)$ matches lists that have only distinct elements, *i.e.* *AllDiff*. In a similar way, as $list(*, x, *, \urcorner x, *)$ matches the lists that have at least two distinct elements, its complement $\urcorner list(*, x, *, \urcorner x, *)$ denotes any list whose elements are all equal. Note the use of a non-linear variable in both positive and negative context. More generally, the presented formalism can be used with arbitrary complex patterns, with no restriction. Without anti-patterns, these constructions would have to be expressed as loops, disjunctions, *etc.*

The anti-patterns offer a compact and expressive representation for sets of terms. Their novelty consists in embedding the complements directly into patterns, and filtering against this new construct. This gives a very appropriate and intuitive solution for building a language extension for expressing negative conditions in languages that offer pattern matching facilities, such as most of the functional programming languages (Haskell, ML, OCaml, *etc.*) and rule-based languages (ASF+SDF, ELAN, Maude, Stratego, Tom, *etc.*).

The use of several nested complement symbols can be cumbersome in some cases, especially for new users. But we can easily imagine scenarios where anti-patterns could be generated. When used in a programming language, syntactic sugar can be provided for several constructs that are translated in anti-patterns during the compilation process (like built-in *AllDiff* or *AllEqual* constrains for instance). Another case when they could be generated would be search engines: the user visually choosing some characteristics and excluding others would lead to the creation of complex anti-patterns. For instance, Google has an option where we can specify what words we do *not* want the result pages to contain. This could eventually be extended to "pages that do not contain the words *word1* and *word2*, except if they also contain *word3*" which corresponds to an anti-pattern with two nested $\urcorner$ symbols. The advantage of generating anti-patterns for several constructs would be that the matchings for all these constructs can be later solved (or compiled) in an uniform way, by a single anti-pattern matching algorithm.

Although the main notations were provided in the Chapter 2 on page 19, we assume that the reader is familiar with the standard notions of algebraic rewrite systems, for example presented in [BN98] and rule-based unification algorithms, see *e.g.* [JK91].

3.2. Anti-patterns in practice

The idea of embedding negations in patterns was initially issued from the challenging experimentations made by Luigi Liquori in his implementation of the imperative rewriting

calculus [Liq06], as well as from the need for negative conditions in the Tom language. We took further these ideas and the *anti-patterns* were born.

The most obvious usage of anti-patterns is in a language that provides pattern-matching, like all the functional languages, including Tom, OCaml, Scala, *etc.* Therefore we present in this section the anti-patterns through some examples in Tom, and we finally explore some other possible scenarios where they could be employed.

3.2.1. Enhancing pattern-matching capabilities of Tom

Pattern matching, a central concept in rewriting theory, is extremely useful when implementing applications that perform data transformation or data analysis. Application domains can range from simplification of formulae, query optimizations to program analysis. In this context, expressing negative conditions is a real need, and most of the time is performed with quite illegible ad-hoc solutions, like `if-then-else` instructions for instance. Anti-patterns seem a very proper way to respond to this need, and in this section we present through some examples how they are integrated in the Tom language. In the following we consider Java as the underlying implementation language.

Let us suppose that we have a class `Car`, with the fields `exteriorColor`, `interiorColor` and `type`, all of type `String`. Given an object `subject`, if it is of type `Car`, and its exterior color is `red` we want to print its type. If the type is `hybrid`, print the exterior color:

```
%match(subject) {
  Car("red",_,type)          -> { print(type); }
  Car(extColor,_,"hybrid")   -> { print(extColor); }
}
```

In the above example, `type` and `extColor` are variables. The `_` is an anonymous variable that stands for anything.

In order to support anti-patterns, we enriched the syntax of the Tom patterns to allow the use of operator '!' (representing '$\urcorner$' that will be further introduced). Therefore, now we can write the following patterns:

```
%match(subject) {
  Car(!"red",_,type) -> { print("A car that is not red of type:" + type); }
  !Car("red",_,_)    -> { print("Not a red car"); }
  !Car("red",_,!"hybrid") -> {
     print("Either not a red car or a hybrid one"); }
  !Car(x,x,"hybrid")    -> {
     print("Not a hybrid car with same interior-exterior colors "); }
}
```

We don't impose any restriction on the use of `!` (non-linearity as well as any nesting level of `!` are allowed). Without the use of anti-patterns, one would be forced to verify additional conditions in the action part, which renders the code quite illegible. For example, the previous `%match` should have been written:

```
%match(subject) {
  Car(x,_,type) -> { if (!x.equals("red")) {
                     print("A car that is not red of type:" + type); } }
  x -> { %match(x) {
           Car("red",_,_) -> { break; /* don't do anything */ }
           _  -> { print("Not a red car"); } } }
  y -> { %match(y) {
           Car(_,_,"hybrid") -> { print("hybrid car"); }
           Car("red",_,_)  -> { break; /* don't do anything */ }
           _  -> { print("Not a red car"); } } }
  z -> { %match(z) {
           Car(x,x,"hybrid") -> { break; /* don't do anything */ }
           _  -> { print("Not a hybrid car with same int.-ext. colors "); }
      } }
}
```

Besides matching simple objects, Tom can also match lists of objects. This is a very useful feature, as it is quite often the case when the data that we manipulate is organized in lists. This is true for any query over a database, which returns a list of results, or the XML documents that are actually lists of nodes.

As we will detail in the next sections, matching lists is a special form of matching associative patterns with neutral elements ($\mathcal{AU}$ matching — see [JK91] for a discussion on unification and matching in equational theories)[1]. Let's consider now that `subject` is a list of cars, and we want to check if it contains a red car, and if yes, to print its type:

```
%match(subject) {
  carList(X*,Car("red",_,type),Y*) -> { print(type); }
}
```

`carList` is a variadic list operator, the variables suffixed by `*` are instantiated with lists (possibly empty), and can be used in the action part: here `X*` is instantiated with the beginning of the list up to the matched object, whereas `Y*` contains the tail. The action is executed for each solution of the matching problem (assigning different values to variables). Patterns can be non-linear: `carList(X*,X*)` denotes a list composed of two identical sublists, whereas `carList(X*,x,Y*,x,Z*)` denotes a list that has twice the same element (`x` is a variable that can only be instantiated with a single value, and not with a sublist).

A possible usage of anti-patterns could be to check that a list respects some conditions. For instance, complementing the previous pattern checks that the list doesn't contain a red car:

```
%match(subject) {
  !carList(X*,Car("red",_,type),Y*) -> { print("no red cars"); }
```

[1] lists usually correspond to $\mathcal{AU}$ patterns in a flattened form

```
}
```

Note that the variables in a negative context, such as `X*`, `Y*` and `type` cannot be instantiated and used in the action part (the system will raise an error forbidding this). When we are not interested in the content of a list variable, such as `X*` and `Y*` in the above pattern, we can use anonymous list variables denoted by `_*`.

In a similar way, the pattern `carList(_*,!Car("red",_,_),_*)` checks that the list contains at least one element different from a red car. We can actually extract all these elements from the list with the following code:

```
%match(subject) {
  carList(_*,x@!Car("red",_,_),_*) -> { print("not a red car:" + x); }
}
```

The symbol @ is used to give a name, an alias, to the matched object in order to use it further in the action part. Complementing this last pattern, read *not a list of cars that contains at least one element different from a red car*, leads to checking that the list only contains red cars: `!carList(_*,!Car("red",_,_),_*)`.

As we previously saw, `carList(_*,x,_*,x,_*)` can be used to check that the list contains at least two equal objects, and eventually collect them. Negating this pattern naturally checks that the list only contains different elements: `!carList(_*,x,_*,x,_*)`, *i.e.* the well-known *AllDiff* constraint from constraint programming.

Besides simply checking conditions on a list, we can also extract all the elements from a given list that respect a certain condition (as was previously the case when using the `@` symbol). For example, the following code prints all the cars that appear only once in the list `subject`:

```
%match(subject) {
  carList(_*,x,_*) && !carList(_*,x,_*,x,_*) << subject -> { print(x); }
}
```

Note that the `&&` is the classical boolean connector $\wedge$ and `<<` is the match symbol. More exactly, writing `pattern << subject` has the same meaning as

```
%match(subject) {
  pattern -> ...
}
```

The idea of the above code is that the first pattern selects an element from the list, and the second one verifies that it doesn't appear twice.

More advanced examples may consist in using anti-patterns to obtain the expressivity of regular expressions. For instance, in the list of cars `subject` we may look for the sublist that contains only red cars: `carList(_*,!carList(_*,!Car("red",_,_),_*),_*)`. For retrieving it, we can name it with `@`:

```
carList(_*,mySbl@!carList(_*,!Car("red",_,_),_*),_*)
```

and further use the variable `mySbl` in the right-hand side of the rules.

Another advantage of using anti-patterns (compared to classical matching plus additional conditions in the action part) is that they may improve efficiency, by verifying some conditions earlier in the matching process.

We hope that these examples illustrate clearly enough the anti-patterns' usefulness. Before presenting their formal definition, we will briefly present other possible scenarios where they could be employed.

3.2.2. Negative conditions in security policies

Recently, there have been several approaches to encoding the security policies using term rewriting [DKKdO07, BFB07]. A classical example for security policies is the one that defines the access to the assignment of the grades in an university. It is quite common that a PhD student gives lectures and assigns grades, which makes him both a student and a faculty member. Consider the following natural language rules:

1. *If the subject is a professor or administrative staff, then permit to assign grades,*
2. *If the subject is a student, then permit to assign grades only if the subject is also a faculty member*

Supposing that we only have professors, administrative staff and students, and that in the signature we have *Student* that has a boolean argument specifying if it is or not a faculty member, we might represent this as follows:

$$request(!\,Student(\,!\,isFacultyMember), AssignGrades) \rightarrow permit$$

The above pattern can be read as *either the subject is not a student, or a student that is also a faculty member.* Using only one level of negation, we would write:

$$\begin{array}{ll} request(Student(isFacultyMember), AssignGrades) & \rightarrow permit \\ request(\,!\,Student(x), AssignGrades) & \rightarrow permit \end{array}$$

and if anti-terms were not available, we would have

$$\begin{array}{ll} request(Student(isFacultyMember), AssignGrades) & \rightarrow permit \\ request(Professor(x), AssignGrades) & \rightarrow permit \\ request(Admin(x), AssignGrades) & \rightarrow permit \end{array}$$

Note that in this latter case, the number of rules depends on the number of categories that can assign grades.

3.2.3. Anti-patterns in firewall rules

Firewalls are other examples where rules based on pattern matching are usually employed. The number of rules is very important for manageability. Therefore, when using a lot of them, the possibility to replace two or three rules by only one rule can make a big difference in the legibility of the filtering conditions.

Syntactic anti-patterns could be useful for expressing some of the filtering conditions for instance. Consider that an Internet user wants to implement the following rule in his firewall: *reject all the packets coming from 10.1.1.1, except if the destination is 192.168.1.1.* We can imagine the following rule:

$$filter(pckt(10.1.1.1,\ !\ 192.168.1.1)) \rightarrow drop$$

Assuming that the evaluation of the rules is performed in order, without anti-patterns this rule may be encoded as follows:

$$\begin{array}{ll} filter(pckt(10.1.1.1, 192.168.1.1)) & \rightarrow accept \\ filter(pckt(10.1.1.1, x)) & \rightarrow drop \end{array}$$

3.3. Anti-terms and their semantics

This section introduces the *anti-terms*. We start by presenting their syntax, and we study how substitutions are applied. We further provide the semantics of anti-terms in the simplest case, *i.e.* when the symbols have no theory associated. When anti-terms contain symbols that have an equational theory associated, we call them *anti-terms modulo*, and we detail their semantics.

We extend the classical notions of matching equations, systems and disjunctions by allowing the *left-hand side* of the match equations to be anti-terms. When anti-terms are used in matching equations, we usually call them *anti-patterns*, and we often refer to these equations as *anti-pattern matching equations*. We study their solutions in the case of syntactic anti-patterns, and we then generalize to anti-patterns modulo.

3.3.1. Syntax and substitutions

Definition 30 (Syntax of anti-terms)**.** *Given $\mathcal{F}$ and $\mathcal{X}$, the syntax of an anti-term is defined as follows:*

$$\mathcal{AT} ::= x \ \mid\ \urcorner\mathcal{AT} \ \mid\ f(\mathcal{AT}, \ldots, \mathcal{AT})$$

where $x \in \mathcal{X}$, $f \in \mathcal{F}$ and the arity is respected. The set of anti-terms is denoted $\mathcal{AT}(\mathcal{F}, \mathcal{X})$ (resp. $\mathcal{AT}(\mathcal{F})$ for ground anti-terms). Any term is an anti-term, i.e. $\mathcal{T}(\mathcal{F}, \mathcal{X}) \subseteq \mathcal{AT}(\mathcal{F}, \mathcal{X})$.

For example, if x, y, z denote variables, a, b, c constants, f, g two function symbols of arity 2 and 1, the following expressions are anti-terms: $\urcorner x$, $\urcorner a$, $\urcorner f(\urcorner a, g(\urcorner x))$, $f(x, y)$, $f(\urcorner a, b)$, $f(x, \urcorner x)$.

The symbol $\urcorner$ should be seen as any other function symbol in the structure of an anti-term. We therefore have for a term $f(x, y)(\wedge) = f$, $f(x, y)(1) = x$, and for an anti-term$\urcorner f(x, y)(\wedge) = \urcorner$, $\urcorner f(x, y)(1) = f$.

Definition 31 (Free variables)**.** *The free variables of an anti-term q are defined inductively by:*

1. $\mathcal{FV}ar(x) = \{x\}$,
2. $\mathcal{FV}ar(\neg q) = \emptyset$,
3. $\mathcal{FV}ar(f(q_1, \dots, q_n)) = \cup_{i=1..n}\mathcal{FV}ar(q_i)$, *with the arity of f equal to n.*

Example 8. Assuming that a is a constant and f is binary, we have: $\mathcal{FV}ar(a) = \emptyset$, $\mathcal{FV}ar(\neg x) = \emptyset$, $\mathcal{FV}ar(f(x, \neg y)) = \{x\}$, $\mathcal{FV}ar(f(x, \neg x)) = \{x\}$, $\mathcal{FV}ar(\neg f(x, \neg x)) = \emptyset$.

Definition 32 (Substitutions on anti-terms)**.** *A substitution σ uniquely extends to an endomorphism σ' on $\mathcal{AT}(\mathcal{F}, \mathcal{X})$ whose scope is explicitly restricted to the set of free variables of the anti-term on which it is applied:*

1. $\sigma'(q) = \sigma'_{\mathcal{FV}ar(q)}(q)$ *if q is not a variable,*
2. $\sigma'_\Delta(x) = \sigma(x)$ *for all* $x \in \mathsf{Dom}(\sigma) \cap \Delta$,
3. $\sigma'_\Delta(x) = x$ *for all* $x \notin \mathsf{Dom}(\sigma) \cap \Delta$,
4. $\sigma'_\Delta(f(t_1, \dots, t_n)) = f(\sigma'_\Delta(t_1), \dots, \sigma'_\Delta(t_n))$ *for* $f \in \mathcal{F}$.

In what follows we omit the set Δ whenever is clear from the context.

Example 9. Note that substitutions are active only on the free variables: $\sigma(f(x, \neg x)) = f(\sigma(x), \neg\sigma(x))$, $\sigma(f(x, \neg y)) = f(\sigma(x), \neg y)$.

The notion of grounding substitutions is also extended to anti-terms (e.g. t) as substitutions (e.g. σ) such that $\mathcal{FV}ar(\sigma(t)) = \emptyset$. For instance, $\sigma = \{x \mapsto a\}$ is a grounding substitution for $f(x, \neg y)$.

3.3.2. Semantics: syntactic anti-terms

Intuitively, the semantics of the complement of a term represents the complement of its semantics in $\mathcal{T}(\mathcal{F})$. Therefore, the complement of a variable $\neg x$ denotes $\mathcal{T}(\mathcal{F})\backslash[\![x]\!]_g = \mathcal{T}(\mathcal{F})\backslash\mathcal{T}(\mathcal{F}) = \emptyset$. Similarly, $\neg f(x)$ denotes $\mathcal{T}(\mathcal{F})\backslash\{f(t) \mid t \in \mathcal{T}(\mathcal{F})\}$. In the following we extend this intuition to complements of complements, as well as complements which occur in subterms, and we formally define the semantics of an anti-term.

Definition 33 (Ground semantics of anti-terms)**.** *The ground semantics of any anti-term $q \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$ is defined recursively in the following way:*

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

where z is a fresh variable and for all $\omega' < \omega$, $q(\omega') \neq \neg$.

Example 10.

1. $[\![\neg a]\!]_g = [\![z]\!]_g \backslash [\![a]\!]_g = \mathcal{T}(\mathcal{F})\backslash\{a\}$,

2. $[\![\urcorner x]\!]_g = [\![z]\!]_g \backslash [\![x]\!]_g = \mathcal{T}(\mathcal{F}) \backslash \mathcal{T}(\mathcal{F}) = \emptyset$, for any variable x,

3. $[\![\urcorner\urcorner x]\!]_g = [\![z]\!]_g \backslash [\![\urcorner x]\!]_g = [\![z]\!]_g \backslash ([\![z']\!]_g \backslash [\![x]\!]_g) = \mathcal{T}(\mathcal{F}) \backslash (\mathcal{T}(\mathcal{F}) \backslash \mathcal{T}(\mathcal{F})) = \mathcal{T}(\mathcal{F})$,

4. $[\![\urcorner g(x)]\!]_g = [\![z]\!]_g \backslash [\![g(x)]\!]_g = \mathcal{T}(\mathcal{F}) \backslash \{g(\sigma(x)) \mid \sigma \in \mathcal{GS}(g(x))\}$,

5. $[\![g(\urcorner x)]\!]_g = [\![g(z)]\!]_g \backslash [\![g(x)]\!]_g = \emptyset$,

6. $[\![\urcorner g(\urcorner x)]\!]_g = [\![z]\!]_g \backslash [\![g(\urcorner x)]\!]_g = \mathcal{T}(\mathcal{F}) \backslash \emptyset = \mathcal{T}(\mathcal{F})$,

7. we can also express that we are looking for something that is either not rooted by g, or it is $g(a)$:
$$\begin{aligned} [\![\urcorner g(\urcorner a)]\!]_g &= [\![z]\!]_g \backslash [\![g(\urcorner a)]\!]_g = [\![z]\!]_g \backslash ([\![g(z')]\!]_g \backslash [\![g(a)]\!]_g) \\ &= \mathcal{T}(\mathcal{F}) \backslash ([\![g(z')]\!]_g \backslash \{g(a)\}) \\ &= \mathcal{T}(\mathcal{F}) \backslash (\{g(\sigma(z')) \mid \sigma \in \mathcal{GS}(g(z'))\} \backslash \{g(a)\}) \\ &= \mathcal{T}(\mathcal{F}) \backslash \{g(z) \mid z \in \mathcal{T}(\mathcal{F}, \mathcal{X})\} \ \cup \ \{g(a)\}, \end{aligned}$$

8. $[\![f(a, \urcorner b)]\!]_g \ = [\![f(a,z)]\!]_g \backslash [\![f(a,b)]\!]_g = \{f(a, \sigma(z)) \mid \sigma \in \mathcal{GS}(f(a,z))\} \backslash \{f(a,b)\}$,

9. $[\![\urcorner f(x,x)]\!]_g \ = [\![z]\!]_g \backslash [\![f(x,x)]\!]_g = \mathcal{T}(\mathcal{F}) \backslash \{f(\sigma(x), \sigma(x)) \mid \sigma \in \mathcal{GS}(f(x,x))\}$

   note the crucial use of non-linearity to denote any term except those rooted by f with identical subterms,

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

The second condition of Definition 33 is essential ($\forall \omega' < \omega,\ q(\omega') \neq \urcorner.$). It prevents replacing a subterm by a fresh variable inside a complemented context (i.e. below a $\urcorner$), which would lead to counter-intuitive situations. For instance, without this condition, for $\urcorner g(\urcorner a)$ we would have had $[\![\urcorner g(\urcorner a)]\!]_g = [\![\urcorner g(z)]\!]_g \backslash [\![\urcorner g(a)]\!]_g = \emptyset$.

3.3.3. Semantics: anti-terms modulo

When terms, respectively anti-terms, contain symbols with equational theories associated, their semantics change. We further define terms and anti-terms semantics modulo an arbitrary *regular* equational theory $\mathcal{E}$, that doesn't contain the symbol $\urcorner$.

Definition 34 (Ground semantics of terms modulo)**.** *Given a regular equational theory $\mathcal{E}$ and $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, the ground semantics of t modulo $\mathcal{E}$ is defined as:*

$$[\![t]\!]_{g\mathcal{E}} = \{t' \mid t' \in_{\mathcal{E}} [\![t]\!]_g\}.$$

Therefore, the ground semantics of t modulo $\mathcal{E}$ is the set of all the ground terms that can be computed from the ground semantics of t by applying the axioms of $\mathcal{E}$.

Definition 35 (Ground semantics of anti-terms modulo)**.** *Given $q \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$ and a regular equational theory $\mathcal{E}$, the ground semantics of q modulo $\mathcal{E}$ is defined recursively in the following way:*

$$[\![q[\daleth q']_\omega]\!]_{g\mathcal{E}} = \begin{cases} [\![q[z]_\omega]\!]_{g\mathcal{E}} \backslash [\![q[q']_\omega]\!]_{g\mathcal{E}}, \; if \; \mathcal{FV}ar(q[\daleth q']_\omega) = \emptyset \\ \bigcup_{\sigma \in \mathcal{GS}(q[\daleth q']_\omega)} [\![\sigma(q[\daleth q']_\omega)]\!]_{g\mathcal{E}} \; otherwise \end{cases}$$

where z is a fresh variable and for all $\omega' < \omega$, $q(\omega') \neq \daleth$.

When $\mathcal{E}$ is the empty theory, this definition is compatible with Definition 33. However, in the equational case a direct adaptation of Definition 33 cannot be used. Consider the term $f(x, f(\daleth a, y))$, where f is $\mathcal{AU}$. This intuitively denotes the lists that contain at least one element different from a, like $f(b, f(a, c))$ for instance. Suppose we use Definition 33 to compute the ground semantics, we would get $[\![f(x, f(z, y))]\!]_{g\mathcal{AU}} \backslash [\![f(x, f(a, y))]\!]_{g\mathcal{AU}}$, which does not contain the term $f(b, f(a, c))$. This happens because giving different values to x, y and applying the $\mathcal{AU}$ axioms differently on the two terms, we obtain different term structures in the two sets. But this is not the intuitive semantics of anti-terms. The second case of Definition 35 prevents exactly these types of situations, by first instantiating the variables before computing the ground semantics and thus splitting the term semantics in a set difference. For instance, choosing $\sigma = \{x \mapsto e_f, y \mapsto f(a, c)\}$ and later $\{z \mapsto b\}$, the above example respects the intuition.

Example 11. In the following examples, the symbol f is considered $\mathcal{AU}$:

1. $$\begin{aligned} [\![f(x, f(\daleth a, y)]\!]_{g\mathcal{AU}} &= \bigcup_\sigma [\![f(\sigma(x), f(\daleth a, \sigma(y)))]\!]_{g\mathcal{AU}} \\ &= \bigcup_\sigma ([\![f(\sigma(x), f(z, \sigma(y)))]\!]_{g\mathcal{AU}} \backslash [\![f(\sigma(x), f(a, \sigma(y)))]\!]_{g\mathcal{AU}}) \\ &= f(a, f(b, c)), f(a, f(a, c)), f(b, f(a, c)), \ldots \end{aligned}$$

2. $$\begin{aligned} [\![\daleth f(x, f(\daleth a, y)]\!]_{g\mathcal{AU}} &= [\![z]\!]_{g\mathcal{AU}} \backslash [\![f(x, f(\daleth a, y))]\!]_{g\mathcal{AU}} \\ &= \mathcal{T}(\mathcal{F}) \backslash \bigcup_\sigma [\![f(\sigma(x), f(\daleth a, \sigma(y)))]\!]_{g\mathcal{AU}} \\ &= \mathcal{T}(\mathcal{F}) \backslash \bigcup_\sigma ([\![f(\sigma(x), f(z, \sigma(y)))]\!]_{g\mathcal{AU}} \backslash [\![f(\sigma(x), f(a, \sigma(y)))]\!]_{g\mathcal{AU}}) \\ &= everything \; that \; is \; not \; an \; f \; or \; an \; f \; with \; only \; \mathrm{a} \; inside \end{aligned}$$

3. $$\begin{aligned} [\![\daleth f(x, x)]\!]_{g\mathcal{AU}} &= [\![z]\!]_{g\mathcal{AU}} \backslash [\![f(x, x)]\!]_{g\mathcal{AU}} \\ &= \mathcal{T}(\mathcal{F}) \backslash \{t' \mid t' \in_{\mathcal{AU}} [\![f(x, x)]\!]_g\} \\ &= everything \; that \; is \; not \; an \; f \; or \; an \; f \; that \; is \; not \; symetric \\ &= a, b, f(a, b), f(a, f(b, c)), \ldots \; but \; not \; f(a, a), f(f(a, b), f(a, b)), \ldots \end{aligned}$$

The simple examples in Example 10 and 11 show that anti-terms provide a compact and expressive representation for the sets of terms. A nice property can be easily derived:

Proposition 1. For any $t \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$ and $\mathcal{E}$, we have: $[\![\daleth\daleth t]\!]_{g\mathcal{E}} = [\![t]\!]_{g\mathcal{E}}$.

Proof. Using the Definition 35, we have

$$[\![\neg\neg t]\!]_{g\mathcal{E}} = [\![z]\!]_{g\mathcal{E}} \backslash [\![\neg t]\!]_{g\mathcal{E}} = [\![z]\!]_{g\mathcal{E}} \backslash ([\![z']\!]_{g\mathcal{E}} \backslash [\![t]\!]_{g\mathcal{E}}) = \mathcal{T}(\mathcal{F}) \backslash (\mathcal{T}(\mathcal{F}) \backslash [\![t]\!]_{g\mathcal{E}}) = [\![t]\!]_{g\mathcal{E}}.$$

□

3.3.4. Matching syntactic anti-patterns

When considering syntactic patterns, a matching equation $p \prec\!\!\prec t$ has a solution when there exists a substitution σ such that $\sigma(p) = t$, that is when $t \in [\![p]\!]_g$. But from Definition 11, this can be seen as $\{t\} = [\![\sigma(p)]\!]_g$, for all $\sigma \in \mathcal{GS}(p)$ solution of $p \prec\!\!\prec t$. This extends naturally to the anti-patterns.

Definition 36 (Solutions of anti-pattern matching)**.** *For all $q \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$ and $t \in \mathcal{T}(\mathcal{F})$, the solutions of the anti-pattern matching equation $q \prec\!\!\prec t$ are:*

$$Sol(q \prec\!\!\prec t) = \{\sigma \mid t \in [\![\sigma(q)]\!]_g,\ with\ \sigma \in \mathcal{GS}(q)\}.$$

Remember that by Definition 32, the substitutions apply only on free variables. Also note that for $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, we have $[\![\sigma(p)]\!]_g = \{\sigma(p)\}$ when $\sigma \in \mathcal{GS}(p)$; this is not always true for the anti-patterns. Take for example $f(x, \neg b)$, and $\sigma = \{x \mapsto a\}$: the set $[\![\sigma(f(x, \neg b))]\!]_g = [\![f(a, \neg b))]\!]_g$ has more than one element, as we saw in Example 10. Here are some examples for the solutions of anti-pattern matching equations:

Example 12.

1. $Sol(f(a, \neg b) \prec\!\!\prec f(a, a)) = \Sigma$,
2. $Sol(\neg g(x) \prec\!\!\prec g(a)) = \{\sigma \mid g(a) \in \mathcal{T}(\mathcal{F}) \backslash \{g(\sigma(x)) \mid \sigma \in \mathcal{GS}(g(x))\}\} = \emptyset$,
3. $Sol(f(\neg a, x) \prec\!\!\prec f(b, c)) = \{x \mapsto c\}$,
4. $Sol(f(x, \neg x) \prec\!\!\prec f(a, b)) = \{x \mapsto a\}$,
5. $Sol(f(x, \neg g(x)) \prec\!\!\prec f(a, g(b))) = \{x \mapsto a\}$,
6. $Sol(f(x, \neg g(x)) \prec\!\!\prec f(a, g(a))) = \emptyset$.

3.3.5. Matching anti-patterns modulo

The Definition 36 is extended to matching modulo $\mathcal{E}$ as follows:

Definition 37 (Solutions of anti-pattern matching modulo)**.** *For all $q \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$ and $t \in \mathcal{T}(\mathcal{F})$, the solutions of the anti-pattern matching equation $q \prec\!\!\prec_{\mathcal{E}} t$ are:*

$$Sol(q \prec\!\!\prec_{\mathcal{E}} t) = \{\sigma \mid t \in [\![\sigma(q)]\!]_{g\mathcal{E}},\ with\ \sigma \in \mathcal{GS}(q)\}.$$

The solutions of a matching system $\exists\bar{x}, (\wedge_i q_i \prec\!\!\prec_{\mathcal{E}} t_i)$, where q_i are anti-patterns, are defined as in the case of terms: a substitution σ is an $\mathcal{E}$-solution of this matching system if there exists a substitution ρ, with domain $\bar{x}$, such that σ is solution of all the matching equations $\rho(q_i) \prec\!\!\prec_{\mathcal{E}} \rho(t_i)$. As t_i are ground terms, this is equivalent to σ solution of $\rho(q_i) \prec\!\!\prec_{\mathcal{E}} t_i$, equivalent from Definition 37 to $t \in [\![\sigma\rho(q)]\!]_{g\mathcal{E}}$. We formalize this in the following definition:

Definition 38. *For all $q_i \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$ and $t_i \in \mathcal{T}(\mathcal{F})$, the solutions of the matching system $\exists\bar{x}(\wedge_{i\in I} q_i \prec\!\!\prec_{\mathcal{E}} t_i)$, with $I \neq \emptyset$, are:*

$$Sol(\exists\bar{x}(\wedge_{i\in I} q_i \prec\!\!\prec_{\mathcal{E}} t_i)) = \{\sigma \mid \mathsf{Dom}(\sigma) = \mathcal{FV}ar(q)\backslash\{\overline{x}\} \text{ and } \exists\rho \text{ with } \mathsf{Dom}(\rho) = \overline{x}, \text{ such that for all } i \in I, t_i \in [\![\sigma\rho(q_i)]\!]_{g\mathcal{E}}\}.$$

This definition has the following consequence:

$$Sol(\forall\bar{x}\ not(q_i \prec\!\!\prec_{\mathcal{E}} t_i) = \{\sigma \mid \forall\rho \text{ with } \mathsf{Dom}(\rho) = \overline{x},\ t \notin [\![\sigma\rho(q)]\!]_{g\mathcal{E}}\},$$

with the same conditions on σ as in Definition 38, and *not* being the classical logic negation. But the ground semantics instantiate anyway $\bar{x}$ with all the possible values, therefore the condition $\forall\rho$ can be omitted, which gives us the following:

$$Sol(\forall\bar{x}\ not(q_i \prec\!\!\prec_{\mathcal{E}} t_i) = \{\sigma \mid \mathsf{Dom}(\sigma) = \mathcal{FV}ar(q)\backslash\{\overline{x}\},\ t \notin [\![\sigma(q)]\!]_{g\mathcal{E}}\}.$$

Let us look at several examples of anti-pattern matching modulo in some usual equational theories:

Example 13.
In the associative theory (f is $\mathcal{A}$):
- $Sol(f(x, f(\urcorner a, y)) \prec\!\!\prec_{\mathcal{A}} f(b, f(a, f(c, d))) = \{x \mapsto f(b, a), y \mapsto d\}$,
- $Sol(\exists x f(x, f(\urcorner a, y)) \prec\!\!\prec_{\mathcal{A}} f(b, f(a, f(c, d))) = \{y \mapsto d\}$,
- $Sol(f(x, f(\urcorner a, y)) \prec\!\!\prec_{\mathcal{A}} f(a, f(a, a)) = \emptyset$.

The following patterns express that we do not want an a below an f:
- $Sol(\urcorner f(x, f(a, y)) \prec\!\!\prec_{\mathcal{A}} f(b, f(a, f(c, d))) = \emptyset$,
- $Sol(\urcorner f(x, f(a, y)) \prec\!\!\prec_{\mathcal{A}} f(b, f(b, f(c, d))) = \Sigma$.

A combination of the two previous examples, $\urcorner f(x, f(\urcorner a, y))$, would naturally correspond to *an* f *with only* a *inside*:
- $Sol(\urcorner f(x, f(\urcorner a, y)) \prec\!\!\prec_{\mathcal{A}} f(a, f(b, a)) = \emptyset$,
- $Sol(\urcorner f(x, f(\urcorner a, y)) \prec\!\!\prec_{\mathcal{A}} f(a, f(a, a)) = \Sigma$.

Non-linearity can be also useful:
$Sol(\urcorner f(x, x) \prec\!\!\prec_{\mathcal{A}} f(a, f(b, f(a, b))) = \emptyset$, but $Sol(\urcorner f(x, x) \prec\!\!\prec_{\mathcal{A}} f(a, f(b, f(a, c))) = \Sigma$.
If besides associative, we consider that f is also commutative, then we have the following results for matching modulo $\mathcal{AC}$: $Sol(f(x, f(\urcorner a, y)) \prec\!\!\prec_{\mathcal{AC}} f(a, f(b, c))) = \{\{x \mapsto a, y \mapsto c\}, \{x \mapsto a, y \mapsto b\}, \{x \mapsto b, y \mapsto a\}, \{x \mapsto c, y \mapsto a\}\}$.

3.4. Solving syntactic and $\mathcal{AU}$ pattern matching

In order to provide equational anti-matching algorithms in the next section, we first need to make precise the matching algorithms that serves as our starting point. Therefore, in this section we first recall a matching algorithm in the syntactic case. We further provide a rule-based presentation of an $\mathcal{A}$ matching algorithm, witch we later adapt for $\mathcal{AU}$. To our knowledge, these algorithms are the first explicit presentations of rule-based approaches for solving $\mathcal{A}$ and $\mathcal{AU}$ matching problems.

3.4.1. Syntactic matching

The solution of a matching system S (in the syntactic case, without quantifiers), when it exists, is unique and is computed by a simple recursive algorithm [Hue76]. This algorithm can be expressed by the set of *rewrite rules* Match, given below. The symbol $\wedge$ is assumed to be associative, commutative and idempotent, S is any conjunction of matching equations, p_i are patterns, t_i are ground terms, x stands for any variable and f, g are any symbols from $\mathcal{T}(\mathcal{F}, \mathcal{X})$:

Decompose	$f(p_1, \ldots, p_n) \prec\!\!\prec f(t_1, \ldots, t_n)$	$\mapsto\!\!\!\rightarrow$	$\bigwedge_{i=1..n} p_i \prec\!\!\prec t_i$
SymbolClash	$f(p_1, \ldots, p_n) \prec\!\!\prec g(t_1, \ldots, t_m)$	$\mapsto\!\!\!\rightarrow$	$\perp$ if $f \neq g$
MergingClash	$x \prec\!\!\prec t_1 \wedge x \prec\!\!\prec t_2$	$\mapsto\!\!\!\rightarrow$	$\perp$ if $t_1 \neq t_2$
Delete	$p \prec\!\!\prec p$	$\mapsto\!\!\!\rightarrow$	$\top$
PropagateClash	$\mathsf{S} \wedge \perp$	$\mapsto\!\!\!\rightarrow$	$\perp$
PropagateSuccess	$\mathsf{S} \wedge \top$	$\mapsto\!\!\!\rightarrow$	S

The soundness and the completeness of Match is expressed as follows:

Theorem 2 ([KK99])**.** *The normal form by the rules in* Match *of any matching equation $p \prec\!\!\prec t$ such that $t \in \mathcal{T}(\mathcal{F})$, exists and is unique.*

1. *if it is of the form $\bigwedge_{i \in I} x_i \prec\!\!\prec t_i$ with $I \neq \emptyset$, then the substitution $\sigma = \{x_i \mapsto t_i\}_{i \in I}$ is the unique match from p to t,*
2. *if it is $\top$ then p and t are identical, i.e. $p = t$,*
3. *if it is $\perp$, then there is no match from p to t.*

3.4.2. Associative matching

As opposed to syntactic matching, matching modulo an equational theory is undecidable as well as not unitary in general [Bür90]. When decidable, matching problems can be quite expensive either to decide matchability or to enumerate complete sets of matchers. For instance, matchability is NP-complete for $\mathcal{AU}$ or $\mathcal{AI}$ (idempotency) [BKN87]. Also, counting the number of minimal complete set of matches modulo $\mathcal{A}$ or $\mathcal{AU}$ is #P-complete [HK95].

In this section we focus on the particular useful case of matching modulo $\mathcal{A}$ and $\mathcal{AU}$. The reason why we chose to detail these specific theories are their tremendous usefulness

in rule-based programming such as ASF+SDF [vdBHdJ+01] or MAUDE [Eke92, Eke03] for instance, where lists, and consequently list-matching, are omnipresent. A list-matching problem $p \prec\!\!\prec t$ is a restricted case of $\mathcal{AU}$-matching, where p and t must have the same top symbol [Rei06b].

Since associativity and neutral element are regular axioms (*i.e.* equivalent terms have the same set of variables), we can apply the combination results for matching modulo the union of disjoint regular equational theories [Nip91, Rin96] to get a matching algorithm modulo the theory combination of an arbitrary number of $\mathcal{A}$, $\mathcal{AU}$ as well as free symbols. Therefore we study in this section matching modulo $\mathcal{A}$ or $\mathcal{AU}$ of a single binary symbol f, whose unit is denoted e_f. The only other symbols under consideration are free constants.

Matching associative patterns

By making precise this algorithm, our purpose is to provide a simple and intuitive one that can be easily proved to be correct and complete and that will be later adapted to anti-pattern matching. With the goal of efficiency, other solutions were developed for instance in [Eke92, Eke03].

Unification modulo associativity has been extensively studied [Plo72, Mak77]. It is decidable, but infinitary, while $\mathcal{A}$-matching is finitary. Our matching algorithm $\mathcal{A}$-Match is described in Figure 3.1 and is quite reminiscent from [Nip90] although not based on a Prolog resolution strategy. It strongly relies on the *syntacticness* of the associative theory. The main idea that characterizes a syntactic theory is to be able to decompose a given problem in a *finite* set of equivalent subproblems.

Proposition 2. Given a matching equation $p \prec\!\!\prec_{\mathcal{A}} t$ with $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $t \in \mathcal{T}(\mathcal{F})$, the application of $\mathcal{A}$-Match always terminates.

Proof. As we deal with a matching problem (where the right-hand side is a ground term), we are interested in a derivation of this problem. For this, the size measure we further define is relative to the size of the right-hand side of the initial problem – always bigger or equal to the left-hand side in order for the match to have a solution.

Let D_0 be the initial problem, *i.e.* $D_0 = p \prec\!\!\prec_{\mathcal{A}} t$. Further on, $D_0 \mapsto\!\!\!\!\rightarrow_{\mathcal{A}-\mathsf{Match}} D_1 \mapsto\!\!\!\!\rightarrow_{\mathcal{A}-\mathsf{Match}} \dots \mapsto\!\!\!\!\rightarrow_{\mathcal{A}-\mathsf{Match}} D_n$. For all $i \in [1..n]$, the size of D_i, denoted by $\|D_i\|$, is a multiset computed in the following way:

- $\|D_j \wedge D_k\| = \|D_j \vee D_k\| = \|D_i\| \cup \|D_j\|$,
- $\|\exists z(D_j)\| = \|D_j\|$,
- $\|\bot\| = \|\top\| = \{0\}$,
- $\|p' \prec\!\!\prec_{\mathcal{A}} t'\| = \{\|t'\|\}$.

Further on, the size of terms is defined as:

- $\|f(t_1, t_2)\| = 1 + \|t_1\| + \|t_2\|$,
- $\|a\| = 1$, for a a constant,

Mutate	$f(p_1,p_2) \lll_{\mathcal{A}} f(t_1,t_2)$	$\mapsto\!\!\!\rightarrow$	$(p_1 \lll_{\mathcal{A}} t_1 \wedge p_2 \lll_{\mathcal{A}} t_2) \vee$ $\exists x(p_2 \lll_{\mathcal{A}} f(x,t_2) \wedge f(p_1,x) \lll_{\mathcal{A}} t_1) \vee$ $\exists x(p_1 \lll_{\mathcal{A}} f(t_1,x) \wedge f(x,p_2) \lll_{\mathcal{A}} t_2)$
SymbolClash$_1$	$f(p_1,p_2) \lll_{\mathcal{A}} a$	$\mapsto\!\!\!\rightarrow$	$\bot$
SymbolClash$_2$	$a \lll_{\mathcal{A}} f(p_1,p_2)$	$\mapsto\!\!\!\rightarrow$	$\bot$
ConstantClash	$a \lll_{\mathcal{A}} b$	$\mapsto\!\!\!\rightarrow$	$\bot$ if $a \neq b$
Replacement	$z \lll_{\mathcal{A}} t \wedge \mathsf{S}$	$\mapsto\!\!\!\rightarrow$	$z \lll_{\mathcal{A}} t \wedge \{z \mapsto t\}\mathsf{S}$ if $z \in \mathcal{FV}ar(\mathsf{S})$

Utility Rules:

Delete	$p \lll_{\mathcal{A}} p$	$\mapsto\!\!\!\rightarrow$	$\top$
Exists$_1$	$\exists z(\mathsf{D}[z \lll_{\mathcal{A}} t])$	$\mapsto\!\!\!\rightarrow$	$\mathsf{D}[\top]$ if $z \notin \mathcal{V}ar(\mathsf{D}[\top])$
Exists$_2$	$\exists z(\mathsf{S}_1 \vee \mathsf{S}_2)$	$\mapsto\!\!\!\rightarrow$	$\exists z(\mathsf{S}_1) \vee \exists z(\mathsf{S}_2)$
DistributeAnd	$\mathsf{S}_1 \wedge (\mathsf{S}_2 \vee \mathsf{S}_3)$	$\mapsto\!\!\!\rightarrow$	$(\mathsf{S}_1 \wedge \mathsf{S}_2) \vee (\mathsf{S}_1 \wedge \mathsf{S}_3)$
PropagateClash$_1$	$\mathsf{S} \wedge \bot$	$\mapsto\!\!\!\rightarrow$	$\bot$
PropagateClash$_2$	$\mathsf{S} \vee \bot$	$\mapsto\!\!\!\rightarrow$	S
PropagateSuccess$_1$	$\mathsf{S} \wedge \top$	$\mapsto\!\!\!\rightarrow$	S
PropagateSuccess$_2$	$\mathsf{S} \vee \top$	$\mapsto\!\!\!\rightarrow$	$\top$

Figure 3.1.: $\mathcal{A}$-Match: p_i are patterns, t_i are ground terms, and S is any conjunction of matching equations. Mutate is the most interesting rule, and it is a direct consequence of the fact that associativity is a syntactic theory. $\wedge, \vee$ are classical boolean connectors.

- $\|x\| = \|t\|$, if $x \in \mathcal{V}ar(p)$, *i.e.* a free variable of the initial problem D_0,
- $\|x\| = \|t_j\| - 1$, if $x \notin \mathcal{V}ar(D_i)$ and $D_{i+1} = C[\exists x(C'[p_j \lll_{\mathcal{A}} t_j])]$ with $x \in \mathcal{V}ar(p_j)$ — here C denotes the context. Therefore, each time a new existential variable is introduced, its size is fixed and it remains unchanged afterwards.

Note that when an existential variable is introduced in a left-hand side of an equation, its size is fixed to the size of the right-hand side minus 1. As further applications of the algorithm never increase the right-hand side, when solved, this variable's size can't exceed its fixed size. Moreover, it is instantiated with its size minus 1, as we can observe from the equations of the right-hand side of Mutate: x can only be instantiated in the second equation from $f(p_1,x) \lll_{\mathcal{A}} t_1$. But $\|f(p_1,x)\| \leq \|t_1\| \Rightarrow 1 + \|p_1\| + \|x\| \leq \|t_1\| \Rightarrow \|x\| \leq \|t_1\| - 1 - \|p_1\|$ which finally results in $\|x\| < \|t_1\| - 1$. For the third equation, the reasoning is the same.

The number of variables' occurrences in D is the sum of the occurrences in each term, and is denoted by $\#\mathcal{V}ar(D)$, *i.e.* $\#\mathcal{V}ar(D) = \sum \#\mathcal{V}ar(t)$, for all $t \in D$. The variables' occurrences in a term are computed as $\#\mathcal{V}ar(t) = \#\{\omega \mid t|_\omega \in \mathcal{X}\}$.

Termination is easy to show for all the rules, except Mutate and Replacement. Therefore, we focus on these two rules and we consider a lexicographical order $\phi = (\phi_1, \phi_2)$, where $\phi_1 = \|D\|$, and $\phi_2 = \#\mathcal{V}ar(D)$, which is decreasing for the application of each of the two rules:

- Mutate: $\|f(p_1,p_2) \lll_{\mathcal{A}} f(t_1,t_2)\| = \{\|f(t_1,t_2)\|\} = \{1 + \|t_1\| + \|t_2\|\}$. The size of

each equation from the right-hand side is strictly smaller:

- $\|p_1 \prec\!\!\prec_{\mathcal{A}} t_1\| = \|t_1\|$
- $\|p_2 \prec\!\!\prec_{\mathcal{A}} t_2\| = \|t_2\|$
- $\|p_2 \prec\!\!\prec_{\mathcal{A}} f(x, t_2)\| = \{\|f(x, t_2)\|\} < \{\|f(t_1, t_2)\|\}$ as $\|x\| = \|t_1\| - 1$
- $\|f(p_1, x) \prec\!\!\prec_{\mathcal{A}} t_1\| = \|t_1\|$
- $\|p_1 \prec\!\!\prec_{\mathcal{A}} f(t_1, x)\| = \{\|f(t_1, x)\|\} < \{\|f(t_1, t_2)\|\}$ as $\|x\| = \|t_2\| - 1$
- $\|f(x, p_2) \prec\!\!\prec_{\mathcal{A}} t_2\| = \|t_2\|$

Therefore for the right-hand side of the rule we have that

$$\phi_1 = \{\{\|t_1\|\}, \{\|t_2\|\}, \{\|t_1\| + \|t_2\|\}, \{\|t_1\|\}, \{\|t_1\| + \|t_2\|\}, \{\|t_2\|\}\}$$

which is strictly smaller that the size of the left-hand side $\{\{1 + \|t_1\| + \|t_2\|\}\}$. This implies that ϕ_1 is decreasing, and although ϕ_2 increases (because we add new variables), ϕ is lexicographically decreasing.

- Replacement: we deal with two types of variables – the free and the quantified ones:
  - when replacing a free variable, the size remains constant, as all the free variables are in the left-hand sides. Therefore ϕ_1 is constant, but ϕ_2 is strictly decreasing.
  - when introduced (by the rule Mutate), a quantified variable appears twice: once on the left-hand side of an equation, and once on the right-hand side. Therefore, this occurrence on the left-hand side, when instantiated, will be used to replace the one in the right-hand side. But, as we noticed before, they can only be instantiated with a term smaller than their size. Consequently, when replaced in an equation E, the size of E decreases. Therefore ϕ_1 is strictly decreasing.

  Thus, in both cases ϕ is decreasing.

□

If no solution is lost in the application of a transformation rule, the rule is called *preserving*. It is a *sound* rule if it does not introduce unexpected solutions.

Proposition 3. The rules in $\mathcal{A}$-Match are sound and preserving modulo $\mathcal{A}$.

Proof. The rule Mutate is a direct consequence of the decomposition rules for syntactic theories presented in [KK90]. The rest of the rules are usual ones for which these results have been obtained for example in [CK01]. □

Theorem 3. *Given a matching equation $p \prec\!\!\prec_{\mathcal{A}} t$, with $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $t \in \mathcal{T}(\mathcal{F})$, the normal form w.r.t. $\mathcal{A}$-Match exists and it is unique. It can only be of the following types:*

1. *$\top$, then p and t are identical modulo $\mathcal{A}$, i.e. $p =_{\mathcal{A}} t$,*

2. *$\bot$, then there is no match from p to t,*

3. *a disjunction of conjunctions $\bigvee_{j\in J}(\wedge_{i\in I} x_{i_j} \ll_{\mathcal{A}} t_{i_j})$ with $I, J \neq \emptyset$, then the substitutions $\sigma_j = \{x_{i_j} \mapsto t_{i_j}\}_{i\in I, j\in J}$ are all the matches from p to t.*

Proof. From Proposition 2 a normal form always exists. Moreover, from Proposition 3 we can infer that it is unique (we do not have two different results for two different applications of the algorithm on the same problem), as after the application of $\mathcal{A}$-Match we have the same solutions as the initial problem. Therefore, we have to prove that (i) all the quantifiers are eliminated and (ii) all match-equation's left-hand sides are variables of the initial equation. We only have existential quantifiers, introduced by Mutate, which are distributed to each conjunction by Exists_2 and later eliminated by the rule Exists_1. The validity of the condition of this latter rule is ensured by the rule Replacement, which leaves only one occurrence of each variable in a conjunction. On the other hand, we never eliminate free variables in a conjunction (only some duplicates), which justifies (ii). Finally, all normal forms are necessarily of the form (1), (2) or (3), otherwise a rule could be further applied. □

Example 14. Applying $\mathcal{A}$-Match for $f \in \mathcal{F}_{\mathcal{A}}$, $x, y \in \mathcal{X}$, and $a, b, c, d \in \mathcal{T}(\mathcal{F})$:
$f(x, f(a, y)) \ll_{\mathcal{A}} f(f(b, f(a, c)), d)$
$\mapsto\!\!\!\!\rightarrow_{\text{Mutate}} (x \ll_{\mathcal{A}} f(b, f(a, c)) \wedge f(a, y) \ll_{\mathcal{A}} d) \vee$
$\exists z(f(a, y) \ll_{\mathcal{A}} f(z, d) \wedge f(x, z) \ll_{\mathcal{A}} f(b, f(a, c))) \vee$
$\exists z(x \ll_{\mathcal{A}} f(f(b, f(a, c)), z) \wedge f(z, f(a, y)) \ll_{\mathcal{A}} d)$
$\mapsto\!\!\!\!\rightarrow_{\text{SymbolClash}_1,\text{PropagateClash}_2} \exists z(f(a, y) \ll_{\mathcal{A}} f(z, d) \wedge f(x, z) \ll_{\mathcal{A}} f(b, f(a, c)))$
$\mapsto\!\!\!\!\rightarrow_{\text{Mutate},\text{SymbolClash}_1} \exists z(f(a, y) \ll_{\mathcal{A}} f(z, d) \wedge$
$((x \ll_{\mathcal{A}} b \wedge z \ll_{\mathcal{A}} f(a, c)) \vee (x \ll_{\mathcal{A}} f(b, a) \wedge z \ll_{\mathcal{A}} c)))$
$\mapsto\!\!\!\!\rightarrow_{\text{DistributeAnd},\text{Replacement},\text{Mutate},\text{SymbolClash}_{1,2}} \exists z(f(a, y) \ll_{\mathcal{A}} f(z, d) \wedge x \ll_{\mathcal{A}} b \wedge z \ll_{\mathcal{A}} f(a, c)) \mapsto\!\!\!\!\rightarrow_{\text{Replacement},\text{Exists},\text{Mutate},\text{SymbolClash}_{1,2}} x \ll_{\mathcal{A}} b \wedge y \ll_{\mathcal{A}} f(c, d)$.

Matching associative patterns with unit elements

It is often the case that associative operators have a unit and we know since the early works on *e.g.* OBJ, that this is quite useful from a rule programming point of view. For example, to state *a list* L *that contains the objects a and b*. This can be expressed by the pattern $f(x, f(a, f(y, f(b, z))))$, where $x, y, z \in \mathcal{X}$, which will match $f(c, f(a, f(d, f(b, e))))$ but not $f(a, b)$ or $f(c, f(a, b))$. When f has for unit e_f, the previous pattern does match modulo $\mathcal{AU}$, producing the substitution $\{x \mapsto e_f, y \mapsto e_f, z \mapsto e_f\}$ for $f(a, b)$, and $\{x \mapsto c, y \mapsto e_f, z \mapsto e_f\}$ for $f(c, f(a, b))$. However, $\mathcal{A}$ is a theory with a finite equivalence class, which is not the case of $\mathcal{AU}$, and an immediate consequence is that the set of matches becomes infinite. For instance, $\mathcal{S}ol(x \ll_{\mathcal{AU}} a)$ $= \{\{x \mapsto a\}, \{x \mapsto f(e_f, a)\}, \{x \mapsto f(e_f, f(e_f, a))\}, \ldots\}$.

In order to obtain a matching algorithm for $\mathcal{AU}$, we replace SymbolClash rules in $\mathcal{A}$-Match to appropriately handle unit elements (remember that we assume, because of modularity, that we only have in $\mathcal{F}$ a single binary $\mathcal{AU}$ symbol f, and constants, includ-

ing e_f):

$$\begin{array}{llcl} \mathsf{SymbolClash}_1^+ & f(p_1,p_2) \prec\!\!\prec_{\mathcal{AU}} a & \mapsto\!\!\!\rightarrow & (p_1 \prec\!\!\prec_{\mathcal{AU}} e_f \wedge p_2 \prec\!\!\prec_{\mathcal{AU}} a)\vee \\ & & & (p_1 \prec\!\!\prec_{\mathcal{AU}} a \wedge p_2 \prec\!\!\prec_{\mathcal{AU}} e_f) \\ \mathsf{SymbolClash}_2^+ & a \prec\!\!\prec_{\mathcal{AU}} f(p_1,p_2) & \mapsto\!\!\!\rightarrow & (e_f \prec\!\!\prec_{\mathcal{AU}} p_1 \wedge a \prec\!\!\prec_{\mathcal{AU}} p_2)\vee \\ & & & (a \prec\!\!\prec_{\mathcal{AU}} p_1 \wedge e_f \prec\!\!\prec_{\mathcal{AU}} p_2) \end{array}$$

In addition, we keep all other transformation rules, only changing all match symbols from $\prec\!\!\prec_{\mathcal{A}}$ to $\prec\!\!\prec_{\mathcal{AU}}$. The new system, named $\mathcal{AU}$-Match, is clearly terminating without producing in general a minimal set of solutions. After proving its correctness, we will see what can be done in order to minimize the set of solutions. The proof of correctness uses the following lemma:

Lemma 1. Let t_1 and t_2 be two ground terms. Matching them modulo $\mathcal{AU}$ is equivalent to match modulo $\mathcal{A}$ their $\mathcal{U}$-normal forms (denoted ${t_1}_{\downarrow_\mathcal{U}}$ and ${t_2}_{\downarrow_\mathcal{U}}$):

$$t_1 \prec\!\!\prec_{\mathcal{AU}} t_2 \Leftrightarrow {t_1}_{\downarrow_\mathcal{U}} \prec\!\!\prec_{\mathcal{A}} {t_2}_{\downarrow_\mathcal{U}}$$

Proof. Direct application of [Hue80, Theorem 3.3], since the unit rules are linear and terminating modulo $\mathcal{A}$, and associativity is regular. □

Proposition 4. The rules of $\mathcal{AU}$-Match are sound and preserving modulo $\mathcal{AU}$.

Proof. Thanks to Proposition 3, we know that the rules are sound and preserving modulo $\mathcal{A}$. In order to be also valid modulo $\mathcal{AU}$, they have to remain valid in the presence of the equations for neutral elements.
Let us first see the preserving property of the rules:

- ConstantClash, Replacement, Delete, Exists_1, Exists_2, DistributeAnd, $\mathsf{PropagateSuccess}_1$, $\mathsf{PropagateClash}_1$, $\mathsf{PropagateSuccess}_2$, $\mathsf{PropagateClash}_2$: these rules do not depend on the theory we consider.
- Mutate: we need to prove that for $\sigma \in \mathcal{S}ol(f(p_1,p_2) =_{\mathcal{AU}} f(t_1,t_2))$, $\exists\rho$ such that at least one of the following is true:
  - $\sigma\rho(p_1) =_{\mathcal{AU}} \rho(t_1) \wedge \sigma\rho(p_2) =_{\mathcal{AU}} \rho(t_2)$
  - $\sigma\rho(p_2) =_{\mathcal{AU}} \rho(f(x,t_2)) \wedge \sigma\rho(f(p_1,x)) =_{\mathcal{AU}} \rho(t_1)$
  - $\sigma\rho(p_1) =_{\mathcal{AU}} \rho(f(t_1,x)) \wedge \sigma\rho(f(x,p_2)) =_{\mathcal{AU}} \rho(t_2)$

  which are equivalent, by Lemma 1, to:

  1. $\sigma\rho(p_1)_{\downarrow_\mathcal{U}} =_{\mathcal{A}} \rho(t_1)_{\downarrow_\mathcal{U}} \wedge \sigma\rho(p_2)_{\downarrow_\mathcal{U}} =_{\mathcal{A}} \rho(t_2)_{\downarrow_\mathcal{U}}$
  2. $\sigma\rho(p_2)_{\downarrow_\mathcal{U}} =_{\mathcal{A}} \rho(f(x,t_2))_{\downarrow_\mathcal{U}} \wedge \sigma\rho(f(p_1,x))_{\downarrow_\mathcal{U}} =_{\mathcal{A}} \rho(t_1)_{\downarrow_\mathcal{U}}$
  3. $\sigma\rho(p_1)_{\downarrow_\mathcal{U}} =_{\mathcal{A}} \rho(f(t_1,x))_{\downarrow_\mathcal{U}} \wedge \sigma\rho(f(x,p_2))_{\downarrow_\mathcal{U}} =_{\mathcal{A}} \rho(t_2)_{\downarrow_\mathcal{U}}$

  But $\sigma \in \mathcal{S}ol(f(p_1,p_2) =_{\mathcal{AU}} f(t_1,t_2)) \Rightarrow f(\sigma\rho(p_1),\sigma\rho(p_2)) =_{\mathcal{AU}} f(\rho(t_1),\rho(t_2))$ for a chosen ρ which is equivalent to $f(\sigma\rho(p_1),\sigma\rho(p_2))_{\downarrow_\mathcal{U}} =_{\mathcal{A}} f(\rho(t_1),\rho(t_2))_{\downarrow_\mathcal{U}}$. We have the following possible cases:

1. neither $f(\sigma\rho(p_1), \sigma\rho(p_2))$ nor $f(\rho(t_1), \rho(t_2))$ can be reduced by $\mathcal{U}$. This means that $f(\sigma\rho(p_1), \sigma\rho(p_2)) =_{\mathcal{AU}} f(\rho(t_1), \rho(t_2)) \Leftrightarrow f(\sigma\rho(p_1), \sigma\rho(p_2)) =_{\mathcal{A}} f(\rho(t_1), \rho(t_2))$, which implies (by the rule Mutate that was proved to be $\mathcal{A}$-preserving) the disjunction of the three cases above.
2. only $f(\sigma\rho(p_1), \sigma\rho(p_2))$ can be reduced by $\mathcal{U}$:
   a) $\sigma\rho(p_1)_{\downarrow\mathcal{U}} \neq e_f$, $\sigma\rho(p_2)_{\downarrow\mathcal{U}} \neq e_f$. This gives $f(\sigma\rho(p_1)_{\downarrow\mathcal{U}}, \sigma\rho(p_2)_{\downarrow\mathcal{U}}) =_{\mathcal{A}} f(\rho(t_1), \rho(t_2))$ which again implies the three cases above.
   b) $\sigma\rho(p_1)_{\downarrow\mathcal{U}} = e_f$. This results in $\sigma\rho(p_2)_{\downarrow\mathcal{U}} =_{\mathcal{A}} f(\rho(t_1), \rho(t_2))$ which is equivalent with the second case for $\rho(x) = \rho(t_1)$.
   c) $\sigma\rho(p_2)_{\downarrow\mathcal{U}} = e_f$. Implies the second case with $\rho(x) = \rho(t_2)$.
3. only $f(\rho(t_1), \rho(t_2))$ can be reduced. As above , we consider all the three possible cases reasoning exactly in the same fashion.
4. both $f(\sigma\rho(p_1), \sigma\rho(p_2))$ and $f(\rho(t_1), \rho(t_2))$ are reducible. This case is just the combination of all the possibilities we have enounced above, therefore nine cases, which are solved similarly.

- SymbolClash$_1^+$: $\sigma \in \mathcal{S}ol(f(p_1, p_2) =_{\mathcal{AU}} g(\bar{t})) \Rightarrow f(\sigma(p_1), \sigma(p_2))_{\downarrow\mathcal{U}} =_{\mathcal{A}} a$. When both $\sigma(p_1)_{\downarrow\mathcal{U}}$ and $\sigma(p_2)_{\downarrow\mathcal{U}}$ are different from e_f, the equation $f(\sigma(p_1)_{\downarrow\mathcal{U}}, \sigma(p_2)_{\downarrow\mathcal{U}}) =_{\mathcal{A}} a$ has no solution as SymbolClash can be applied. If at least one of them is equal to e_f, we have the exact correspondence with the right-hand side of the rule: $\sigma(p_1)_{\downarrow\mathcal{U}} =_{\mathcal{A}} e_f \wedge \sigma(p_2)_{\downarrow\mathcal{U}} =_{\mathcal{A}} a \ \vee\ \sigma(p_1)_{\downarrow\mathcal{U}} =_{\mathcal{A}} a \wedge \sigma(p_2)_{\downarrow\mathcal{U}} =_{\mathcal{A}} e_f$.
- SymbolClash$_2^+$: The same reasoning as above.

The soundness justification follows the same pattern. For example, for the rule Mutate, which is the most interesting one, we have to prove that there exists ρ, such that that given σ which validates at least one of the disjunctions, we obtain the left-hand side of the rule. As above, first case is when only $\sigma\rho(p_1)$ and $\sigma\rho(p_2)$ can be reduced by $\mathcal{U}$, and $\sigma\rho(p_1)_{\downarrow\mathcal{U}} \neq e_f$ and $\sigma\rho(p_2)_{\downarrow\mathcal{U}} \neq e_f$. The question if $\sigma\rho(p_1)_{\downarrow\mathcal{U}} =_{\mathcal{A}} \rho(t_1) \wedge \sigma\rho(p_2)_{\downarrow\mathcal{U}} =_{\mathcal{A}} \rho(t_2)$ implies $f(\sigma(p_1)_{\downarrow\mathcal{U}}, \sigma\rho(p_2)_{\downarrow\mathcal{U}}) =_{\mathcal{A}} f(\rho(t_1), \rho(t_2))$ is obviously true. The rest of the cases are similar. □

In order to avoid redundant solutions we further consider that all the terms are in normal form *w.r.t.* the rewrite system $\mathcal{U} = \{f(e_f, x) \to x, f(x, e_f) \to x\}$. Therefore, we perform a normalized rewriting [Mar96] modulo $\mathcal{U}$. This technique ensures that before applying any rule from Figure 3.1, the terms are in normal forms *w.r.t.* $\mathcal{U}$.

We presented in this section rule-based algorithms for solving syntactic, $\mathcal{A}$ and $\mathcal{AU}$ matching. In the following sections, we will adapt these algorithms to solve anti-pattern matching problems.

3.5. Solving equational anti-pattern matching

In this section we expose several techniques for solving anti-pattern matching equations. Given an anti-pattern q and a ground term t, we first present a solution for transforming

the equation $q \prec\!\!\prec_{\mathcal{E}} t$ into an equational problem that has the same solutions and that no longer contains $\urcorner$ symbols. As this equational problem is very similar to a disunification one, we further analyze how disunification can be used to solve it (for the syntactic case). But disunification is too general, so we further propose more efficient approaches. We exemplify them on the $\mathcal{AU}$ case, but they can be easily adapted for other theories as well, including the empty one.

3.5.1. From anti-pattern matching to equational problems

A natural question that raises is how anti-pattern matching and equational problems are related. For instance, the interpretation of $\urcorner q \prec\!\!\prec t$ should not be $q \neq t$. Although this may be correct in the case of ground terms, like $\urcorner a \prec\!\!\prec b$, it is not true in the general case. Take for example $\urcorner g(x) \prec\!\!\prec g(a)$, which according to Definition 36 has no solution. But the solutions of $g(x) \neq g(a)$ are the solutions of $x \neq a$, *e.g.* $x = b$, $x = c$, *etc.* In this section we provide a way of transforming any anti-pattern matching problem into a corresponding equational one that has the same set of solutions.

Given an anti-pattern q and a ground term t, we consider the following transformation rule ElimAnti. Applying recursively this rule we can transform any anti-pattern matching problem into an equational one, by eliminating all $\urcorner$ symbols:

$$\mathsf{ElimAnti} \quad q[\urcorner q']_\omega \prec\!\!\prec_{\mathcal{E}} t \quad \mapsto\!\!\!\!\rightarrow \quad \exists z\; q[z]_\omega \prec\!\!\prec_{\mathcal{E}} t \;\wedge \forall x \in \mathcal{FV}ar(q')\; not(q[q']_\omega \prec\!\!\prec_{\mathcal{E}} t)$$
$$\text{if } \forall\, \omega' < \omega,\; q(\omega') \neq \urcorner \text{ and } z \text{ a fresh variable}$$

An anti-pattern matching problem P not containing any $\urcorner$ symbol, is a first-order formula where the symbol *not* is the usual negation of predicate logic, the symbol $\prec\!\!\prec_{\mathcal{E}}$ is interpreted as $=_{\mathcal{E}}$ and the symbol $\forall$ is the usual universal quantification: $\forall x \mathsf{P} \equiv not(\exists x\; not(\mathsf{P}))$. Therefore it is exactly an $\mathcal{E}$-disunification problem. For instance, if we apply this rule on the example we provided earlier, $\urcorner g(x) \prec\!\!\prec g(a)$, we obtain $\exists z\; z \prec\!\!\prec g(a) \;\wedge\; \forall x\; not(g(x) \prec\!\!\prec g(a))$ which is equivalent with $\forall x\; g(x) \neq g(a)$, that has no solution. Thus, for this example this transformation is valid. As shown below they are also valid in the general case:

Proposition 5. The rule ElimAnti is sound and preserving modulo $\mathcal{E}$: it does not introduce unexpected solutions, and no solution is lost in the application of the rule.

Proof. We consider a position ω such that $q[\urcorner q']_\omega$ and $\forall\, \omega' < \omega,\; q(\omega') \neq \urcorner$. Considering as usual that $Sol(A \wedge B) = Sol(A) \cap Sol(B)$ we have the following result for the right-hand side of the rule:

$$Sol(\exists z\; q[z]_\omega \prec\!\!\prec_{\mathcal{E}} t \;\wedge\; \forall x \in \mathcal{FV}ar(q')\; not(q[q']_\omega \prec\!\!\prec_{\mathcal{E}} t))$$
$$= Sol(\exists z\; q[z]_\omega \prec\!\!\prec_{\mathcal{E}} t\;) \cap Sol(\forall x \in \mathcal{FV}ar(q') not(q[q']_\omega \prec\!\!\prec_{\mathcal{E}} t))$$

From Definition 38, $Sol(\exists z\; q[z]_\omega \prec\!\!\prec_{\mathcal{E}} t)$ is equal to:

$$\{\sigma \mid \mathsf{Dom}(\sigma) = \mathcal{FV}ar(q[z]) \backslash \{z\} \text{ and } \exists \rho \text{ with } \mathsf{Dom}(\rho) = \{z\}, t \in [\![\sigma\rho(q[z]_\omega)]\!]_{g\mathcal{E}}\} \quad (3.1)$$

Also from the consequence of Definition 38, $Sol(\forall x \in \mathcal{FV}ar(q')\ not(q[q']_\omega \prec\!\!\prec_\mathcal{E} t))$ is equal to:

$$\{\sigma \mid t \notin [\![\sigma(q[q']_\omega)]\!]_{g\mathcal{E}}\ with\ \mathsf{Dom}(\sigma) = \mathcal{FV}ar(q[q']) \setminus \mathcal{FV}ar(q')\} \tag{3.2}$$

For the left part of the rule ElimAnti, by Definition 37, we have:

$$\begin{aligned} Sol\quad & (q[\urcorner q']_\omega \prec\!\!\prec_\mathcal{E} t) = \{\sigma \mid t \in [\![\sigma(q[\urcorner q']_\omega)]\!]_{g\mathcal{E}},\ with\ \mathsf{Dom}(\sigma) = \mathcal{FV}ar(q[\urcorner q'])\} \\ & = \{\sigma \mid t \in ([\![\sigma(q[z]_\omega)]\!]_{g\mathcal{E}} \backslash [\![\sigma(q[q']_\omega)]\!]_{g\mathcal{E}}),\ with \ldots\},\ \text{since } \forall \omega' < \omega, q(\omega') \neq \urcorner \\ & = \{\sigma \mid t \in [\![\sigma(q[z]_\omega)]\!]_{g\mathcal{E}}\ and\ t \notin [\![\sigma(q[q']_\omega)]\!]_{g\mathcal{E}},\ with\ \mathsf{Dom}(\sigma) = \mathcal{FV}ar(q[\urcorner q'])\} \\ & = \{\sigma \mid t \in [\![\sigma(q[z]_\omega)]\!]_{g\mathcal{E}},\ with \ldots\} \cap\ \{\sigma \mid t \notin [\![\sigma(q[q']_\omega)]\!]_{g\mathcal{E}}\ with \ldots\} \end{aligned} \tag{3.3}$$

Now it remains to check the equivalence of (3.3) with the intersection of (3.1) and (3.2). First of all, $\mathcal{FV}ar(q[z]) \backslash \{z\} = \mathcal{FV}ar(q[q']) \setminus \mathcal{FV}ar(q') = \mathcal{FV}ar(q[\urcorner q'])$ which means that we have the same domain for σ in (3.3), (3.1), and (3.2). Therefore, we have to prove:

$$\{\sigma \mid \exists \rho \text{ with } \mathsf{Dom}(\rho) = \{z\} \text{ and } t \in [\![\sigma\rho(q[z]_\omega)]\!]_{g\mathcal{E}}\} = \{\sigma \mid t \in [\![\sigma(q[z]_\omega)]\!]_{g\mathcal{E}}\} \tag{3.4}$$

But σ does not instantiate z, and this means that the ground semantics will give to z all the possible values for the right part of (3.4). At the same time, having ρ existentially quantified allows z to be instantiated with any value such that $t \in [\![\rho\sigma(q[z]_\omega)]\!]_{g\mathcal{E}}$ is valid, and therefore (3.4) is true. As we considered the symbol $\urcorner$ at an arbitrary position, we can conclude that the rule is sound and preserving, wherever it is applied on a term. □

The normal forms *w.r.t.* ElimAnti of anti-pattern matching problems are specific equational problems. Although equational problems are undecidable in general [Tre92], even in case of $\mathcal{A}$ or $\mathcal{AU}$ theories, we will see that the specific equational problems issued from anti-pattern matching are decidable for the empty, $\mathcal{A}$ and $\mathcal{AU}$ theories.

Summarizing, if we know how to solve equational problems modulo $\mathcal{E}$, then any anti-pattern matching problem modulo $\mathcal{E}$ can be translated into equivalent equational problems using ElimAnti and further solved. These statements are formalized by the following Proposition:

Proposition 6. An anti-pattern matching problem can *always* be translated into an equivalent equational problem in a finite number of steps.

Proof. We showed in the proof of Proposition 5 that ElimAnti preserves the solutions if applied on a matching problem. Each of its applications transforms one equation in two equivalent equations (that preserve solutions). Each new equation contains less occurrences of $\urcorner$, therefore, for a finite number n of $\urcorner$ symbols, ElimAnti terminates and it is easy to show that the normal forms contain at most 2^n equations and disequations. □

3.5.2. Solving syntactic anti-pattern matching via disunification

Using the rewrite rule ElimAnti, we can eliminate all $\urcorner$ symbols from any anti-pattern matching problem. The normal forms have the following structure: $\exists z\ q \prec\!\!\prec t\ \wedge\ \forall x\ not(\exists z'\ q' \prec\!\!\prec t\ \wedge\ \forall x'\ not(\ldots))$. We interpret the symbol $\prec\!\!\prec$ as $=$, and we consider a set of boolean simplification rules, called DeMorgan, that is applied on these normal forms:

$$
\begin{array}{lcl|lcl}
not(\exists z\ P) & \mapsto\!\!\!\!\!\rightarrow & \forall z\ not(P) & not(not(a)) & \mapsto\!\!\!\!\!\rightarrow & a \\
not(\forall z\ P) & \mapsto\!\!\!\!\!\rightarrow & \exists z\ not(P) & not(a = b) & \mapsto\!\!\!\!\!\rightarrow & a \neq b \\
not(a \wedge b) & \mapsto\!\!\!\!\!\rightarrow & not(a) \vee not(b) & not(a \neq b) & \mapsto\!\!\!\!\!\rightarrow & a = b \\
not(a \vee b) & \mapsto\!\!\!\!\!\rightarrow & not(a) \wedge not(b) & & &
\end{array}
$$

The resulting expression no longer contains any *not*, and thus is a classical equational problem. We call it an *anti-pattern disunification problem.*

A natural way to solve these types of problems is to use a disunification algorithm such as described in [CL90]. As it is not of high interest for this paper, we don't present disunification in detail. Instead we give in Figure 3.2 the set of rules we consider. The interested reader can refer to [CL90] for a detailed presentation of disunification.

Disunification rules

[CL90] presents a set of disunification rules that is proved to be sound and preserving. Moreover, irreducible problems for these rules are definitions with constraints, *i.e.* either $\top$, $\perp$ or a conjunction of equalities and disequalities. In Figure 3.2 we present this set of rules, but tailored for anti-pattern matching problems. It is still sound and preserving, but also ensures (thanks to Theorem 4) that for each problem a normal form exists and is unique. We will further call it AntiMatchDisunif.

From the classical presentation of disunification rules, three rules have been removed. They were no longer necessary in the restricted case of the anti-patterns, as their application conditions are never fulfilled.

Three new rules, that were proved to be sound and preserving in [Com88], were added. They ensure the elimination of all variables that are existentially quantified. The justification is simple, and consists in showing that any problem containing an occurrence of an existentially quantified variable is reducible: if there is such a variable, one of the three introduced rules is tried. The condition $z \notin \mathcal{V}ar(\mathsf{S})$ may prevent from applying a rule. In that case, we have $z \in \mathcal{V}ar(\mathsf{S})$ and therefore one of the following rules can be applied: Replacement (or Merging), Decompose (or Clash) — if the variable z is inside a term.

In [CL90] there is a clear separation between the elimination of parameters and the rules that reach definitions with constraints. But, as affirmed both in [CL90] and [Com88], such a strict control is only for presentation purposes. In our algorithm, we use a single step approach.

Universality$_1$	$\forall z(z = t \wedge \mathsf{S})$	$\mapsto\!\!\!\!\rightarrow$	$\bot$
Universality$_2$	$\forall z(z \neq t \wedge \mathsf{S})$	$\mapsto\!\!\!\!\rightarrow$	$\bot$
Universality$_3$	$\forall z\ \mathsf{S}$	$\mapsto\!\!\!\!\rightarrow$	S if $z \notin \mathcal{V}ar(\mathsf{S})$
Universality$_4$	$\forall z(\mathsf{S} \wedge (z \neq t \vee \mathsf{S}'))$	$\mapsto\!\!\!\!\rightarrow$	$\forall z(\mathsf{S} \wedge (t \mapsto z)\mathsf{S}')$
Universality$_5$	$\forall z(\mathsf{S} \wedge (z = t \vee \mathsf{S}'))$	$\mapsto\!\!\!\!\rightarrow$	$\forall z(\mathsf{S} \wedge \mathsf{S}')$ if $z \notin \mathcal{V}ar(\mathsf{S}')$
Replacement	$z = t \wedge \mathsf{S}$	$\mapsto\!\!\!\!\rightarrow$	$z = t \wedge (t \mapsto z)\mathsf{S}$
Elimination$_1$	$a = a$	$\mapsto\!\!\!\!\rightarrow$	$\top$
Elimination$_2$	$a \neq a$	$\mapsto\!\!\!\!\rightarrow$	$\bot$
PropagateClash$_1$	$\mathsf{S} \wedge \bot$	$\mapsto\!\!\!\!\rightarrow$	$\bot$
PropagateClash$_2$	$\mathsf{S} \vee \bot$	$\mapsto\!\!\!\!\rightarrow$	S
PropagateSuccess$_1$	$\mathsf{S} \wedge \top$	$\mapsto\!\!\!\!\rightarrow$	S
PropagateSuccess$_2$	$\mathsf{S} \vee \top$	$\mapsto\!\!\!\!\rightarrow$	$\top$
Clean$_1$	$a \wedge a$	$\mapsto\!\!\!\!\rightarrow$	a
Clean$_2$	$a \vee a$	$\mapsto\!\!\!\!\rightarrow$	a
Clash$_1$	$f(p_1, \ldots, p_n) = g(t_1, \ldots, t_n)$	$\mapsto\!\!\!\!\rightarrow$	$\bot$ if $f \neq g$
Clash$_2$	$f(p_1, \ldots, p_n) \neq g(t_1, \ldots, t_n)$	$\mapsto\!\!\!\!\rightarrow$	$\top$ if $f \neq g$
Decompose$_1$	$f(p_1, \ldots, p_n) = f(t_1, \ldots, t_n)$	$\mapsto\!\!\!\!\rightarrow$	$\bigwedge_{i=1..n} p_i = t_i$
Decompose$_2$	$f(p_1, \ldots, p_n) \neq f(t_1, \ldots, t_n)$	$\mapsto\!\!\!\!\rightarrow$	$\bigvee_{i=1..n} p_i \neq t_i$
Merging$_1$	$z = t \wedge z = u$	$\mapsto\!\!\!\!\rightarrow$	$z = t \wedge t = u$
Merging$_2$	$z \neq t \vee z \neq u$	$\mapsto\!\!\!\!\rightarrow$	$z \neq t \vee t \neq u$
Merging$_3$	$z = t \wedge z \neq u$	$\mapsto\!\!\!\!\rightarrow$	$z = t \wedge t \neq u$
Merging$_4$	$z = t \vee z \neq u$	$\mapsto\!\!\!\!\rightarrow$	$t = u \vee z \neq u$
Removed rules:	OccurCheck, Explosion, Elimination of disjunctions		
New rules:			
Exists$_1$	$\exists z\ \mathsf{S}$	$\mapsto\!\!\!\!\rightarrow$	S if $z \notin \mathcal{V}ar(\mathsf{S})$
Exists$_2$	$\exists z(\mathsf{S} \wedge (z \neq t \vee \mathsf{S}'))$	$\mapsto\!\!\!\!\rightarrow$	S if $z \notin \mathcal{V}ar(\mathsf{S})$
Exists$_3$	$\exists z(\mathsf{S} \wedge (z = t \vee \mathsf{S}'))$	$\mapsto\!\!\!\!\rightarrow$	S if $z \notin \mathcal{V}ar(\mathsf{S})$

Figure 3.2.: Simplified presentation of the disunification rules: AntiMatchDisunif

Solved forms

In the following we show that an *anti-pattern disunification problem* (resulting from the application of ElimAnti, followed by DeMorgan can be simplified by the rewrite system AntiMatchDisunif, given in Figure 3.2, such that it does not contain any disjunction or disequality.

Example 15. If we consider $f(x, \urcorner y) \prec\!\!\prec f(a, b)$, the corresponding anti-pattern disunification problem is computed in the following way:

$$\begin{aligned} f(x, \urcorner y) \prec\!\!\prec f(a,b) &\mapsto\!\!\!\!\rightarrow f(x, \urcorner y) = f(a,b) \\ &\mapsto\!\!\!\!\rightarrow \exists z\ f(x,z) = f(a,b) \wedge \forall y\ not(f(x,y) = f(a,b)) \\ &\mapsto\!\!\!\!\rightarrow \exists z\ f(x,z) = f(a,b) \wedge \forall y\ f(x,y) \neq f(a,b). \end{aligned}$$

Proposition 7. Given an *anti-pattern disunification problem*, the normal form wrt. the rewrite system AntiMatchDisunif does not contain disjunctions or disequalities.

Proof. We consider an anti-pattern $q \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$, and an arbitrary application of ElimAnti:

$$q[\urcorner q']_\omega = t \quad \mapsto\!\!\!\!\rightarrow \quad \exists z\ q[z]_\omega = t\ \wedge\ \forall x \in \mathcal{FV}ar(q')\ not(q[q']_\omega = t)$$

If a disequality or a disjunction is produced, it comes from the $not(q[q']_\omega = t)$. We now consider the variables that occur in this expression. Each of them belongs to one of the following classes:

1. the free variables of q',
2. the free variables of $q[q']_\omega$ — excepting the free variables of q',
3. the variables of $q[q']_\omega$ that are not free.

In the following we show that the normal form cannot contain such a variable. Therefore, the normalization of $\forall x \in \mathcal{FV}ar(q'), not(q[q']_\omega = t)$ leads to either $\top$ or $\bot$:

1. these are universally quantified variables, and they will be eliminated by Universality rules,
2. let us consider $y \in \mathcal{FV}ar(q[q']_\omega) \backslash \mathcal{FV}ar(q')$, and let us suppose that the reduction of $not(q[q']_\omega = t)$ generates the disequality $y \neq t_{|\omega_1}$, then the reduction of the first part $\exists z\ q[z]_\omega = t$ will generate $y = t_{|\omega_2}$, with $\omega_2 = \omega_1$ because t and the skeleton of q are the same in both parts. By applying the Replacement rule, all the occurrences of $y \neq t_{|\omega_1}$ are transformed in $t_{|\omega_1} \neq t_{|\omega_1}$ and later eliminated,
3. any variable that is not free (i.e. is under a $\urcorner$) will be universally quantified by a further application of the rule ElimAnti, therefore later eliminated by Universality$_1$ or Universality$_2$.

□

Theorem 4. *Given an* anti-pattern disunification problem*, its normal form wrt. the rewrite system* AntiMatchDisunif *exists and is unique.*

1. *when it is of the form $\bigwedge_{i \in I} x_i = t_i$ with $I \neq \emptyset$ and $x_i \neq x_j$ for all $i \neq j$, the substitution $\sigma = \{x_i \mapsto t_i\}_{i \in I}$ is the solution of the matching problem,*
2. *when it is $\top$, any substitution σ is a solution of the matching problem,*
3. *when it is $\bot$, the matching problem has no solution.*

Proof. By applying Proposition 7. □

Simple examples

Let us show on a few examples how the rules behave. First with one complement:

$f(a, \urcorner b) \prec\!\!\prec f(a, a)$
$\mapsto\!\!\!\!\rightarrow f(a, \urcorner b) = f(a, a)$
$\mapsto\!\!\!\!\rightarrow \exists z f(a, z) = f(a, a) \wedge not(f(a, b) = f(a, a))$
$\mapsto\!\!\!\!\rightarrow \exists z f(a, z) = f(a, a) \wedge f(a, b) \neq f(a, a)$
$\mapsto\!\!\!\!\rightarrow \exists z(a = a \wedge z = a) \wedge (a \neq a \vee b \neq a)$
$\mapsto\!\!\!\!\rightarrow \exists z(z = a) \wedge (\perp \vee \top)$
$\mapsto\!\!\!\!\rightarrow \top \wedge \top$
$\mapsto\!\!\!\!\rightarrow \top.$

Of course complements can be nested as illustrated below:

$\urcorner f(a, \urcorner b) \prec\!\!\prec f(a, b)$
$\mapsto\!\!\!\!\rightarrow \urcorner f(a, \urcorner b) = f(a, b)$
$\mapsto\!\!\!\!\rightarrow \exists z\ z = f(a, b) \wedge not(f(a, \urcorner b) = f(a, b))$
$\mapsto\!\!\!\!\rightarrow \exists z\ z = f(a, b) \wedge not(\ \exists z' f(a, z') = f(a, b) \wedge not(f(a, b) = f(a, b)))$
$\mapsto\!\!\!\!\rightarrow \exists z\ z = f(a, b) \wedge (\forall z' f(a, z') = f(a, b) \vee f(a, b) = f(a, b))$
$\mapsto\!\!\!\!\rightarrow \top \wedge (\forall z'(a = a \wedge z' = b) \vee (a = a \wedge b = b))$
$\mapsto\!\!\!\!\rightarrow \forall z'(z' = b) \vee \top$
$\mapsto\!\!\!\!\rightarrow \top.$

We can also consider anti-pattern problems with variables, such as $f(\urcorner a, x) \prec\!\!\prec f(b, c)$, whose solution is $\{x \mapsto c\}$. The pattern can be non-linear: $f(x, \urcorner x) \prec\!\!\prec f(a, b)$, leading to $\{x \mapsto a\}$. Nested negation and non-linearity can be combined:

$\urcorner f(x, \urcorner g(x)) \prec\!\!\prec f(a, g(b))$
$\mapsto\!\!\!\!\rightarrow \urcorner f(x, \urcorner g(x)) = f(a, g(b))$
$\mapsto\!\!\!\!\rightarrow \exists z\ z = f(a, g(b)) \wedge \forall x\ not(f(x, \urcorner g(x)) = f(a, g(b)))$
$\mapsto\!\!\!\!\rightarrow \exists z\ z = f(a, g(b)) \wedge \forall x\ not(\exists z'\ f(x, z') = f(a, g(b)) \wedge \forall x\ not(f(x, g(x)) = f(a, g(b))))$
$\mapsto\!\!\!\!\rightarrow \exists z\ z = f(a, g(b)) \wedge \forall x(\forall z'\ f(x, z') = f(a, g(b)) \vee \exists x\ f(x, g(x)) = f(a, g(b)))$
$\mapsto\!\!\!\!\rightarrow \top \wedge \forall x\ (\forall z'\ (x = a \wedge z' = g(b)) \vee \exists x\ (x = a \wedge g(x) = g(b)))$
$\mapsto\!\!\!\!\rightarrow \forall x\ (x = a \wedge \forall z'\ (z' = g(b)) \vee \exists x\ (x = a \wedge x = b))$
$\mapsto\!\!\!\!\rightarrow \forall x\ (x = a \wedge \perp \vee \exists x\ (x = a \wedge a = b))$
$\mapsto\!\!\!\!\rightarrow \forall x\ (\perp \vee \exists x\ (x = a \wedge \perp))$
$\mapsto\!\!\!\!\rightarrow \forall x\ (\perp \vee \perp)$
$\mapsto\!\!\!\!\rightarrow \perp.$

Summing up the relations with disunification

When comparing anti-pattern problems with general disunification ones, there are many similarities, but some important differences also. In the anti-pattern case, a solved form does not contain any quantifier whereas disunification allows existential ones. Another important difference is the unitary property (Theorem 4) which is obviously not true for disunification: $x \neq a$ has many solutions in general. Disunification contains rules (called *globally preserving*) that return an equational problem whose solutions are a subset of the given problem. The Explosion and the Elimination of disjunctions rules are

such examples. In our case, the complexity is dramatically reduced since these rules are unnecessary.

3.5.3. More tailored approaches

As we saw in the previous section, solving equational problems resulting from normalization with ElimAnti can be performed with techniques like disunification. But these techniques were designed to cover more general problems, and in our case, a more efficient and tailored approach can be developed. Given a finitary $\mathcal{E}$-match algorithm, a first solution would be to normalize each match equation separately, then to combine the results using replacements and some cleaning rules (as ForAll, NotOr, NotTrue, NotFalse from Figure 3.4). This approach can be used to effectively solve syntactic, $\mathcal{A}$, $\mathcal{AU}$, and $\mathcal{AC}$ anti-pattern matching problems. We further detail the $\mathcal{AU}$ case.

A specific case: $\mathcal{AU}$ anti-pattern matching

Definition 39 ($\mathcal{AU}$-AntiMatch)**.** *Given an $\mathcal{AU}$ anti-pattern matching problem $q \prec\!\!\prec_{\mathcal{AU}} t$, apply the rules from Figure 3.4 on page 54, giving a higher priority to* ElimAnti*.*

Note that instead of giving a higher priority to ElimAnti the algorithm can be decomposed in two steps: first normalize with ElimAnti to eliminate all $\urcorner$ symbols, then apply all the other rules.

We further prove that the algorithm is correct. Moreover, the normal forms of its application on an $\mathcal{AU}$ anti-pattern matching equation do not contain any $\urcorner$ or *not* symbols. Actually they are the same as the ones exposed in Theorem 3.

Proposition 8. The application of $\mathcal{AU}$-AntiMatch is sound and preserving.

Proof. For ElimAnti these properties were shown in the proof of Proposition 5. Similarly, Proposition 4 states the sound and preserving properties for the rules of $\mathcal{AU}$-Match. The rest of the rules are trivial. □

Theorem 5. *The normal forms of $\mathcal{AU}$-*AntiMatch *are $\mathcal{AU}$-matching problems in solved form.*

Proof. The normal forms clearly do not contain any $\urcorner$ symbols, as we normalize with ElimAnti. Universal quantifications are also eliminated by the rule ForAll followed by Exists_1 and Exists_2. Let us now prove that the *not* symbols are also eliminated. The matching equations containing only ground terms are clearly reduced to either $\top$ or $\perp$ and further eliminated. The variables under the *not* symbol can be of two types: quantified — which will be eliminated by the rule Exists_1 — and not quantified. In this case, it means that they were not under a $\urcorner$ symbol, and therefore they are free variables that we can find in the context of *not*. In other words, for any $x_i \prec\!\!\prec_{\mathcal{AU}} t_i$ under the *not* symbol, where x_i is not universally quantified, there exists a corresponding $x_i \prec\!\!\prec_{\mathcal{AU}} t_i$ in the context. Given that, the rule Replacement will transform the equations under the *not* in simpler equations that will be further reduced to $\top$.

□

ElimAnti	$q[\urcorner q']_\omega \prec\!\!\prec t$	$\mapsto\!\!\!\!\rightarrow$	$\exists z\ q[z]_\omega \prec\!\!\prec t\ \wedge \forall x \in \mathcal{FV}ar(q')\ not(q[q']_\omega \prec\!\!\prec t)$ if $\forall\ \omega' < \omega,\ q(\omega') \neq \urcorner$ and z a fresh variable

DeMorgan rules :			
	$not(\exists z\ P)$	$\mapsto\!\!\!\!\rightarrow$	$\forall z\ not(P)$
	$not(\forall z\ P)$	$\mapsto\!\!\!\!\rightarrow$	$\exists z\ not(P)$
	$not(a \wedge b)$	$\mapsto\!\!\!\!\rightarrow$	$not(a) \vee not(b)$
	$not(a \vee b)$	$\mapsto\!\!\!\!\rightarrow$	$not(a) \wedge not(b)$
	$not(not(a))$	$\mapsto\!\!\!\!\rightarrow$	a
	$not(a = b)$	$\mapsto\!\!\!\!\rightarrow$	$a \neq b$
	$not(a \neq b)$	$\mapsto\!\!\!\!\rightarrow$	$a = b$
Disunification rules :			
Universality$_1$	$\forall z(z = t \wedge \mathsf{S})$	$\mapsto\!\!\!\!\rightarrow$	$\bot$
Universality$_2$	$\forall z(z \neq t \wedge \mathsf{S})$	$\mapsto\!\!\!\!\rightarrow$	$\bot$
Universality$_3$	$\forall z\ \mathsf{S}$	$\mapsto\!\!\!\!\rightarrow$	S if $z \notin \mathcal{V}ar(\mathsf{S})$
Universality$_4$	$\forall z(\mathsf{S} \wedge (z \neq t \vee \mathsf{S}'))$	$\mapsto\!\!\!\!\rightarrow$	$\forall z(\mathsf{S} \wedge (t \mapsto z)\mathsf{S}')$
Universality$_5$	$\forall z(\mathsf{S} \wedge (z = t \vee \mathsf{S}'))$	$\mapsto\!\!\!\!\rightarrow$	$\forall z(\mathsf{S} \wedge \mathsf{S}')$ if $z \notin \mathcal{V}ar(\mathsf{S}')$
Replacement	$z = t \wedge \mathsf{S}$	$\mapsto\!\!\!\!\rightarrow$	$z = t \wedge (t \mapsto z)\mathsf{S}$
Elimination$_1$	$a = a$	$\mapsto\!\!\!\!\rightarrow$	$\top$
Elimination$_2$	$a \neq a$	$\mapsto\!\!\!\!\rightarrow$	$\bot$
PropagateClash$_1$	$\mathsf{S}\ \wedge\ \bot$	$\mapsto\!\!\!\!\rightarrow$	$\bot$
PropagateClash$_2$	$\mathsf{S} \vee \bot$	$\mapsto\!\!\!\!\rightarrow$	S
PropagateSuccess$_1$	$\mathsf{S}\ \wedge\ \top$	$\mapsto\!\!\!\!\rightarrow$	S
PropagateSuccess$_2$	$\mathsf{S} \vee \top$	$\mapsto\!\!\!\!\rightarrow$	$\top$
Clean$_1$	$a \wedge a$	$\mapsto\!\!\!\!\rightarrow$	a
Clean$_2$	$a \vee a$	$\mapsto\!\!\!\!\rightarrow$	a
Clash$_1$	$f(p_1, \ldots, p_n) = g(t_1, \ldots, t_n)$	$\mapsto\!\!\!\!\rightarrow$	$\bot$ if $f \neq g$
Clash$_2$	$f(p_1, \ldots, p_n) \neq g(t_1, \ldots, t_n)$	$\mapsto\!\!\!\!\rightarrow$	$\top$ if $f \neq g$
Decompose$_1$	$f(p_1, \ldots, p_n) = f(t_1, \ldots, t_n)$	$\mapsto\!\!\!\!\rightarrow$	$\bigwedge_{i=1..n} p_i = t_i$
Decompose$_2$	$f(p_1, \ldots, p_n) \neq f(t_1, \ldots, t_n)$	$\mapsto\!\!\!\!\rightarrow$	$\bigvee_{i=1..n} p_i \neq t_i$
Merging$_1$	$z = t \wedge z = u$	$\mapsto\!\!\!\!\rightarrow$	$z = t \wedge t = u$
Merging$_2$	$z \neq t \vee z \neq u$	$\mapsto\!\!\!\!\rightarrow$	$z \neq t \vee t \neq u$
Merging$_3$	$z = t \wedge z \neq u$	$\mapsto\!\!\!\!\rightarrow$	$z = t \wedge t \neq u$
Merging$_4$	$z = t \vee z \neq u$	$\mapsto\!\!\!\!\rightarrow$	$t = u \vee z \neq u$
Exists$_1$	$\exists z\ \mathsf{S}$	$\mapsto\!\!\!\!\rightarrow$	S if $z \notin \mathcal{V}ar(\mathsf{S})$
Exists$_2$	$\exists z(\mathsf{S} \wedge (z \neq t \vee \mathsf{S}'))$	$\mapsto\!\!\!\!\rightarrow$	S if $z \notin \mathcal{V}ar(\mathsf{S})$
Exists$_3$	$\exists z(\mathsf{S} \wedge (z = t \vee \mathsf{S}'))$	$\mapsto\!\!\!\!\rightarrow$	S if $z \notin \mathcal{V}ar(\mathsf{S})$

Figure 3.3.: The complete set of rules for solving anti-pattern matching problems via disunification: we first normalize with ElimAnti, second we replace the symbol $\prec\!\!\prec$ with $=$ and we normalize with DeMorgan rules. We finally normalize with the disunification rules.

ElimAnti	$q[\urcorner q']_\omega \prec\!\!\prec_{\mathcal{AU}} t$	$\mapsto\!\!\!\!\rightarrow$	$\exists z\ q[z]_\omega \prec\!\!\prec_{\mathcal{AU}} t \ \wedge\ \forall x \in \mathcal{FV}ar(q')\ not(q[q']_\omega \prec\!\!\prec_{\mathcal{AU}} t)$ if $\forall\ \omega' < \omega,\ q(\omega') \neq \urcorner$ and z a fresh variable
ForAll	$\forall \bar{y}\ not(\mathsf{D})$	$\mapsto\!\!\!\!\rightarrow$	$not(\exists \bar{y}\ \mathsf{D})$
NotOr	$not(\mathsf{D}_1 \vee \mathsf{D}_2)$	$\mapsto\!\!\!\!\rightarrow$	$not(\mathsf{D}_1) \wedge not(\mathsf{D}_2)$
NotTrue	$not(\top)$	$\mapsto\!\!\!\!\rightarrow$	$\bot$
NotFalse	$not(\bot)$	$\mapsto\!\!\!\!\rightarrow$	$\top$
$\mathcal{AU}$-Match Rules:			
Mutate	$f(p_1, p_2) \prec\!\!\prec_{\mathcal{AU}} f(t_1, t_2)$	$\mapsto\!\!\!\!\rightarrow$	$(p_1 \prec\!\!\prec_{\mathcal{AU}} t_1 \wedge p_2 \prec\!\!\prec_{\mathcal{AU}} t_2) \vee$ $\exists x(p_2 \prec\!\!\prec_{\mathcal{AU}} f(x, t_2) \wedge f(p_1, x) \prec\!\!\prec_{\mathcal{AU}} t_1) \vee$ $\exists x(p_1 \prec\!\!\prec_{\mathcal{AU}} f(t_1, x) \wedge f(x, p_2) \prec\!\!\prec_{\mathcal{AU}} t_2)$
$\mathsf{SymbolClash}_1^+$	$f(p_1, p_2) \prec\!\!\prec_{\mathcal{AU}} a$	$\mapsto\!\!\!\!\rightarrow$	$(p_1 \prec\!\!\prec_{\mathcal{AU}} e_f \wedge p_2 \prec\!\!\prec_{\mathcal{AU}} a) \vee$ $(p_1 \prec\!\!\prec_{\mathcal{AU}} a \wedge p_2 \prec\!\!\prec_{\mathcal{AU}} e_f)$
$\mathsf{SymbolClash}_2^+$	$a \prec\!\!\prec_{\mathcal{AU}} f(p_1, p_2)$	$\mapsto\!\!\!\!\rightarrow$	$(e_f \prec\!\!\prec_{\mathcal{AU}} p_1 \wedge a \prec\!\!\prec_{\mathcal{AU}} p_2) \vee$ $(a \prec\!\!\prec_{\mathcal{AU}} p_1 \wedge e_f \prec\!\!\prec_{\mathcal{AU}} p_2)$
ConstantClash	$a \prec\!\!\prec_{\mathcal{AU}} b$	$\mapsto\!\!\!\!\rightarrow$	$\bot$ if $a \neq b$
Replacement	$z \prec\!\!\prec_{\mathcal{AU}} t \wedge \mathsf{S}$	$\mapsto\!\!\!\!\rightarrow$	$z \prec\!\!\prec_{\mathcal{AU}} t \wedge \{z \mapsto t\}\mathsf{S}$ if $z \in \mathcal{FV}ar(\mathsf{S})$
Utility Rules:			
Delete	$p \prec\!\!\prec_{\mathcal{AU}} p$	$\mapsto\!\!\!\!\rightarrow$	$\top$
Exists_1	$\exists z(\mathsf{D}[z \prec\!\!\prec_{\mathcal{AU}} t])$	$\mapsto\!\!\!\!\rightarrow$	$\mathsf{D}[\top]$ if $z \notin \mathcal{V}ar(\mathsf{D}[\top])$
Exists_2	$\exists z(\mathsf{S}_1 \vee \mathsf{S}_2)$	$\mapsto\!\!\!\!\rightarrow$	$\exists z(\mathsf{S}_1) \vee \exists z(\mathsf{S}_2)$
DistributeAnd	$\mathsf{S}_1 \wedge (\mathsf{S}_2 \vee \mathsf{S}_3)$	$\mapsto\!\!\!\!\rightarrow$	$(\mathsf{S}_1 \wedge \mathsf{S}_2) \vee (\mathsf{S}_1 \wedge \mathsf{S}_3)$
$\mathsf{PropagateClash}_1$	$\mathsf{S} \wedge \bot$	$\mapsto\!\!\!\!\rightarrow$	$\bot$
$\mathsf{PropagateClash}_2$	$\mathsf{S} \vee \bot$	$\mapsto\!\!\!\!\rightarrow$	S
$\mathsf{PropagateSuccess}_1$	$\mathsf{S} \wedge \top$	$\mapsto\!\!\!\!\rightarrow$	S
$\mathsf{PropagateSuccess}_2$	$\mathsf{S} \vee \top$	$\mapsto\!\!\!\!\rightarrow$	$\top$

Figure 3.4.: $\mathcal{AU}$-AntiMatch

$\mathcal{AU}$-AntiMatch is a general algorithm, that solves any anti-pattern matching problem. Note that it can produce 2^n matching equations, where n is the number of $\urcorner$ symbols in the initial problem. For instance, applying ElimAnti on $f(a, \urcorner b) \prec\!\!\prec_{\mathcal{AU}} f(a, a)$ gives $\exists z f(a, z) \prec\!\!\prec_{\mathcal{AU}} f(a, a) \wedge not(f(a, b) \prec\!\!\prec_{\mathcal{AU}} f(a, a))$. Note that all equations have the same right-hand sides $f(a, a)$, and *almost* the same left-hand sides $f(a, _\ )$. Therefore, when solving the second equation for instance, we perform some matches that were already done when solving the first one. This approach is clearly not optimal, and in the following we propose a more efficient one.

A more efficient algorithm

In this section we consider a subclass of anti-patterns, called $Pure\mathcal{FV}ars$, and we present a more efficient algorithm that has the same complexity as $\mathcal{AU}$-Match. In particular, it does no longer produce the 2^n equations introduced by $\mathcal{AU}$-AntiMatch.

Definition 40 (PureFVars)**.** *Given $\mathcal{F}$,$\mathcal{X}$ we define a subclass of anti-patterns:*

$$PureFVars = \left\{ q \in \mathcal{AT}(\mathcal{F},\mathcal{X}) \;\middle|\; \begin{array}{c} q = \mathrm{C}[f(t_1,\ldots,t_i,\ldots,t_j,\ldots,t_n)], \\ \forall i \neq j,\ \mathcal{FV}ar(t_i)\ \cap\ \mathcal{NFV}ar(t_j) = \emptyset \end{array} \right\}$$

The anti-patterns in $Pure\mathcal{FV}ars$ are special cases of non-linearity respecting that at any position, we don't find a term that has a free variable in one of its children, and the same variable under a $\urcorner$ in another child. For instance, $f(x,x) \in Pure\mathcal{FV}ars$, $f(\urcorner x,\urcorner x) \in Pure\mathcal{FV}ars$, but $f(x,\urcorner x) \notin Pure\mathcal{FV}ars$.

Definition 41 ($\mathcal{AU}$-AntiMatchEfficient)**.** *The algorithm corresponds to $\mathcal{AU}$-AntiMatch, where the rule ElimAnti is replaced with the following one, and which has no longer any priority:*

$$\text{ElimAnti'}\quad \urcorner q \prec\!\!\prec_{\mathcal{AU}} t \quad \mapsto\!\!\!\rightarrow \quad \forall x \in \mathcal{FV}ar(q)\ not(q \prec\!\!\prec_{\mathcal{AU}} t)$$

Note that our algorithms are finitary and based on decomposition. Therefore, when considering syntactic or regular theories the composition results for matching algorithms are still valid. Note also that $Pure\mathcal{FV}ars$ is trivially stable *w.r.t.* to this algorithm and that now the rules apply on problems that potentially contain $\urcorner$ symbols. For instance, we may apply the rule Mutate on $f(a,\urcorner b) \prec\!\!\prec_{\mathcal{AU}} f(a,a)$. The algorithm is still terminating, with the same arguments as in the proof of Proposition 2, but the proof of Proposition 4 is no longer valid in this new case. The correctness of the algorithm has to be established again:

Proposition 9. Given an anti-pattern matching equation $q \prec\!\!\prec_{\mathcal{AU}} t$, with $q \in Pure\mathcal{FV}ars$, the application of $\mathcal{AU}$-AntiMatchEfficient is sound and preserving.

Proof. The most interesting rule is Mutate. First, let us prove the preserving property: $\sigma \in Sol(f(p_1,p_2) \prec\!\!\prec_{\mathcal{AU}} f(t_1,t_2))$ implies from Definition 36 that

$$f(t_1,t_2) \in [\![f(\sigma(p_1),\sigma(p_2))]\!]_{g\mathcal{AU}},\ \text{with } \sigma \in \mathcal{GS}(f(p_1,p_2)) \Rightarrow \exists t \in [\![f(\sigma(p_1),\sigma(p_2))]\!]_g$$

such that $f(t_1,t_2) =_{\mathcal{AU}} t$. But $t \in [\![f(\sigma(p_1),\sigma(p_2))]\!]_g$ implies that $t = f(u,v)$, where $u \in [\![\sigma(p_1)]\!]_g$ and $v \in [\![\sigma(p_2)]\!]_g$. Further more, $f(t_1,t_2) =_{\mathcal{AU}} f(u,v)$ is equivalent (from Proposition 4) with

$(t_1 =_{\mathcal{AU}} u \wedge t_2 =_{\mathcal{AU}} v)$
$\vee\ \exists x(t_2 =_{\mathcal{AU}} f(x,v) \wedge f(t_1,x) =_{\mathcal{AU}} u)$
$\vee\ \exists x(t_1 =_{\mathcal{AU}} f(u,x) \wedge f(x,t_2) =_{\mathcal{AU}} v)$.

But $u \in [\![\sigma(p_1)]\!]_g$ and $v \in [\![\sigma(p_2)]\!]_g$, and therefore we have that

$(t_1 \in [\![\sigma(p_1)]\!]_g \wedge t_2 \in [\![\sigma(p_2)]\!]_g)$
$\vee\ (t_2 \in [\![\sigma(f(x,p_2))]\!]_g \wedge f(t_1,x) \in [\![\sigma(p_1)]\!]_g)$
$\vee\ (t_1 \in [\![\sigma(f(p_1,x))]\!]_g \wedge f(x,t_2) \in [\![\sigma(p_2)]\!]_g)$

which means exactly that σ is the solution of the right-hand side of our initial equation, except for the fact that $\sigma \in \mathcal{GS}(f(p_1, p_2))$ and we need other domains for σ. For instance, for $t_1 \in [\![\sigma(p_1)]\!]_g$ we need that $\sigma \in \mathcal{GS}(p_1)$. But this is immediately implied by the restriction of the class $Pure\mathcal{FV}ars$, because it is not possible to have a variable that is free in $f(p_1, p_2)$ and not free in p_1. Therefore $\sigma \in \mathcal{GS}(f(p_1, p_2))$ is equivalent with $\sigma \in \mathcal{GS}(p_1)$ when applying σ on p_1. Consequently, we have that the rule preserves the solutions. The soundness follows the same reasoning. The proof for the rest of the rules is trivial. □

This approach is much more efficient, as no duplications are being made. Let us see on a simple example: $f(x, \urcorner a) \prec\!\!\prec_{\mathcal{AU}} f(a, b) \mapsto\!\!\!\!\rightarrow_{\mathsf{Mutate}} (x \prec\!\!\prec_{\mathcal{AU}} a \wedge \urcorner a \prec\!\!\prec_{\mathcal{AU}} b) \vee \mathsf{D}_1 \vee \mathsf{D}_2 \mapsto\!\!\!\!\rightarrow_{\mathsf{ElimAnti'}} (x \prec\!\!\prec_{\mathcal{AU}} a \wedge not(a \prec\!\!\prec_{\mathcal{AU}} b)) \vee \mathsf{D}_1 \vee \mathsf{D}_2 \mapsto\!\!\!\!\rightarrow_{\mathsf{ConstantClash}} (x \prec\!\!\prec_{\mathcal{AU}} a \wedge not(\perp)) \vee \mathsf{D}_1 \vee \mathsf{D}_2 \mapsto\!\!\!\!\rightarrow_{\mathsf{NotFalse,PropagateSuccess_2}} x \prec\!\!\prec_{\mathcal{AU}} a \vee \mathsf{D}_1 \vee \mathsf{D}_2$. We continue in a similar way for D_1,D_2 and we finally obtain the solution $\{x \mapsto a\}$.

In practice, when implementing an anti-pattern matching algorithm, one can imagine the following approach: a traversal of the term is done, and if the special non-linear case is detected (*i.e.* $\notin Pure\mathcal{FV}ars$), then $\mathcal{AU}$-AntiMatch is applied; otherwise we apply $\mathcal{AU}$-AntiMatchEfficient.

In this section we presented several techniques to solve anti-pattern matching problems: using disunification, using a more specific algorithm, and finally a more efficient approach for a subclass which encompasses most of the practical cases. We also conjecture that modifying the universal quantification of ElimAnti' to only quantify variables that respect the condition $\mathcal{FV}ar(q_1) \cap \mathcal{NFV}ar(q_2) = \emptyset$ of $Pure\mathcal{FV}ars$, would still lead to a sound and complete algorithm. For instance, when applying ElimAnti' to $f(x, \urcorner x)$, the variable x would not be quantified. This algorithm has been experimented and tested without showing any counter example. Proving this conjecture is part of our future work.

3.6. Compiling anti-patterns

In this section we present how anti-patterns were integrated in Tom. Our goal here is not to give a formal framework for compiling the anti-patterns, but rather to give an idea of how they were compiled with regard to the algorithms presented in this chapter.

Integrating the anti-patterns in Tom was quite a challenge. Each of the presented algorithms was used at some point. The order in which they are presented in this thesis follows exactly not only the order of their discovery but also how anti-pattern marching was supported in Tom. It is actually by implementing the algorithms that we had the ideas for more simple and efficient ones.

A first particularity of the implementations when compared to the algorithms presented is that Tom has to compile the pattern matching, and therefore the anti-pattern matching as well, without knowing the subject. Therefore, the subject is always a variable, and the code produced by Tom uses methods that decompose the subject at runtime according to its type and the pattern against which it is matched. We will

present in detail the compilation process in a further chapter, but let's look at a simple example in order to have a more precise idea of how this works:

```
// pseudocode for the compilation of f(x, a) ≺≺ t
if(is_fsym_f(t)) then  // if the subject is an 'f'
  assign x = subterm_f(t, 1) in // assign to x the first subterm of t
  assign t_2 = subterm_f(t, 2) in
  if(is_fsym_a(t_2)) then  // if t_2 is an 'a' we have a MATCH !
    ... // do something
  endif ;
endif ;
```

The first algorithm that we used was of course the disunification one. It wasn't completely integrated in the compiler, it was more like a separate module. When the compiler encountered an anti-pattern, he delegated the compilation to the anti-pattern matching module and it got back the result. The implementation was quite straightforward and respected in detail the algorithm we presented. It had however several drawbacks: first, it wasn't very efficient, as we may imagine. Secondly, it was quite a lot of code compared to the compiler of Tom: it was half the size of the whole compiler. Last but not least, anti-pattern matching is morally just an extension of the pattern matching. Therefore, implementing an anti-pattern matching algorithm completely separately from the pattern matching algorithm already implemented by the Tom compiler leads to an undesirable duplication of code.

All these issues led us to search more efficient approaches. Therefore, we exploited another particularity of the implementation that is not valid in the general setting the algorithms were presented: the evaluation of the code is performed from left to right. If we use this property, a lot of the duplications introduced by the ElimAnti can be avoided. For instance, for solving $\mathtt{f}(\mathtt{a}, \urcorner \mathtt{b}) \prec\!\!\prec \mathtt{t}$, we apply ElimAnti followed by $\mathsf{Decompose}_1$ and we obtain:

$$\begin{aligned}&\exists \mathtt{z}\ \mathtt{f}(\mathtt{a},\mathtt{z}) \prec\!\!\prec \mathtt{t} \wedge \mathtt{not}(\mathtt{f}(\mathtt{a},\mathtt{b}) \prec\!\!\prec \mathtt{t})\\ &= \exists \mathtt{z}(\mathtt{is_fsym}_\mathtt{f}(\mathtt{t}) \wedge \mathtt{is_fsym}_\mathsf{a}(\mathtt{subterm}_\mathtt{f}(\mathtt{t},1)) \wedge \mathtt{z} \prec\!\!\prec \mathtt{subterm}_\mathtt{f}(\mathtt{t},2))\\ &\wedge \mathtt{not}(\mathtt{is_fsym}_\mathtt{f}(\mathtt{t}) \wedge \mathtt{is_fsym}_\mathsf{a}(\mathtt{subterm}_\mathtt{f}(\mathtt{t},1)) \wedge \mathtt{is_fsym}_\mathsf{b}(\mathtt{subterm}_\mathtt{f}(\mathtt{t},2)))\end{aligned}$$

If the above expression is evaluated from left to right, the tests $\mathtt{is_fsym}_\mathtt{f}(\mathtt{t})$ and $\mathtt{is_fsym}_\mathsf{a}(\mathtt{subterm}_\mathtt{f}(\mathtt{t},1))$ can be safely eliminated from the second part of the expression. They were already performed in the first part, and therefore we are sure that they were evaluated to true when we reach their second appearance. Therefore we can reduce the expression to:

$$\begin{aligned}&\exists \mathtt{z}(\mathtt{is_fsym}_\mathtt{f}(\mathtt{t}) \wedge \mathtt{is_fsym}_\mathsf{a}(\mathtt{subterm}_\mathtt{f}(\mathtt{t},1)) \wedge \mathtt{z} \prec\!\!\prec \mathtt{subterm}_\mathtt{f}(\mathtt{t},2))\\ &\wedge \mathtt{not}(\mathtt{is_fsym}_\mathsf{b}(\mathtt{subterm}_\mathtt{f}(\mathtt{t},2)))\end{aligned}$$

This type of simplification has an important impact on the performance of the algorithm, especially in the case of complex patterns or when several complement symbols are used. The existentially quantified variables can be also eliminated and we obtain:

$$\texttt{is_fsym}_\texttt{f}(\texttt{t}) \wedge \texttt{is_fsym}_\texttt{a}(\texttt{subterm}_\texttt{f}(\texttt{t},1)) \wedge \texttt{not}(\texttt{is_fsym}_\texttt{b}(\texttt{subterm}_\texttt{f}(\texttt{t},2)))$$

This result is actually what the algorithm $\mathcal{AU}$-AntiMatchEfficient would produce. There remain however two questions:

1. What happens with the universal quantifiers? How are they handled in the generated code?
2. How about the non-linearity problem enounced in the case of $\mathcal{AU}$-AntiMatchEfficient? Is the generated code correct for the anti-patterns that are not in the $Pure\mathcal{FV}ars$?

To answer the first question, let's take a look at the code that corresponds to a simple example: if we have that $\texttt{f}(\texttt{a}, \urcorner\texttt{g}(\texttt{x})) \prec\!\!\prec \texttt{t}$, then we want to print the message `OK`:

```
if(is_fsym_f(t)) then  // if the subject is an 'f'
  assign t_1 = subterm_f(t, 1) in
  assign t_2 = subterm_f(t, 2) in
  if(is_fsym_a(t_1)) then  // if t_1 is an 'a'
    assign doesMatch = false in // put a flag where the anti-pattern starts
    if(is_fsym_g(t_2)) then  // if t_2 is an 'g'
      assign x = subterm_g(t_2, 1) in
      assign doesMatch = true // we succeeded in matching the whole subterm
    endif ;
    if(equal(doesMatch, false)) then  // if we didn't have a match
      print(OK)
    endif ;
  endif ;
endif ;
```

In the above code we use a boolean variable, `doesMatch`, to test if the match of the subterm under the $\urcorner$ symbol was successful. If it is the case, we set its value to `true`. After we finish matching what was under the complement symbol, we test if `doesMatch` has the value `false`. If yes, it means that the subterm doesn't match, and this happens no matter what values we assign to variables. If `doesMatch` is set to `true`, it means that we found some values for the variables such that the subterm matches. This is how the universal quantifiers are encoded. Therefore we print the message `OK` only if the `g(x)` didn't match, whatever we assigned to `x`.

To answer the second question, we provide the code for the same matching problem, except that instead of $\texttt{f}(\texttt{a}, \urcorner\texttt{g}(\texttt{x})) \prec\!\!\prec \texttt{t}$ we consider $\texttt{f}(\texttt{x}, \urcorner\texttt{g}(\texttt{x})) \prec\!\!\prec \texttt{t}$:

```
if(is_fsym_f(t)) then  // if the subject is an f
  assign x = subterm_f(t, 1) in
  assign t_2 = subterm_f(t, 2) in
  assign doesMatch = false in // put a flag where the anti-pattern starts
  if(is_fsym_g(t_2)) then  // if t_2 is an g
    if(equal(x, subterm_g(t_2, 1))) then  // test the non-linearity constraint
```

```
    assign doesMatch = true // we succeeded in matching the whole subterm
   endif ;
  endif ;
  if(equal(doesMatch, false)) then  // if we didn't have a match
   print(OK)
  endif ;
endif ;
```

As we can notice, besides the first apparition of the variable `x` where it is assigned, all the subsequent apparitions are transformed into equality tests. Therefore, in a non-linear pattern, the first time a variables appears it it assigned, and all the other times we find the same variable we just test for the equality with the term it was supposed to be assigned to the variable. This is possible thanks to the scope of the variables in the code. Because of the way code is arranged (we always match the positive subterms before the complemented ones), a variable in a positive position will always have in its scope the variables from the negative positions. Therefore, an implicit propagation is performed, similar to the one the rule Replacement ensures.

3.7. Related work

Negation is part of our common way of reasoning, and therefore it has been widely studied and integrated in programming languages. Consequently, there is a huge amount of work that can be related in a way or another with the content of this work. In spite of this, the anti-patterns are quite a novelty for pattern matching languages. It is important to stress that we introduced the anti-patterns with the purpose of having a compact and permissive representation to match *ground terms*: the use of nested negations replaces the use of conjunctions and/or disjunctions and there is no restriction to linear terms for example. When using them with lists, they can spare the user of writing even more complex constructions, like loops combined with conditional statements. It is also a useful representation which is both intuitive and easy to compile in an efficient way. In the context of TOM, general algorithms such as disunification [Com91, CL90, Com88] could have been used. But since pattern-matching is the main execution mechanism, we were interested in a specialized approach that is both simpler and more efficient.

This section is divided in two parts: the first one describes some approaches that are closer to syntactic anti-patterns, while the second one mostly focuses on $\mathcal{AU}$ anti-patterns.

3.7.1. Syntactic anti-patterns

Lassez [LM87] presented a way of expressing exclusion by the means of counter-examples: typically, the expression $f(x,y)/\{f(a,u) \vee f(u,a)\}$ represents all the ground instances of $f(x,y)$, different from $f(a,u)$ and $f(u,a)$. Even though this is a useful and close

approach, it is more restrictive than the anti-patterns. Consider for example the anti-pattern $\urcorner f(a, \urcorner b)$, that cannot be represented by terms with counter-examples, unless we allow the counter-examples to also have counter-examples, i.e. $z/\{f(a, y/\{b\})\}$ — an issue not addressed in [LM87]. Moreover, the application domain of terms with counter-examples was rather machine learning than efficient term rewriting. This may explain why they restricted to linear terms and studied if these types of expressions have an equivalent representation using disjunctions. Actually, complementing non-linear terms was not very much addressed (except for disunification) and standard algorithms that computes complements are incorrect for non-linear terms, as mentioned in [Mom00]. Complementing higher order patterns is also considered only in the linear case.

Although the syntax of set constraints [AW92, MNP97, AKW95, CP94] allows the use of complement without any restriction of linearity or level of complement, we are not aware of any good semantics for the general case. Moreover, despite the fact that theoretically it is possible to have a constraint of the form $f(a, b) \subseteq \neg f(a, \neg b)$, existing implementations do not allow the complement in its fully generality. For example the CLP(Set) language in B-Prolog[2] allows the use of the symbol '\' as a unary operator representing the complement. However, it is only defined for variables, and not for constants. Another example is CLP(SC) [Fos96], where we are restricted to use only predicates of arity 0 and 1, which obviously cannot have the same expressiveness as anti-patterns. Besides that, it does not provide variable assignments. Constraints over features trees [BBN$^+$93, AKPS94, BS95] include the *exclusion constraint* which is a formula of the form $\neg\exists y(xfy)$, which says that the feature f is undefined for x, i.e. there is no edge that starts from x labeled with f. A more complex semantics of nested negations is not provided, for example to express that *there is no 'a' in relation with x, unless x is in relation with 'b'*.

CDuce[3] allows for the use of complement when declaring types but it restricts it to be used on types alone, and do not deal with variables complements.

ASF+SDF has a basic form of negative matching condition, denoted `p!:=t`, but negations inside the patterns like for instance $f(!a)$ are not supported. Consequently, nested negations are not supported as well.

Luigi Liquori integrated a form of anti-patterns in his implementation of the imperative rewriting calculus [Liq06]. Our work extends this presentation, providing formal semantics for both syntactic and equational anti-patterns, as well as several algorithms for solving anti-pattern matching problems.

The constrained terms, as defined in [Com88], can be used to obtain the semantics of some anti-patterns. They may have constraints — conjunction of disequalities — attached to their variables. Considering for example $f(a, \urcorner b)$, this is semantically equivalent to $f(a, z)$, *constrained by* $z \neq b$. But for a more complex expression, like $f(a, \urcorner g(b, \urcorner c))$, this approach is not expressive enough because the use of disjunctions in the constraints is not allowed.

[2] `http://www.probp.com/`
[3] `http://www.cduce.org/`

3.7.2. $\mathcal{AU}$ anti-patterns

Although we generalized the notion of anti-patterns to an arbitrary equational theory, most of the employed examples and algorithms in this paper concern $\mathcal{AU}$ anti-patterns. As we deal with terms (seen as trees), the pattern matching on XML documents is probably the closest to this work – as XML documents are trees built over associative-commutative symbols. We compare in this section the capabilities to express negative conditions of the main query languages with our approach based on anti-patterns.

TQL [CG04] is a query language for semistructured data based on the ambient logic that can be used to query XML files. It is a very expressive language and it can be used to capture most of the examples we provided along the paper. Moreover, TQL supports unlimited negation. The data model of TQL is unordered, it relies on $\mathcal{AC}$ operators and unary ones. Therefore, syntactic patterns are not supported in their full generality. For instance, it is not possible to express a pattern such as $\urcorner f(a, \urcorner b)$. More generally, syntactic anti-patterns and associative operators cannot be combined. In [CG04], the authors state that the extension of TQL with ordering is an important open issue. Compared to TQL, Tom is a mature implementation that can be easily integrated in a Java programming environment. It also offers good performance when dealing with large documents.

XDO2 [ZLCD05] is another query language for XML. It expresses negation with the use of a *not-predicate*, thus being able to support nested negations and negation of sub-trees. For instance, the following query retrieves the companies that don't have employees who have the sex M and age *40*:

```
/db/company:$c <= /root/company : $c/not(employee/[sex:"M",age:40])
```

In [ZLCD05] the authors present the main features of the language, but they do not provide the semantics for negation in the general case. The examples that they offer in [ZLCD05] are simple cases of negations, easy to express both in TQL and in the presented anti-pattern framework. Note also that non-linearity (which is a difficult and important part) was not studied in [ZLCD05].

XQuery provides a function *not()* for supporting negations. It can only be applied on a boolean argument, and returns the inverse value of its argument. The language also provides constructs like *some* and *every* which can be used to obtain the semantics of some anti-patterns. But this gives quite complicated queries that could be a lot simpler and compact by using anti-patterns.

3.8. Conclusion

3.8.1. Synthesis

We introduced in this chapter the notion of anti-patterns. We presented how they are integrated in the Tom language and the expressivity they add to the pattern-matching capabilities of this language. We then defined their semantics in both the syntactical and equational case. We extended the classical notion of matching patterns and ground

terms to matching between anti-patterns and ground terms. Based on the *syntacticness* of $\mathcal{AU}$ theory, we presented a rule-based algorithm for solving $\mathcal{AU}$ matching problems, which we later extended to solve anti-pattern matching problems. We also investigated how disunification can be used to solve syntactical anti-pattern matching, and we proved that anti-pattern matching is unitary and that the computed solved forms do not contain any disequality — properties that are not true for general disunification problems. We finally explored more efficient approaches, which we also exemplified on the $\mathcal{AU}$ theory.

3.8.2. Future work

The study of anti-patterns opens a number of challenging directions. A first one would be to prove the correctness of the last algorithm presented as a conjecture.

Other appealing directions consist in studying some theoretical properties such as the confluence, termination, and complete definition of systems that include anti-patterns. For instance, suppose that we have a rewrite system composed of several rewrite rules, some of which have their left-hand sides anti-patterns. How can we check the confluence and the termination of this system? Is it complete? Another interesting perspective is the study of unification problems in the presence of anti-patterns.

4. Contraintes non-atomiques comme membre gauche des règles

S'inspirant de l'expressivité des règles de production, nous proposons dans ce chapitre plusieurs extensions aux constructions de filtrage fournies par Tom. Par conséquent, la condition pour l'application d'une règle n'est plus une simple contrainte de filtrage, mais une combinaison de contraintes de filtrage et d'anti-filtrage.

4.1. Motivation

Lorsque j'ai démarré ma thèse en 2005, les constructions de filtrage de Tom correspondaient à celles présentées dans [MRV03]. Bien que expressives, elles ne permettent pas de décrire entièrement de manière algébrique la notion de règle conditionnelle. En effet, l'utilisateur est obligé de recourir aux constructions du langage hôte, ou souvent aux constructions **%match** imbriquées, qui rendent le code difficile à lire et à maintenir.

Les règles de production sont connues pour leur pouvoir d'expressivité important, permettant de combiner des conditions de filtrage et des prédicats. La similarité entre ces règles et celles de Tom rend possible l'utilisation de certains de leur concepts dans le cadre de filtrage de Tom.

4.1.1. Le filtrage classique

Le filtrage de Tom, comme nous avons détaillé dans la section 1.1.2 page 8, est introduit par la construction **%match** ayant pour syntaxe :

```
%match(subject_1,...,subject_n) {
  pattern_11,...,pattern_1n -> { action_1 }
  ...
  pattern_m1,...,pattern_mn -> { action_m }
}
```

La sémantique est similaire à celle d'une construction classique *switch/case*, avec la différence principale que la discrimination se fait sur des termes plutôt que sur des valeurs atomiques comme des entiers ou des caractères : du haut vers le bas, pour chaque ensemble de motifs `pattern_i1,...,pattern_in` qui filtrent vers les sujets `subject_1,...,subject_n`, l'action `action_i` correspondante est exécutée. Comme dans le cas de *switch/case*, si une action interrompt le flot de contrôle du programme (avec une instruction `break` ou `return` par exemple), les autres actions ne seront plus

exécutées. Les actions sont des instructions du langage hôte, comme JAVA par exemple, pouvant récursivement contenir des constructions TOM.

Cette construction permet d'encoder des systèmes de réécriture de termes, des règles conditionnelles, des calculs non-déterministes, *etc*. Par exemple, la règle de distributivité de la multiplication par rapport à l'addition peut être décrite par la règle de réécriture $or(x, plus(y, z)) \rightarrow plus(or(x, y), or(x, z))$. Pour implémenter cette règle en TOM on pourrait utiliser le code suivant :

```
%match(s) {
  or(x,plus(y,z)) -> { return `plus(or(x,y),or(x,z)); }
}
```

4.1.2. Limitations de l'approche actuelle

Le filtrage de TOM, tel qu'il a été présenté dans la section précédente, nous permet d'encoder de manière très expressive des règles de réécriture classiques, ou des analyses assez simples. Mais pour décrire un système de réécriture conditionnel, ou des transformations qui nécessitent des conditions plus complexes qu'un seul problème de filtrage, la construction **%match** est assez limitée. Bien-sûr que toutes ces conditions peuvent être encodées dans la partie *action*, en utilisant des instructions JAVA du type `if-then-else` et des constructions **%match** imbriquées. Considérons la règle suivante : étant donnée une classe `Person`, ayant les champs `age` et `profession`, et un objet `pers` de cette classe, on veut exécuter l'instruction (ou suite d'instructions) `action` si l'âge de `pers` est plus grand que `60` et la profession est soit `Professor` soit `Researcher`. Une possibilité d'écrire cette règle dans TOM serait la suivante :

```
%match(pers) {
  Person(age,profession) -> {
    if (age > 60) {
      %match(profession) {
        Professor[] -> { action }
        Researcher[] -> { action }
      }
    }
  }
}
```

Si l'action est une suite de plusieurs instructions JAVA, le code devient encore plus compliqué, parce qu'on est obligé d'utiliser un *flag* booléen pour ne pas dupliquer l'action.

Le principal problème avec cette approche est le fait que la sémantique du code, son intention, sont cachés par le mélange entre les instructions JAVA et le filtrage. La solution idéale serait d'avoir toutes les conditions dans la partie gauche de règle, et de ne pas faire des vérifications dans la partie action. De cette façon, on pourrait dire facilement quelles sont les conditions nécessaires pour qu'une action puisse être exécutée.

En plus, le fait d'avoir toutes les conditions dans la partie Tom des règles nous permettrait de faire plus d'optimisations et de vérifications. Par exemple, dans le code ci-dessus, il n'y a aucune liaison entre les deux constructions **%match**. Elles sont traitées complètement séparément par le compilateur de Tom.

4.1.3. Les règles de production

Les systèmes de règles de production (*business rule management systems*), un type particulier de systèmes experts [Jac99], sont un autre exemple de logiciels à base de règles. Une sémantique basée sur des systèmes de réécriture pour ces systèmes à été étudiée dans [SS96, CKMM04a, CKMM04b]. Parmi les instances les plus connues il y a ILog JRules, JBoss Rules, Jess ou Claire [CJL99].

Dans ces systèmes, les règles ont la forme suivante :

IF *conditions* THEN *action*

Les *conditions* sont habituellement des conditions de filtrage sur un ensemble d'objets appelé *Working Memory* (WM), combinées avec d'autres types de conditions, comme celles numériques ou booléenes par exemple. Les actions sont des appels de fonctions, qui peuvent effectuer différentes tâches allant d'une opération sur la WM (insertion ou rétraction d'objets) à l'affichage de messages par exemple. Un exemple de règle dans Jess est :

```
(defrule welcome-toddlers
"Give a special greeting to young children"
(person {age < 3})
=>
(printout t "Hello, little one!" crlf))
```

La règle sera déclenchée pour chaque objet `person` dans WM qui satisfait la condition `age < 3`. Une règle ne se déclenche pas deux fois de suite pour les mêmes objets qui satisfont ses conditions.

Un des points forts des règles de production est leur expressivité, grâce aux conditions variées qui peuvent être exprimées. Par exemple, nous pouvons combiner plusieurs problèmes de filtrage, conditions booléenes sur les variables (avec les opérateurs `>,<,==`, *etc*), expressions régulières, *etc.* L'expression suivante est un exemple de condition d'une règle dans JBoss Rules :

```
( Cheese( cheeseType : type )
  and (Person( favouriteCheese == cheeseType, sex == "f", age > 60 )
      or Person( favouriteCheese == cheeseType, sex == "m", age > 65 ) )
```

Cette condition vérifie si dans le WM il y a un objet `Cheese` et aussi une femme âgée de plus de 60 ans ou un homme âgée de plus de 65 ans ayant ce fromage comme fromage favori.

4.1.4. Un filtrage plus expressif

Les règles de Tom restent différentes de celles de production, essentiellement parce que le filtrage n'est pas fait sur un ensemble fini d'objets, mais chaque motif est filtré contre un seul objet. Les actions des règles se ressemblent dans les deux cas : elles peuvent être composées d'instructions, d'appels de fonctions, *etc.* Ainsi, les règles de Tom sont un compromis entre les règles de réécriture et celles de production.

Pour résoudre les inconvénients de **%match** présentés dans la section 4.1.2 page 64, une solution pratique serait de rendre les parties gauches des règles Tom plus proches de celles des règles de production. De cette façon, la condition d'application d'une règle ne sera plus une seule contrainte de filtrage, mais une combinaison de plusieurs contraintes de filtrage et d'autres conditions. L'idée principale est de permettre d'exprimer le plus possible dans les parties Tom des règles, et non pas dans Java, pour rendre le code plus concis, plus facile à lire, à vérifier et à optimiser.

4.2. Nouveau filtrage de Tom

Dans le but de rendre l'expressivité du **%match** plus proche de celle des règles de production, nous présentons dans ce chapitre les extensions qu'on a apporté aux membres gauches des règles de Tom.

4.2.1. Syntaxe

La nouvelle syntaxe de **%match** est présentée dans la figure suivante. La syntaxe BNF des conditions est détaillée dans la figure 4.1b.

```
%match {
  condition_1 -> { action_1 }
  ...
  condition_n -> { action_n }
}
```

(a) Le nouveau **%match**.

```
⟨condition⟩  ::=  ⟨term⟩ ⟨operator⟩ ⟨term⟩
               |  ⟨condition⟩ && ⟨condition⟩
               |  ⟨condition⟩ || ⟨condition⟩
               |  ( ⟨condition⟩ )
⟨operator⟩   ::=  << | > | >= | < | <= | == | !=
```

(b) La syntaxe simplifiée des conditions.

Fig. 4.1.: La syntaxe de nouveau **%match**.

L'opérateur `<<` dénote une contrainte de filtrage. Par exemple `p << t` est un problème classique de filtrage entre le motif `p` et le sujet `t`, comme on l'a défini dans la section 2.2.2 page 25. La partie gauche d'une contrainte de filtrage correspond à un motif dans l'ancienne construction **%match**. La partie droite est un terme pouvant contenir des variables Java, des appels de fonctions et des constructeurs Tom. La nouveauté par rapport aux sujets de la construction **%match** précédente est que maintenant elle peut aussi contenir des variables Tom qui se trouvent dans la partie

gauche d'une autre contrainte de filtrage. Par exemple, dans les contraintes suivantes : `Person(age,profession) << pers && Professor[] << profession`, la deuxième occurrence de la variable `profession` est instanciée par la première contrainte.

Les autres opérateurs ont la même signification que d'habitude. Les contraintes qui sont construites avec ces opérateurs sont appelées *contraintes booléenes*, parce que le résultat de leur évaluation est toujours *vrai* ou *faux*. Les membres gauches et droites de ces contraintes sont construits exactement de la même manière que les membres droits des contraintes de filtrage. Autrement dit, ce sont des termes clos.

Les `&&` et `||` sont les connecteurs booléens habituels, respectivement *and* et *or*. Les parenthèses sont utilisées pour combiner d'une manière naturelle les contraintes. Comme habituellement dans une expression logique, les parenthèses nous permettent de changer l'ordre d'évaluation de contraintes.

Note A. Pour garder la compatibilité entre l'ancienne construction **%match** et la nouvelle, en pratique nous utilisons une syntaxe qui combine les deux. Par exemple, les constructions **%match** suivantes ont la même sémantique :

```
%match(s1,s2) {
  p11,p12 && condition_1 -> { action_1 }
  p21,p22 || condition_2 -> { action_2 }
}

%match {
  p11 << s1 && p12 << s2 && condition_1 -> { action_1 }
  p21 << s1 && p22 << s2 || condition_2 -> { action_2 }
}
```

L'ancienne syntaxe est très pratique quand toutes les règles d'un **%match** filtrent vers les mêmes sujets. Elle permet de ne pas dupliquer les sujets dans toutes les contraintes de filtrage.

4.2.2. Les restrictions

La syntaxe des conditions présentée dans la figure 4.1b page précédente est très permissive. Par conséquent, des situations indésirables peuvent apparaître, où le comportement serait difficile à définir. Dans cette section, on présente deux restrictions qui sont requises pour l'utilisation des conditions. Un point important à noter est que toutes ces restrictions sont imposées de manière automatique et vérifiées par le compilateur.

Les variables dans les disjonctions

Pour pouvoir définir une sémantique claire de la partie action d'une règle, nous devons nous assurer que toutes les variables Tom venant de la partie gauche de la règle sont instanciées au moment de l'exécution de l'action. Dans la construction **%match** précédente, ce problème était naturellement résolu par le fait qu'on avait une seule contrainte

de filtrage. Le passage dans la partie droite de règle se fait seulement si une substitution pour *toutes* les variables du motif est trouvée. Avec la nouvelle syntaxe qui permet l'utilisation des disjonctions, nous ne sommes plus dans cette situation. Par exemple, la règle `f(x,y) << s1 || g(x) << s2 -> { return 'f(y,x) ; }` est refusée par le compilateur. Un message d'erreur indique que la variable `y` ne peut pas être utilisée si elle ne se trouve pas dans toutes les parties de la disjonction. Dans la suite, nous formalisons cette condition.

En utilisant les règles de transformation de la figure suivante, où c_i dénote une condition, les parenthèses d'une condition peuvent être éliminées. Par conséquent, toutes les conditions peuvent être vues comme des disjonctions de conjonctions. Nous définissons

$$\begin{array}{lll} c_1 \texttt{ \&\& (} c_2 \texttt{ || } c_3 \texttt{)} & \mapsto\!\!\!\!\twoheadrightarrow & c_1 \texttt{ \&\& } c_2 \texttt{ || } c_1 \texttt{ \&\& } c_3 \\ c_1 \texttt{ \&\& (} c_2 \texttt{)} & \mapsto\!\!\!\!\twoheadrightarrow & c_1 \texttt{ \&\& } c_2 \\ & & \texttt{si } c_2 \text{ ne contient pas des disjonctions} \end{array}$$

FIG. 4.2.: Élimination des parenthèses dans une condition

l'ensemble des variables libres d'une condition normalisée en préalable avec le système ci-dessus de manière inductive :
- $\mathcal{FV}ar(p \texttt{ << } s) = \mathcal{FV}ar(p)$
- $\mathcal{FV}ar(c_1 \texttt{ \&\& } c_2) = \mathcal{FV}ar(c_1) \bigcup \mathcal{FV}ar(c_2)$
- $\mathcal{FV}ar(c_1 \texttt{ || } c_2) = \mathcal{FV}ar(c_1) \bigcap \mathcal{FV}ar(c_2)$

Les variables libres d'un motif p sont définies de manière habituelle (voir la définition 31 page 33 pour une présentation formelle). Les variables anonymes sont ignorées. Par exemple, $\mathcal{FV}ar(f(x, _)) = \{x\}$.

Avec cette définition, nous pouvons alors énoncer la règle suivante : *toutes les variables utilisées dans la partie action, provenant des motifs* TOM*, doivent se trouver parmi les variables libres de la condition.*

Ceci est une restriction qui empêche l'exécution de la partie droite d'une règle avec des variables potentiellement non-instanciées.

Les autres systèmes de règles n'ont pas tous le même comportement : par exemple, dans JESS aucune vérification n'est faite, et s'il y a une variable définie dans certains membres d'une disjonction, mais pas dans tous, une erreur peut intervenir à l'exécution (*runtime error*). Dans JBOSS RULES, une nouvelle règle est définie pour chaque membre d'une disjonction, ce qui permet de faire facilement cette vérification à la compilation et d'interdire ce genre de situations.

Les références circulaires

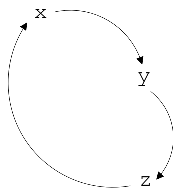
Un aspect très utile des nouvelles conditions est le fait qu'on puisse utiliser dans le sujet d'une contrainte de filtrage des variables TOM qui se trouvent dans la partie gauche d'une autre contrainte. Par exemple, étant donné l'objet `s`, la règle qui vérifie que cet objet est un `f` dont le premier fils est soit un `f(a,b)` ou bien un `g(x)` pourrait s'écrire en utilisant les nouvelles conditions de la façon suivante :

```
f(z,_) << s && ( f(a,b) << z || g(x) << z )  -> { action }
```

Comme nous pouvons remarquer dans cette construction, la variable `z` est une variable TOM, et elle est utilisée comme sujet pour d'autres contraintes de filtrage. Habituellement, le sujet d'une contrainte est une variable qui vient du JAVA, considérée comme un terme clos. Le fait qu'on autorise dans la partie droite des contraintes de filtrage en même temps des variables JAVA et TOM, peut rendre impossible de faire la différence entre les deux types. Par conséquent, nous pouvons écrire la contrainte suivante : `x << x`. Pour cette construction, nous n'avons aucune possibilité d'en déduire si la variable `x` à droite vient du JAVA, et dans ce cas on doit instancier celle de gauche à sa valeur, où elles représentent toutes les deux la même variable TOM et la contrainte peut se réduire à vrai. Donc le compilateur doit nous assurer que parmi les contraintes on n'a pas des références circulaires de ce type. Cette vérification doit marcher même quand les cycles sont moins évidents, comme par exemple dans l'expression suivante :

```
f(g(x),a()) << y && f(b(),y) << z && g(f(z,c())) << x
```

Pour la détection des références circulaires, nous construisons un graphe de dépendances entre les variables. Pour cela nous utilisons une approche classique, comme par exemple dans les problèmes d'unification [MM82]. Pour une condition c donnée, un graphe est construit pour chaque membre de la disjonction résultant de la normalisation de c avec les règles de la figure 4.2 page précédente. La construction du graphe est faite de la façon suivante : pour chaque contrainte de filtrage `p << t`, chaque variable libre de `p` est considérée comme un nœud, avec des arêtes vers toutes les variables de `t`. Si sur un de ces graphes ainsi construits le compilateur détecte un cycle, l'utilisateur recevra un message d'erreur indiquant la variable pour laquelle une référence circulaire a été trouvée. Par exemple, pour la contrainte présentée ci-dessus le graphe suivant est construit :



4.2.3. L'exécution des actions

Comme dans le cas des règles de production, une action est exécutée quand les conditions de la partie gauche sont satisfaites. Pour mieux comprendre quand une condition est satisfaite, on va la considérer dans la forme normale par rapport aux règles de transformation de la figure 4.2 page ci-contre. Ces règles éliminent les parenthèses et mettent la condition en forme normale disjonctive. Étant donnée cette forme, l'action d'une règle est exécutée pour chaque substitution qui rend la disjonction vraie, *i.e.* au moins un de ses membres. En fait, on pourrait voir chaque membre d'une disjonction comme étant une règle distincte ayant la même action. Ce comportement est similaire avec celui de JESS ou JBOSS RULES.

Une remarque importante est le fait que chaque variable utilisée dans une condition est considérée comme une variable TOM si elle apparaît au moins une fois dans la partie gauche d'une contrainte de filtrage, et comme une variable JAVA dans le cas contraire. Par exemple, dans l'expression suivante `x`, `y` sont des variables TOM instanciées par le filtrage, tandis que `z` est une variable JAVA :

```
f(x,a()) << someJavaFunction(y) && f(b(),y) << z
```

Exemple 16. Un exemple élémentaire pour le fonctionnement des disjonctions. On donne la signature GOM dans la partie gauche, et on considère un **%match** qui utilise cette signature :

`Term =`
`  | a()`
`  | b()`
`  | c()`
`  | g(t:Term)`
`  | f(l:Term,r:Term)`
	`%match(s) {`
	`  a() || b() << s          -> { print("a() or b()"); }`
	`  a() || a() << s          -> { print("a()"); }`
	`  f(b(),_) || g(b()) << s  -> { print("f or g of b()"); }`
	`  f(x,_) || g(x) << g(c()) -> { print("f or g, x=" + ‘x); }`
	`}`

Dans cet exemple, étant donné le sujet `s` avec la valeur `a()`, le code produit la sortie suivante :

```
a() or b()
a()
a()
f or g, x=c()
```

Nous pouvons remarquer que la deuxième action a été exécutée deux fois, une fois pour chaque partie de la disjonction évaluée à vrai. D'une manière similaire, pour le sujet `s` instancié avec `f(b(),c())`, le code afficherait :

```
f or g of b()
f or g, x=b()
f or g, x=c()
```

4.3. Synthèse

Les résultats présentés dans ce chapitre sont d'ordre pratique : toutes les conditions pour l'application d'une règle peuvent être mises dans la partie gauche, permettant d'avoir un code plus clair et plus concis. Toutes les notions de ce chapitre ont été implémentées et sont disponibles dans la version diffusée du langage TOM.

4.3.1. Plus d'expressivité et d'efficacité

Les résultats présentés dans ce chapitre permettent essentiellement d'obtenir une meilleure expressivité du filtrage. Avec ces nouvelles conditions, l'exemple présenté dans la section 4.1.2 page 64 pourrait s'écrire de la façon suivante :

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

Nous pouvons remarquer que l'action n'est plus dupliquée, et que toutes les conditions pour l'application de la règle sont dans la partie gauche.

Nous avons utilisé les nouvelles conditions pour réécrire certaines parties du code de Tom. La règle suivante est utilisée avant la génération du code pour mettre les conditions booléenes sur une variables le plus proche possible de la déclaration de cette variable. De cette manière l'efficacité s'améliore dans certains cas parce que le filtrage est arrêté plus tôt quand les conditions ne sont pas satisfaites. L'idée principale est que si on trouve dans une conjonction (`And`) une contrainte de filtrage (`MatchConstraint`) et plus loin dans la même conjonction une contrainte booléene (`BooleanConstraint`) qui utilise la variable qui se trouvent à gauche de la contrainte de filtrage, alors on essaie de mettre la contrainte booléene le plus proche possible de celle de filtrage. Pour plus d'informations le lecteur peut consulter le chapitre 5 page 75. Ici on présente juste l'encodage de la règle avec l'ancien formalisme et avec le nouveau :

```
And(X*,match@MatchConstraint[Pattern=pat],Y*,
        num@BooleanConstraint[Pattern=x,Subject=y],Z*) << subject -> {
 if (`pat == `x || `pat == `y) {
   %match(And(Y*)) {
     !And(_*,MatchConstraint[Pattern=x],_*) -> {
       %match(And(Y*)) {
         !And(_*,MatchConstraint[Pattern=y],_*) -> {
           return `And(X*,match,num,Y*,Z*);
         }
       }
     }
   }
 }
}
```

Les deux dernières conditions nous assurent que la condition booléene n'est pas mise avant la déclaration d'une de ses variables. Avec les nouvelles conditions, ce code peut se réécrire de la façon suivante :

```
And(X*,match@MatchConstraint[Pattern=pat],Y*,
        num@BooleanConstraint[Pattern=x,Subject=y],Z*) << subject
 && (pat << x || pat << y)
 && !And(_*,MatchConstraint[Pattern=x],_*) << And(Y*)
 && !And(_*,MatchConstraint[Pattern=y],_*) << And(Y*) -> {
  return `And(X*,match,num,Y*,Z*);
```

```
}
```

La deuxième approche a deux avantages : d'abord, l'intention du code est plus claire. L'action de la règle est séparée de ses conditions d'application, permettant de déduire plus facilement quand elle sera exécutée. Un autre inconvénient de l'ancienne approche est qu'il n'y a aucune liaison entre les **%match** imbriqués, et Tom ne peut pas faire des éventuelles optimisations (Tom ne parse pas les constructions du langage hôte, et chaque **%match** est compilé indépendamment). Par exemple, la liste `And(Y*)` est construite deux fois. Dans le deuxième encodage, le compilateur de Tom peut détecter cette double construction (puisqu'elles sont toutes les deux dans la même règle) et utiliser une variable intermédiaire.

Plus d'efficacité ?

L'utilisation des conditions booléenes peut aussi améliorer la vitesse d'exécution dans certains cas, en arrêtant le filtrage plus tôt. Par exemple, les deux constructions **%match** de la figure 4.3 produisent toutes les deux le même résultat (elles affichent les éléments d'une liste d'entiers qui apparaissent deux fois et qui sont plus grands que `5`), mais celle de gauche, figure 4.3a, est plus efficace que l'autre parce que la condition booléene (`x > 5`) est vérifiée juste après la première instanciation de la variable à laquelle elle est attachée (`x`), et le filtrage pour les autres éléments du motif ne continue pas si la condition n'est pas satisfaite.

```
%match(s) {
  list(_*,x_*,x,_*) && x > 5 -> {
    print('x);
  }
}
```

(a) Un **%match** avec contraintes booléenes

```
%match(s) {
  list(_*,x_*,x,_*) -> {
    if (x > 5) { print('x); }
  }
}
```

(b) Le codage des contraintes booléenes en Java

Fig. 4.3.: Un exemple d'utilisation des contraintes booléenes pour augmenter l'efficacité par rapport au code Java équivalent.

Par exemple, si `s=list(3,4,7,4,6,8,8}`, le fonctionnement du code avec la contrainte booléene est le suivant : il donne la valeur `3` à `x`, et pour continuer le filtrage il vérifie la condition `3 > 5` ; comme celle-ci est fausse, il donne à `x` la deuxième valeur `4`. Dans le cas sans contrainte de la figure 4.3b, les valeurs `3` et `4` sont d'abord vérifiés pour voir s'ils apparaissent deux fois dans la liste, et juste après, la condition `x > 5` est évaluée. Autrement dit, toute la liste est parcourue avant de vérifier la condition `x > 5`.

Il est, toutefois, nécessaire de faire un étude plus élaboré sur les dépendances entre les différents contraintes et quelles sont toutes les optimisations qui peuvent être faites, pour pouvoir évaluer sur des exemples plus complexes les gains de vitesse apportés par l'utilisation de conditions Tom par rapport aux conditions Java.

4.3.2. Les difficultés

Les difficultés sont plutôt d'ordre technique. Nous avons voulu fournir une syntaxe très permissive à l'utilisateur, pour lui laisser la possibilité d'écrire les conditions de manière naturelle, en se reposant sur le compilateur pour lui signaler les éventuelles incohérences.

Par exemple, le fait de permettre l'utilisation des variables TOM dans les parties droites des contraintes de filtrage, soulève un problème d'ordonnancement : on doit s'assurer que ces variables sont instanciées avant de les utiliser comme sujets. Une possibilité aurait été d'introduire un nouveau symbole, comme `;` par exemple, pour dénoter une séquence d'exécution. Mais dans ce cas, en plus de faire croître le nombre de symboles dans les motifs et rendre le code moins lisible, on aurait obligé l'utilisateur à réfléchir à l'ordre dans lequel il écrit ses contraintes. Nous avons choisi une autre variante, qui consiste à faire l'ordonnancement des contraintes d'une manière automatique au moment de la compilation. Par exemple, les contraintes qui déclarent certaines variables suite à leur décomposition, sont mises avant celles qui utilisent ces variables. Par conséquent, les deux constructions **%match** suivantes sont équivalentes : elles seront compilées dans du code JAVA identique, qui déclare et instancie `z` avant de l'utiliser pour filtrer contre `f(a(),_)` :

```
%match {
  g(z) << g(a()) && f(a(),_) << z -> { action; }
}

%match {
  f(a(),_) << z && g(z) << g(a()) -> { action; }
}
```

Les autres difficultés viennent de la nécessité d'imposer les restrictions de la section 4.2.2 page 67 de manière automatique.

Un aspect assez important est aussi le fait que l'action n'est pas dupliquée dans le cas des disjonctions. C'est important pour avoir un code généré dont la taille est linéaire par rapport à la taille des motifs. C'est aussi important pour fournir des messages d'erreur plus précis, en particulier des numéros de lignes corrects.

4.3.3. Perspectives

Nous envisageons plusieurs perspectives d'évolution de TOM ouvertes par cette approche. Une première direction est l'intégration de nouvelles contraintes, comme par exemple $z \in \mathsf{WM}$ (teste l'appartenance d'un objet à WM — la WM (*working memory*) est une collection de faits utilisée par les systèmes de règles de production). Cela nous permettrait, ayant une implémentation de WM, de pouvoir résoudre des problèmes comme ceux résolus aujourd'hui par les systèmes de règles de production. Les conditions booléenes, qui aujourd'hui ne sont que des simples tests, peuvent aussi devenir des vraies contraintes numériques, et TOM pourrait appeler un *solver* de contraintes comme ILOG CP pour leur résolution.

Une autre question intéressante est de savoir dans quelle mesure l'intégration dans la syntaxe du `not` logique est nécessaire, même si jusqu'à présent nous n'avons pas senti ce besoin. Est-ce que il y a des cas difficiles à exprimer avec les anti-patterns et où l'utilisation du `not` logique serait plus naturelle, vu que pour les conditions booléenes on fournit déjà l'inverse du chaque opérateur ? Si la réponse est positive, quel serait le résultat de l'application de l'opérateur `not` sur des problèmes de filtrage contenant des anti-patterns ? Comme les problèmes d'anti-filtrage se réduisent à des problèmes de disunification contenant des `not`, cette question ne devrait pas poser trop de difficultés.

L'utilisation de variables Tom dans les parties droites des contraintes de filtrage pose aussi des problèmes pour le vérificateur de type de Tom, qui n'a pas été conçu pour ces situations. Une propagation de types parmi les contraintes devrait être effectuée afin d'avoir un typage plus fin.

5. Compiling constraints with pluggable rewrite systems

TOM's pattern matching, although close to the one usually employed by functional or rewrite-based languages, has several particularities. On one hand it is quite complex (syntactic matching, associative matching with neutral elements, unlimited negations in patterns, *etc*) and on the other hand it is embedded in a host language upon which TOM doesn't have any information — no assumptions can be made about the availability of specific instructions (like *switch* or *jump*) and data structures are user-defined via mappings. These aspects make unsuitable classical compilation techniques.

We study in this chapter a way of compiling different combinations of complex match and anti-match constraints, both in the syntactic and associative case. The main goal is to obtain an implementation that is well structured, self-described and easy to extend in order to support new types of constraints.

5.1. Several approaches for compiling pattern matching

Pattern matching is a central notion in functional programming languages like for instance OCAML, SCALA, ML or HASKELL. It is also the main execution mechanism in rewrite based languages such as ASF+SDF, ELAN, MAUDE or Stratego. It is not surprising then that it has been extensively studied in both contexts. The methods for compiling pattern matching can be mainly divided in three categories: naive (or *one-to-one*) matchers (possibly optimized in a separate phase [Ses96]), algorithms based on decision trees [Car84, Grä91] and those that use backtracking automata [Aug85]. These last two approaches are also called *many-to-one*, because the compilation involves several rules and a subject. They allow to efficiently select the rule of the system that reduces the subject.

The typical pattern matcher takes as input a subject and a sequence of rules, producing as output the right-hand side of the first rule whose left-hand side matches, or fails (in TOM the patterns don't have to be exhaustive like for example in OCAML, and the failure just transfers the execution to the next instruction that follows the **%match** construct).

5.1.1. One-to-one pattern matching

The simplest way to compile pattern matching is to consider each rule independently. In other words, the first pattern is tried, if it fails, then the next one is tried, *etc*, until the subject matches some pattern or there is no more pattern left. This approach is

called *one-to-one*, because the matching only involves a subject and a rule at a given moment. It is similar to the naive string matchers, such as the ones presented for instance in [CD89].

As we may expect, this is very inefficient, because tests of the same subterms of the subject may be performed repeatedly. The worst case complexity is the product of the number of rules and the size of the subject.

Naive matchers can be optimized in a subsequent phase, resulting in a more efficient automaton. This is an approach employed by Sestoft in [Ses96] for compiling ML-style pattern matches. Sestoft's method first uses a naive matcher, then it instruments the matcher with positive and negative information about the term being matched, information obtained from all tests previously considered in the match. The result is a decision tree based compiler.

5.1.2. Decision trees

This is one of the first approaches to pattern matching compilation. It consists in transforming a sequence of patterns into a *decision tree*, *i.e.* a tree that encodes the patterns and defines the order in which the subterms of the subject have to be tested at run-time for determining which pattern matches. This method was described in [Car84] and later on in [Grä91].

The main advantage of this approach is that it guaranties that a subterm of the subject is never tested twice. Therefore a good run-time performance is obtained. Another advantage is that different checks (like non-exhaustive matchings or dead code) can be performed on the decision trees directly, since they are complete [Mar07]. The main disadvantage comes from the fact that the code produced may be exponential, given that patterns may be copied and compiled several times.

Optimizations for this method consist in reducing the number of nodes in the decision tree. They are mostly based on heuristics, as the problem of computing the minimal number of nodes is NP-hard. An early work is [BM85], also briefly presented in [AM87]. More recent experiments with different heuristics are presented in [SR00].

Examples of current compilers that use decision trees are SML/NJ [AM91], ELAN [KM01, Mor99] and F#. F# uses the generalized compilation algorithm of [SR00] with a left-to-right heuristic [SNM07].

5.1.3. Backtracking automata

Another well known approach is to compile the pattern matching problem into a backtracking automata, first presented by Augustsson in [Aug85]. This method consists in merging all the patterns that have the same outermost constructor, and to replace all the subpatterns with variables. Then the variables are further matched with the corresponding subpatterns in a similar way. The advantage over the naive method is obvious, as a lot of tests' duplications are avoided. This technique is used in languages such as Haskell-HBC and OCaml.

The main advantage of this method over the decision tree approach is that the patterns and actions are never copied, therefore the output code is linear in the size of the patterns. The main inconvenient is that the same subterm of the subject may be tested several times, yielding a potentially poor run-time performance. Several optimizations for this technique were developed in [FM01]. They mostly rely on sophisticated switch constructs, that are later compiled in more basic constructs like jump tables, exits and traps.

Huet and Lévy also defined a method for constructing match trees for non-ambiguous term rewriting systems [HL79], method employed later on by Puel and Suárez [PS93] for compiling pattern matching in lazy languages.

Other techniques for tree pattern matching are the ones issued from the works on algorithms for string matching, such as those presented by Hoffmann and O'Donnell [HO82]. Their goal was to find the position of all the subterms of a given subject term that are matched by one of the patterns. In [HO82] several methods are proposed, that consist in reducing the tree matching problem to string matching and then using well established techniques to solve this later one. These methods are not very appropriate for compiling pattern matching, mostly because this involves finding the first pattern that matches the *entire* subject. Moreover, the overhead of transforming the pattern matching problem into string matching wouldn't pay off comparing to other approaches that work directly with the tree structure, such as those exposed in the previous sections.

5.1.4. Tom: one-to-one matching + optimization

There are several particularities of Tom's pattern matching when compared to classical pattern matching in functional languages. These restrain Tom's compiler from using well established techniques such as decision trees and backtracking automata, presented in the previous sections.

The most important difference is that the use of mappings allows the terms to be represented in user-defined manner. Therefore the matching automata generated by the compiler has to be independent from the actual representation of the terms. Moreover, Tom has several target languages, which constraints the compiler to only use a sub-set of instructions that are available in all these host languages. Thus techniques such as the one presented for instance in [FM01], based on *switch* and *jump* instructions, couldn't be employed.

Another argument for not adopting a classical approach for compilation is that the pattern matching of Tom is more complex than in other languages. In particular, patterns can be syntactic or associative, linear or non-linear, and they can contain complements (anti-patterns). Moreover, new types of constraints should be easy to integrate.

The compilation method chosen in Tom is close to that of Sestoft [Ses96]. Left hand sides of the rules are first compiled independently in a naive, left-to-right fashion. The result is expressed in an intermediate language called `PIL`. Then, several optimizations are performed on the generated `PIL` code. The optimization phase is completely separated from the compilation one. More information are available in [BM06].

This approach has several advantages, the most notably being the fact that the compilation and optimization are completely separated. Consequently, the compiler gets a lot simpler and easier to extend. As we will further see in this section, this type of modularity was particulary helpful for the implementation of a completely new compiler, without being necessary to re-implement all the optimizations as well. This couldn't have been the case if the optimizations were mingled in the compilation phase.

5.2. Presentation of the Tom system

5.2.1. Global view of the system

The purpose of the Tom system is to transform a file that mixes host language constructs with those of Tom in a file that only contains host language code. The big picture of this functionality is given in the Figure 5.1.

The input is a Tom file containing a **%match** instruction that encodes the addition of the Peano integers using pattern matching. We suppose the following Gom signature for the integers:

```
Nat = zero()
    | suc(pred:Nat)
```

As we can see in this figure, what Tom system does is to transform each Tom construct from the input file in semantically equivalent code expressed in the host language, leaving untouched the host language code. Note that we consider Java as the host language in this chapter.

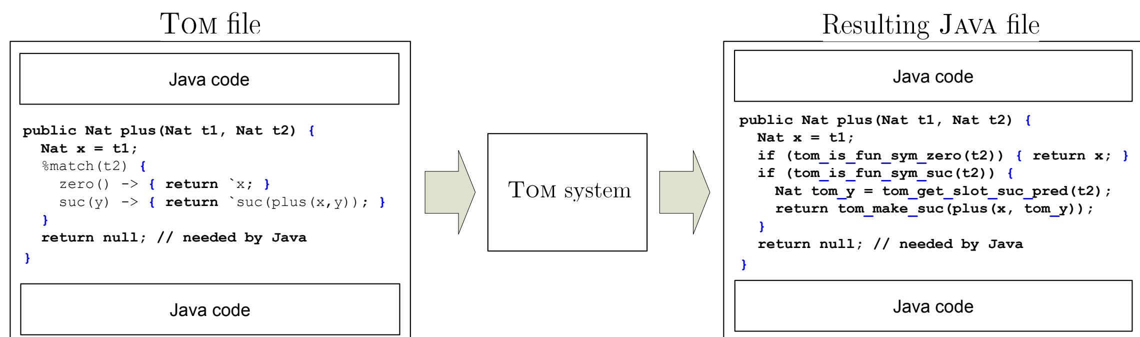


Figure 5.1.: A general view of the way Tom system works

In Figure 5.1 the Tom system is seen as a black box. Performing a zoom on it gives us the description presented in Figure 5.2 on the facing page. The system has a pipeline structure, where each phase transforms the given input and passes the result to the next one. We provide more details for each of these phases in the following.

- *Parser*: The parser produces an Abstract Syntax Tree (AST) that contains special nodes for the Tom constructs such as **%match**. The AST is built using Gom

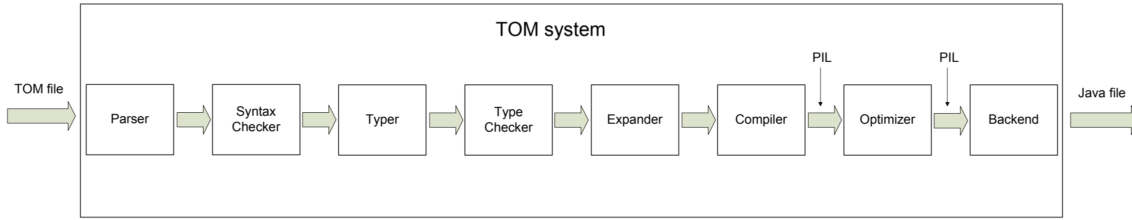


Figure 5.2.: Inside view of the TOM system

terms, which allow the following phases to easily perform matches and to use strategies to implement transformations on this tree.

- *Syntax Checker*: The syntax checker performs a series of verifications on the received AST, like for instance it verifies that an algebraic type is not declared twice. More complex verifications are those presented in Section 4.2.2: for example, given a **%match** construct, it ensures that there are no circular dependencies between match constraints.
- *Typer*: It types all the symbols and variables used, according to the types declared in the mappings.
- *Type Checker*: Performs several verifications concerning the types of the symbols, like for instance it checks that two occurrences of the same variable have the same type.
- *Expander*: Performs some transformations on the AST, preparing the compilation. One of the simplest is to change the name of the variables used in patterns to avoid conflicts with variables having the same name in the host language.
- *Compiler*: Its main mission is to locate the **%match** in the received AST, and to transform them into a series of instructions expressed in an intermediate language, called `PIL`. We will give more information about `PIL` in the Section 5.2.2 on page 81.
- *Optimizer*: Performs different optimizations on the `PIL` instructions received, like reducing the number of assignments and tests corresponding to a **%match**.
- *Backend*: Translates the `PIL` constructs into host language instructions. There are several back-ends, one for each host language (C, JAVA, OCAML, *etc*).

General design considerations

The use of Tom as a programming language. Most of the work of the TOM system involves traversing the AST of the input program and performing pattern matching on some of its nodes in order to analyze or to transform them. As this meets exactly the application domain of the language, most of the system can be expressed very naturally

in Tom. Therefore all the phases of the system are written in Tom (using Java as a host language).

PIL as a target language for the compiler. As we already mentioned, the compiler doesn't directly produce host language code. Its output is expressed using an intermediate language called PIL, presented in detail in the following sections. Using an intermediate language has several advantages:

- various host languages can be supported. Moreover, adding the support for another language consists only in adding another back-end, without any change in the other phases of the system.
- optimizations that are specific to pattern matching can be easily performed on this intermediate language (see [BM06] for more details).
- proofs can be much more easily generated for PIL, due to its simplicity, than for a general programming language like C or Java. The interested reader can find more information about certifying the syntactic pattern matching of Tom in [KMR05] or [Rei06b].

Representation of internal ASTs with Gom. Tom may manipulate very large terms, and it should be able to scale to any size of the input files, in a similar way as the Java compiler does for instance. Therefore the choice of the internal term storage of the system is crucial to the performance. This is why we chose Gom to represent the internal ASTs corresponding to the input files.

Usually the intermediate terms that are created during the compilation have lots of overlaps, therefore the maximal sharing feature of Gom is very important for memory saving. This ensures that new terms are created only if they don't already exist. Moreover, fast equality checks between terms are guaranteed, as only the pointers need to be compared and not the whole structure. Please see [Rei06a] and [Rei06b] for more details about Gom as well as for a comparison with similar tools such as ApiGen [dJO04, vdBMV05].

Another important benefit of using Gom to build the intermediate ASTs is that matchings and strategies come for free. Nothing has to be done in order to use **%match** and **%strategy** constructs with these structures, as the necessary Tom mappings as well as the support for the strategies are automatically generated by Gom.

5.2.2. Existing compiler: a brief description

In this section we describe the comportment of the existing Tom compiler. We characterize the input and output data and we briefly describe the compilation process. We further discuss the limitations of the current compilation scheme and we present the main ideas for an alternative approach.

This section is quite important for setting the framework of the new compiler as well. As we will further see, we are only interested in changing the compilation method,

without interfering with the input and output data. Therefore, the new compiler should handle the same input as the previous one (actually it will support a superset of it), and it will have to produce the same output.

The input data

The goal of the compiler is to produce PIL, described in detail in the next section, corresponding to the **%match** constructs in the TOM input files. As we previously mentioned, each rule in a **%match** is compiled independently. Therefore, we can consider that the input of the compiler is a list of tuples $\langle constraint, action\rangle$, where each of these tuples is compiled separately. The constraint can be either a syntactic matching problem or an associative matching one:

$$\langle constraint\rangle \quad ::= \quad \langle pattern\rangle \ \texttt{<<} \ \langle term\rangle$$

Here $\langle pattern\rangle$ denotes a TOM pattern, and $\langle term\rangle$ is a ground term built with constructors, JAVA variables, JAVA function calls and variables that are used in TOM patterns. It is actually the subject of a **%match** construction.

The PIL language

In order to be data-structure independent and to to be able to support several target languages, all TOM instructions, and consequently **%match**, are compiled into an intermediate language called PIL. The instructions of this language are enough to express the matching algorithms that we are interested in. Moreover, they are close to the ones in languages like JAVA, C or OCAML, allowing a direct translation.

We present the syntax of the PIL language in Figure 5.3 on the following page. $\mathbb{N}$ is the usual set of natural numbers and $\mathbb{B}$ represents the set of boolean values *true* and *false*. $\mathcal{F}$, $\mathcal{X}$ and $\mathcal{T}(\mathcal{F})$ are respectively the set of symbols, variables and ground terms.

In addition to being a ground term from $\mathcal{T}(\mathcal{F})$ or a variable, a term from $\langle term\rangle$ can be also built with the constructions:

- $\mathtt{subterm_f(t,i)}$: returns the $\mathtt{i}$ subterm of the term $\mathtt{t}$, given that the head symbol of $\mathtt{t}$ is $\mathtt{f}$.
- $\mathtt{get_head_f(t)}$: if $\mathtt{f}$ is an associative symbol, and the flattened form of $\mathtt{t}$ (as defined in Section 2.2.1 on page 25) is $\mathtt{f(t_1,\ldots,t_n)}$, then this construction returns $\mathtt{t_1}$.
- $\mathtt{get_tail_f(t)}$: if $\mathtt{f}$ is an associative symbol, and the flat form of $\mathtt{t}$ is $\mathtt{f(t_1,\ldots,t_n)}$, then this construction returns $\mathtt{f(t_2,\ldots,t_n)}$.
- $\mathtt{get_slice_f(t_{start},t_{end})}$: if $\mathtt{f}$ is an associative symbol, and the flattened forms of $\mathtt{t_{start}}$ and $\mathtt{t_{end}}$ are respectively $\mathtt{f(t_i,\ldots,t_n)}$ and $\mathtt{f(t_j,\ldots,t_n)}$, we distinguish the following cases (please note that always $\mathtt{i} \le \mathtt{j}$):
    1. $\mathtt{i} = \mathtt{j}$: this means that $\mathtt{t_{start}} = \mathtt{t_{end}}$, and $\mathtt{get_slice_f(t_{start},t_{end})}$ returns the empty list,

$$
\begin{array}{lll}
\texttt{PIL} & ::= & \langle instr \rangle \\
\textbf{variable} & ::= & x \in \mathcal{X} \\
\langle term \rangle & ::= & t \in \mathcal{T}(\mathcal{F}) \\
 & | & \textbf{variable} \\
 & | & \texttt{subterm}_\texttt{f}(\langle term \rangle, \texttt{n})\ (\texttt{f} \in \mathcal{F}, \texttt{n} \in \mathbb{N}) \\
 & | & \texttt{get_head}_\texttt{f}(\langle term \rangle)\ (\texttt{f} \in \mathcal{F}) \\
 & | & \texttt{get_tail}_\texttt{f}(\langle term \rangle)\ (\texttt{f} \in \mathcal{F}) \\
 & | & \texttt{get_slice}_\texttt{f}(\langle term \rangle, \langle term \rangle)\ (\texttt{f} \in \mathcal{F}) \\
\langle bexpr \rangle & ::= & b \in \mathbb{B} \\
 & | & \texttt{eq}(\langle term \rangle, \langle term \rangle) \\
 & | & \texttt{is_fsym}_\texttt{f}(\langle term \rangle) \\
 & | & \texttt{is_empty}_\texttt{f}(\langle term \rangle)\ (\texttt{f} \in \mathcal{F}) \\
\langle instr \rangle & ::= & \texttt{assign}\ \textbf{variable} = \langle term \rangle\ \texttt{in}\ \langle instr \rangle \\
 & | & \texttt{if}(\langle bexpr \rangle)\ \texttt{then}\ \langle instr \rangle\ \texttt{else}\ \langle instr \rangle\ \texttt{endif} \\
 & | & \texttt{do}\ \langle instr \rangle\ \texttt{while}(\langle bexpr \rangle) \\
 & | & \langle instr \rangle ; \langle instr \rangle \\
 & | & \texttt{hostcode}(\textbf{variable*}) \\
 & | & \texttt{nop}
\end{array}
$$

Figure 5.3.: The syntax of the intermediate language PIL.

2. $\texttt{i} < \texttt{j}$: $\texttt{get_slice}_\texttt{f}(\texttt{t}_\texttt{start}, \texttt{t}_\texttt{end})$ returns $\texttt{f}(\texttt{t}_\texttt{i}, \ldots, \texttt{t}_\texttt{j})$.

The set of expressions ⟨*bexpr*⟩ contains the boolean values, as well as the predicates: `eq` for comparing two terms, `is_fsym` that allows to check if the head symbol of a term is equal to a given symbol, `is_empty` for checking if a term is an empty variadic symbol, and `not` that corresponds to the classical negation. An instruction from the set ⟨*instr*⟩ can be:

- `assign`, which represents the assignment of a value to a variable. The third argument is the scope of the variable. This can be seen as a `let` instruction from OCaml, but it is not purely functional: the variable can be reassigned in its scope.
- `if`, which is the classical instruction *if-then-else*.
- `do`...`while`..., corresponding to the *do-while* instruction from imperative languages.
- a sequence of instructions.
- `hostcode`, a piece of host-code language (like Java), that may contain variables coming from Tom patterns. As we are only interested in compiling **%match** constructs, while ignoring the host language instructions, the `hostcode` instruction is an abstraction of the host language code. It only contains Tom variables, such that we can replace these variables with their corresponding value after the matching process. The rest of the host language code is stored in a string.

- the empty instruction, that doesn't do anything.

Note A. The following PIL instructions correspond to the predicates with the same names defined in the mappings: `subterm`, `get_head`, `get_tail`, `eq`, `is_fsym`, `is_empty`.

Compilation scheme

The compilation of the patterns is performed in a single step. As we mentioned before, it is a *one-to-one* approach that analyses the patterns from left to right using recursive calls to specific methods. These methods generate directly the corresponding PIL as they encounter the symbols in the patterns.

Examples of the generated code

In order to support the intuition of the correspondence between TOM constructs and PIL, we further give some examples of the generated PIL code for several **%match** instructions. For all these examples, we consider the following GOM signature:

```
Term = a()
     | b()
     | g(t:Term)
     | f(t1:Term,t2:Term)
     | list(Term*)
```

Example 17. Given the construct **%match** on the left column, the right column contains the corresponding PIL code, as generated by the compiler.

```
... Java code ...
...
%match(s) {
  a()      -> { print("a"); }
  g(b())   -> { print("g(b)"); }
  f(x,a()) -> { print("x␣=␣" + `x); }
}
...
... Java code ...
```

```
hostcode(...) ;
if(is_fsym_a(s)) then hostcode()
else nop endif ;
if(is_fsym_g(s)) then
  assign t_1 = subterm_g(s, 1) in
  if(is_fsym_b(t_1)) then hostcode()
  else nop endif
else nop endif ;
if(is_fsym_f(s)) then
  assign t_1 = subterm_f(s, 1) in
  assign x = t_1 in
  assign t_2 = subterm_f(s, 2) in
  if(is_fsym_a(t_2)) then hostcode(x)
  else nop endif ;
else nop endif ;
hostcode(...) ;
```

Let's take a closer look at the third rule: the pattern on the left side specifies that if the subject is an `f`, with the second subterm an `a`, then the value of the first subterm should be given to `x`. This is translated in a straightforward way into PIL: we test if the root symbol of the subject is `f`, and if true we instantiate an intermediate variable t_1

with the value of the first subterm, which we also use to instantiate x. We continue with testing that the second subterm is rooted by a, and if true we execute the action.

Example 18. In the case of the associative matching, loops are generated for a pattern that contains list variables (the ones suffixed by *). We consider a simple list pattern on the left column, and we give the corresponding PIL code on the right one.

```
... Java code ...
...
%match(s) {
  list(X*,g(x),Y*) -> {
    print("x␣=␣" + ‘x);
  }
}
...
... Java code ...
```

```
hostcode(...) ;
if(is_fsym_list(s)) then
  assign begin = s in
  assign end = s in
  do
    assign X* = get_slice_list(begin, end) in
    if(not(is_empty_list(end))) then
      assign v_1 = get_head_list(end) in
      if(is_fsym_g(v_1)) then
        assign t_1 = subterm_g(v_1, 1) in
        assign x = t_1 in
        assign Y* = get_tail_list(end) in
          hostcode(x)
      else nop endif
    else nop endif ;
    if(is_empty_list(end)) then
      assign end = begin            //to stop the loop
    else
      assign end  = get_tail_list(end)
    endif
  while(not(eq(begin, end)))
else nop endif ;
hostcode(...) ;
```

If we consider that `s = list(a(),g(b()))`, the above PIL code operates in the following way: it first checks that the head symbol of s is list, and it assigns s to two variables, begin and end. As X* is a sublist variable, and it should be given all the possible values, a loop is constructed. The variables begin and end have the same value at this first iteration, therefore X* is assigned the empty list list(). The variable v_1 is assigned the value a(), and then tested if its root symbol is g. It is not, so the control jumps to the last instruction in the loop, where end is assigned the value list(g(b())) (when end is equal with the empty list, this instruction ensures that the loop stops by assigning the value of begin to end, otherwise an *impossible* situation). With `end = list(g(b()))`, the second iteration of the loop gives the value a() to X*, v_1 gets g(b()), and further on `x = b()`, `Y* = list()`, and the action is executed.

Limitations

The limitations of the TOM compiler come from the way it is implemented, and not from the compilation method in itself. The arguments for a naive compilation, followed by an optimization phase presented in Section 5.1.4 on page 77 remain perfectly valid.

The compilation of each pattern is performed in a single step, from left to right, using recursive method calls. Although very efficient in terms of compilation time, this approach led to a compiler that was quite hard to extend. Even for implementing the slightest change in the pattern's syntax, one had to understand all the chain of recursive calls, and any modification to the code could have broken the compiler. The support for more complex left hand sides of the rules, such as those presented in Chapter 4, that include anti-patterns, conjunctions, disjunctions, *etc*, necessitated a tremendous effort, resulting in a code that was hard to understand and to maintain.

Ideas for improvement

The compilation method presented is too rigid for mainly two reasons. The first one is because compilation is performed in a single step. Therefore any modification to the input data leads to a modification to the compilation algorithm. Having an intermediate representation of the data would help to solve this problem: a modification to the input would only require a modification to the transformation to this intermediate representation. The second step, from the intermediate representation to `PIL`, wouldn't need to be modified. Therefore changes are easier to cope with, because a smaller part of code is concerned. We wouldn't have to modify the entire compilation algorithm.

The second reason for the lack of flexibility is that this approach is not modular. In an ideal case, adding something to the input data (a new type of constraint for instance) would only lead to *adding* some code that handles this to the compiler. We shouldn't have to modify the existing code in order to support new features. This is a well known principle in software development, and perhaps the most establish technique to implement the necessary modularity is the one of a platform based on *plug-ins*. In this scenario, each plug-in performs a specific task independently, without any awareness of the other plug-ins. The platform only launches the registered plug-ins. Therefore, when a new feature is required, a new plug-in is developed and plugged into the platform, without any modification to the other components or plug-ins of the system.

We advocate that a plug-in architecture for compiling pattern matching would lead to an extensible and easy to maintain system.

5.3. An approach based on rewriting and constraints

In this section we propose a new method for the compilation of complex pattern matching conditions, based on successive elementary transformations that are implemented using rewrite rules controlled by strategies.

The new compiler is placed in a well establish environment. As we didn't replace the entire system, the new compiler had to be able to replace the old one, without any changes on the other phases of the system. Therefore, he had to respect a series of requirements: the input that it can process had to be a superset of the old one and the output had to be `PIL` code. Moreover, the internal ASTs that it works on had to be represented with Gom.

In this context, our contribution was to conceive a new method for compiling different combinations of pattern matching conditions, together with a new architecture for the compiler. The new architecture had to allow a greater flexibility for supporting other types of constraints in the future. The main idea is to use a pack of rewrite rules for each type of transformation, and to control their application with strategies. For instance, a set of rules can handle the decompositions needed for syntactical matching, another one for the associative one, two others to generated the `PIL` code in each case and so on. Therefore, when we want to add a new construct to the language, it is enough to plug a new pack of rules that can handle this new construct and generate the necessary code.

The use of strategies is crucial because it ensures that each set of rules works independently. It allows processing certain nodes of the AST wherever they are in the tree, thus independently from the context. This enables the rules to work without requiring any update when the context changes, like for instance when adding a new type of input constraint. Therefore a high modularity can be obtained by having separate packets of rules for separate tasks, without any interference at all among them. When adding a new set, no modification to the existing rules is required.

5.3.1. General architecture of the compiler

A general schema of the internal architecture of the Tom compiler is presented in the following figure. As in the case of the system itself, the compiler has also a pipeline architecture. The main idea is that the input conditions are decomposed in smaller entities by the propagators (one for each type of constraint), until a normal form is reached. The result is just a set of really basic constraints, like for instance assignments of variables, tests for the head symbol of terms or equality tests. These entities pass to the scheduler which is making sure that they are in the good order before the generation of code. The generator is translating these constraints into semantically equivalent `PIL` code. A simple example illustrating the compilation steps is given in Figure 5.5 on the next page.

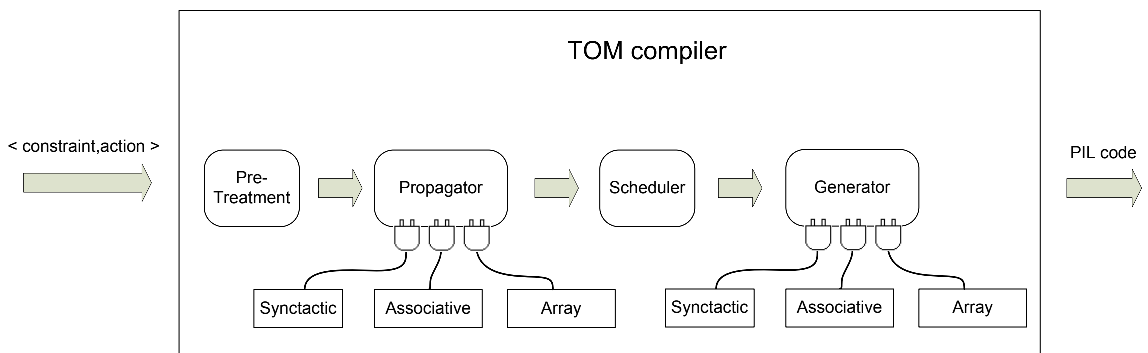


Figure 5.4.: The general architecture of the Tom compiler

We would like to emphasize two main characteristics of the compiler:

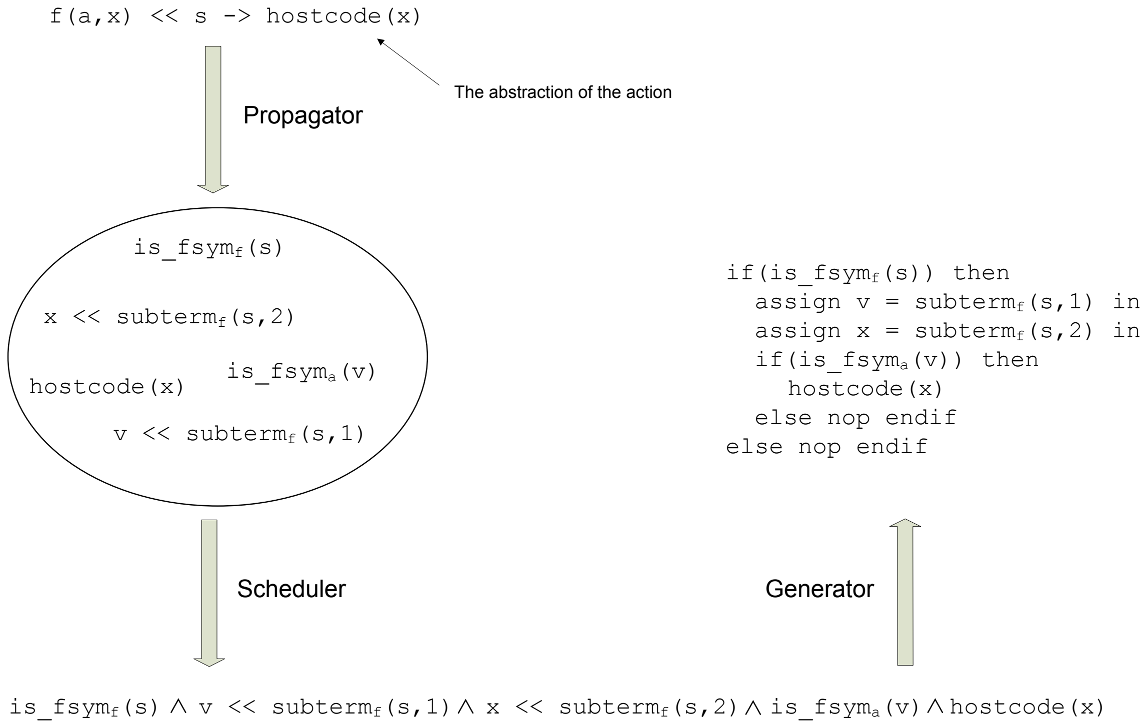


Figure 5.5.: A simple example of the compilation process. The propagator decomposes the initial problem in smaller constraints, which are then passed to the scheduler that arranges them in the good order. The generator further translates these constraints into `PIL` code.

1. Each of the phases of the compilation consists of several transformations of its input data. All these transformations are implemented in a declarative way, using rewrite rules controlled by strategies. In this way, as we will further see, the code has a great degree of legibility and it reflects perfectly its intention. Moreover, the rewrite rules as they are implemented have a straightforward correspondence with their formal description.

2. The propagator, as well as the generator, are actually platforms on which we can plug different modules. Each module can implement the transformations needed for a certain type of constraint, like for instance we have a plug-in that handles syntactical matching, one for associative one and so on. This allows extensions of the language to be implemented quite easily. For instance, adding a new type of condition would only require the writing of a new propagator and a generator that are further plugged into the compiler, without interfering at all with the other modules.

5.3.2. Input constraints

The input constraints are more complex than in the case of the old compiler. They can be a combination of syntactic matching problems, associative matching ones and boolean constraints, connected with the operators *and/or* — denoted by $\wedge$ respectively $\vee$, and eventually using parentheses for changing the evaluation priority. A simplified abstract syntax is given in the following figure, which corresponds to the syntax of the input conditions presented in Chapter 4, Figure 4.1 on page 66. These are the types of constraints that are currently supported, but the architecture of the compiler should be easy to extend in order to support other types as well.

$$
\begin{array}{lcl}
\langle constraint \rangle & ::= & \langle pattern \rangle\ \langle operator \rangle\ \langle term \rangle \\
 & & \mid \langle constraint \rangle \wedge \langle constraint \rangle \\
 & & \mid \langle constraint \rangle \vee \langle constraint \rangle \\
 & & \mid \texttt{(}\ \langle constraint \rangle\ \texttt{)} \\
\langle operator \rangle & ::= & \texttt{<<} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{==} \mid \texttt{!=}
\end{array}
$$

Figure 5.6.: The simplified abstract syntax of the compiler's input conditions. $\langle pattern \rangle$ is a TOM pattern, and $\langle term \rangle$ is a term built with constructors, JAVA variables, JAVA function calls and variables that are used in TOM patterns.

The operator `<<` denotes a matching constraint, while the other operators have the usual signification. They are trivially translated in host code instructions without any special treatment.

5.3.3. Decomposition and propagation

This phase is in charge with the decomposition of the input conditions into very simple, basic constraints, that have a straightforward translation to `PIL` instructions. Here is also where the propagation of variables for non-linear conditions is performed.

This stage of the compiler is organized as a platform that has as mission to launch the registered plug-ins. Each plug-in is in charge with the decomposition and transformation of one type of constraint. All the registered plug-ins are launched one after the other until no one performs any transformation anymore, *i.e.* the input data has reached a normal form *w.r.t.* the rules of all the plug-ins.

We suppose that the input constraints of this phase no longer contain any aliases or anonymous variables in the patterns (thanks to the pre-treatment phase, presented later in this chapter). We detail in the following sections the plug-ins for the syntactic and associative matching (the one for array matching is very similar to the associative one) and we present the syntax of the output constraints for the propagator.

Propagator for the syntactic matching

This plug-in is in charge with the decompositions needed for matching syntactic patterns. It contains the transformation rules presented in the following figure. We use $C[constraint]$ to denote any context in which the constraint *constraint* may appear, $\mathtt{x}$ denotes a variable, $\mathtt{f}$ a non-associative symbol, $\mathtt{p_i}$ a pattern and $\mathtt{t_i}$ any term.

$$
\begin{array}{lll}
\mathsf{Decompose}_1: & \mathtt{f}(\mathtt{p}_1,\mathtt{p}_2,\ldots,\mathtt{p}_n) \ll \mathtt{t} & \twoheadrightarrow \mathtt{is_fsym_f}(\mathtt{t}) \wedge \mathtt{p}_1 \ll \mathtt{subterm_f}(\mathtt{t},1) \wedge \\
& & \quad \mathtt{p}_2 \ll \mathtt{subterm_f}(\mathtt{t},2) \wedge \ldots \wedge \mathtt{p}_n \ll \mathtt{subterm_f}(\mathtt{t},n) \\
& & \quad \text{if } f \text{ is not an associative symbol} \\
\mathsf{Replace}_1: & \mathtt{x} \ll \mathtt{t}_1 \wedge C[\mathtt{x} \ll \mathtt{t}_2] & \twoheadrightarrow \mathtt{x} \ll \mathtt{t}_1 \wedge C[\mathtt{eq}(\mathtt{x},\mathtt{t}_2)] \\
\mathsf{AntiMatch}: & !\mathtt{p} \ll \mathtt{t} & \twoheadrightarrow \mathtt{anti}(\mathtt{p} \ll \mathtt{t})
\end{array}
$$

Figure 5.7.: Transformation rules of the syntactic propagator

As we may notice, the rule $\mathsf{Replace}_1$ is not actually doing a replacement, as in usual algorithms, like for instance the one we presented in Section 3.4.1 on page 39. It is only replacing other match constraints having the same variable as left-hand side with equality tests. It is both an optimization and a way to avoid some unexpected side effects than could have occurred from the double construction of the terms if we propagated to something like $C[\mathtt{eq}(\mathtt{t}_1,\mathtt{t}_2)]$ (t_1 can be built for instance from function calls in the host language).

This approach works due to the linear nature of the code that will be generated, where the instantiation of a variable always precedes other uses of this variable. The right ordering of the constraints before generating the PIL is handled by the scheduler, described in detail in Section 5.3.4 on page 92. For example, the constraints $\mathtt{x} \ll \mathtt{t}_1 \wedge \mathtt{eq}(\mathtt{x},\mathtt{t}_2) \wedge \ldots$ and $\mathtt{eq}(\mathtt{x},\mathtt{t}_2) \wedge \mathtt{x} \ll \mathtt{t}_1 \wedge \ldots$ will be both transformed into the following PIL code:

```
assign x = t1 in
  if(eq(x, t2)) then
    ...
  else nop endif
```

Example 19. Considering the input $\mathtt{f}(\mathtt{x},!\mathtt{x}) \ll \mathtt{s}$, the rules of the syntactic propagator are applied in the following way:

$$
\begin{array}{l}
\mathtt{f}(\mathtt{x},!\mathtt{x}) \ll \mathtt{s} \twoheadrightarrow_{\mathsf{Decompose}_1} \mathtt{is_fsym_f}(\mathtt{s}) \wedge \mathtt{x} \ll \mathtt{subterm_f}(\mathtt{s},1) \wedge !\mathtt{x} \ll \mathtt{subterm_f}(\mathtt{s},2) \\
\quad \twoheadrightarrow_{\mathsf{AntiMatch}} \mathtt{is_fsym_f}(\mathtt{s}) \wedge \mathtt{x} \ll \mathtt{subterm_f}(\mathtt{s},1) \wedge \mathtt{anti}(\mathtt{x} \ll \mathtt{subterm_f}(\mathtt{s},2)) \\
\quad \twoheadrightarrow_{\mathsf{Replace}} \mathtt{is_fsym_f}(\mathtt{s}) \wedge \mathtt{x} \ll \mathtt{subterm_f}(\mathtt{s},1) \wedge \mathtt{anti}(\mathtt{eq}(\mathtt{x},\mathtt{subterm_f}(\mathtt{s},2)))
\end{array}
$$

Propagator for the associative matching

Decomposing the constraints for the associative matching requires only the rule presented in the following figure. We consider as in the case of the syntactic matching that $\mathtt{p_i}$ are

patterns, $\mathtt{t}$ a term, and $\mathtt{begin_j}$, $\mathtt{end_j}$, $\mathtt{v_i}$ fresh variables. The term that is decomposed is in flattened form, and $\mathtt{p_i}$ are not sublist variables.

$$
\begin{aligned}
\textsf{Decompose}_2:\ \mathtt{f(p_1,\ldots,p_i,X^*,\ldots,p_n) << t} \ \mapsto\!\!\!\rightarrow\ & \mathtt{is_fsym_f(t)} \wedge \mathtt{v_1 << t} \wedge \\
& \mathtt{p_1 << get_head_f(v_1)} \wedge \mathtt{v_2 << get_tail_f(v_1)} \wedge \\
& \ldots \wedge \mathtt{p_i << get_head_f(v_i)} \wedge \mathtt{v_{i+1} << get_tail_f(v_i)} \wedge \\
\mathtt{begin_j << v_{i+1}} \wedge \mathtt{end_j << v_{i+1}} \wedge\ & \mathtt{X^* << variableHeadList_f(begin_j,end_j)} \wedge \\
& \mathtt{v_{i+2} << end_j} \wedge \\
\ldots \wedge \mathtt{p_n << get_head_f(v_n)} \wedge\ & \mathtt{v_{n+1} << get_tail_f(v_n)} \wedge \mathtt{is_empty_f(v_{n+1})} \\
& \text{if } \mathtt{f} \text{ is an associative symbol}
\end{aligned}
$$

Figure 5.8.: Transformation rule of the associative propagator

In order to build the loops needed for the associative matching, we introduced the term `variableHeadList`. A constraint like for instance $\mathtt{X^* << variableHeadList_f(t_1,t_2)}$ will lead to the generation of a loop that instantiates $\mathtt{X^*}$ with the term returned by the PIL instruction $\mathtt{get_slice_f(t_1,t_2)}$, where $\mathtt{t_2}$ varies in the generated loop from its initial value to the empty list. For example, $\mathtt{X^* << variableHeadList_f(f(a,b),f(a,b))}$ is transformed by the generator in a loop that gives $\mathtt{X^*}$ the values $\mathtt{f()}$, $\mathtt{f(a)}$, and respectively $\mathtt{f(a,b)}$.

There is no need for rules that handle the anti-patterns or the replacement of variables, because they are already taken care of in the syntactic propagator. Rules don't need to be duplicated, because the plug-ins are called one after the other until no more changes occur.

Note B. An interesting observation about the rule in this propagator is that in the case the pattern doesn't contain any list variables (suffixed by `*`), there is no `variableHeadList` created by the decomposition. This means that in the resulted `PIL` code, no loop will be generated. The pattern is basically treated as a syntactic one, with the only difference that the access to the subterms is performed using `get_tail` and `get_head` methods instead of `subterm`. This is similar to the result of the pre-processing phase of the ASF+SDF compiler [vdBHKO02], which makes sure that list patterns containing only a list variable or no list variables at all never cause backtracking.

Output constraints

The output of the propagator is a constraint with the syntax given in Figure 5.9 on the next page. The constraint `anti(constraint)` denotes an *anti-constraint*, coming from the decomposition of a match equation that contains an anti-pattern. The use of the term `variableHeadList` was detailed in the previous section. The rest of the constraints are related to the `PIL` instructions presented in Section 5.2.2 on page 81.

$$
\begin{array}{lcl}
\langle constraint \rangle & ::= & \langle term \rangle\ \langle operator \rangle\ \langle term \rangle \\
 & & \mid \texttt{anti}(\langle constraint \rangle) \\
 & & \mid \langle constraint \rangle \wedge \langle constraint \rangle \\
 & & \mid \langle constraint \rangle \vee \langle constraint \rangle \\
 & & \mid \texttt{(}\ \langle constraint \rangle\ \texttt{)} \\
 & & \mid \langle boolconstraint \rangle \\
\langle term \rangle & ::= & t \in \mathcal{T}(\mathcal{F}) \\
 & & \mid x \in \mathcal{X} \\
 & & \mid \texttt{subterm}_{\texttt{f}}(\langle term \rangle, \texttt{n}) \\
 & & \mid \texttt{get_head}_{\texttt{f}}(\langle term \rangle) \\
 & & \mid \texttt{get_tail}_{\texttt{f}}(\langle term \rangle) \\
 & & \mid \texttt{variableHeadList}_{\texttt{f}}(\langle term \rangle, \langle term \rangle) \\
\langle operator \rangle & ::= & \texttt{<<} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{==} \mid \texttt{!=} \\
\langle boolconstraint \rangle & ::= & \texttt{eq}(\langle term \rangle, \langle term \rangle) \\
 & & \mid \texttt{is_fsym}_{\texttt{f}}(\langle term \rangle)
\end{array}
$$

Figure 5.9.: Propagator's output syntax. $\texttt{f} \in \mathcal{F}$, $\texttt{n} \in \mathbb{N}$.

Note C. If the propagator is complete, *i.e.* it contains a plug-in for handling any allowed type of input constraint, the $\texttt{t}_1$ in the output constraints of the type $\texttt{t}_1 \texttt{ << } \texttt{t}_2$ should always be a variable.

Example 20. In this example we use most of the output constraints presented. We give the output of the propagation phase for a given input in order to support the intuition about the constraints that are used. The rules that perform these transformations are those presented in the previous sections. The translation of this constraints to `PIL` code is carried out by the generators and it will be detailed in the Section 5.3.6 on page 94.

Let's consider the following input condition given in a **%match**, where `getNumber` is a host language function that returns an integer:

```
list(X*,!g(x),y,Y*) << s && 3 > getNumber(y)
```

For this input, the propagator outputs the following constraint ($\texttt{v}_\texttt{i}$, $\texttt{begin}_1$ and $\texttt{end}_1$ are fresh variables):

$$
\begin{array}{l}
\texttt{is_fsym}_{\texttt{list}}(\texttt{s}) \wedge \texttt{v}_1 \texttt{ << s} \wedge \\
\texttt{begin}_1 \texttt{ << } \texttt{v}_1 \wedge \texttt{end}_1 \texttt{ << } \texttt{v}_1 \wedge \texttt{X}^* \texttt{ << } \texttt{variableHeadList}_{\texttt{list}}(\texttt{begin}_1, \texttt{end}_1) \wedge \\
\texttt{v}_2 \texttt{ << } \texttt{end}_1 \wedge \texttt{v}_3 \texttt{ << } \texttt{get_head}_{\texttt{list}}(\texttt{v}_2) \wedge \\
\texttt{anti}(\texttt{is_fsym}_{\texttt{g}}(\texttt{v}_3) \wedge \texttt{x << } \texttt{subterm}_{\texttt{g}}(\texttt{v}_3, 1)) \wedge \\
\texttt{v}_4 \texttt{ << } \texttt{get_tail}_{\texttt{list}}(\texttt{end}_1) \wedge \texttt{y << } \texttt{get_head}_{\texttt{list}}(\texttt{v}_4) \wedge \texttt{Y}^* \texttt{ << } \texttt{get_tail}_{\texttt{list}}(\texttt{v}_4) \wedge \\
\texttt{3 > getNumber(y)}
\end{array}
$$

5.3.4. Scheduling

The constraints, as issued by the propagator, need to be arranged in the proper order before they are transformed into `PIL` code. For instance, the constraints that will be transformed in variable's declarations should always precede constraints that represent other uses of those variables. A simple example is the expression $\ldots \texttt{eq}(\texttt{x}, \texttt{t}_2) \wedge \texttt{x << t}_1 \ldots$ that should be transformed into $\ldots \texttt{x << t}_1 \wedge \texttt{eq}(\texttt{x}, \texttt{t}_2) \ldots$ before the generation phase.

Handling the ordering of the constraints in a separate phase instead of doing it in the propagators has several advantages, among which the most important are:

- The code in the propagators is simpler, and easier to write. When writing a new propagator, the user doesn't have to worry about introducing a rule that accidentally switch some constraints. Moreover, the rules that she writes can have some sides effects from the interactions with the other propagators, effects that are not very easy to predict. Therefore, we feel that is better to restrict the role of the propagators to only decompose constraints into the smallest possible entities.
- Some optimizations are very easy to test and to implement with this approach. In some cases, it is enough to add a rule that rearranges the constraints to obtain interesting speed gains of the generated code. One example is putting the numeric conditions as close as possible to the last assignments of the variables that they contain, as we described in the Section 4.3.1 on page 72.
- It also arranges the constraints if the user didn't wrote them in the good order. Even if we take care of not switching the constraints in the propagators, the user may have provided the constraints in the bad order (either from a mistake, or perhaps having a different order was more natural or legible when she wrote the code).

We use the notation $c_1 \succ c_2, \ldots, c_n$ to specify that the constraint c_1 should precede the constraints $c_2, \ldots, c_n$. The scheduling phase currently makes sure that the following ordering is respected in the conjunctions (we use _ to denote any value, `x` a variable and `t` a term):

- $\texttt{is_fsym}_\texttt{f}(\texttt{t}) \succ _ \texttt{ << subterm}_\texttt{f}(\texttt{t}, _), _ \texttt{ << get_head}_\texttt{f}(\texttt{x})$: we should check for the head symbol first before trying to get a subterm or the head of a list.
- $_ \texttt{ << get_head}_\texttt{f}(\texttt{x}) \succ _ \texttt{ << get_tail}_\texttt{f}(\texttt{x})$: only if we succeeded in getting the head of the list we try to get the tail as well (this is because the `PIL` corresponding to `get_head` will test if the list is not empty).
- $\texttt{x << } _ \succ \texttt{is_fsym_}(\texttt{x}), \texttt{is_empty_}(\texttt{x}), _ \texttt{ << } C[\texttt{x}], \texttt{variableHeadList_}(_, \texttt{x}), \texttt{variableHeadList_}(\texttt{x}, _), \texttt{eq}(\texttt{x}, _)$: a variable should be first instantiated before being used.
- $_ \texttt{ << variableHeadList_}(_, \texttt{x}) \succ _ \texttt{ << x}$: uses of the *end* variables should be inside the generated loop that affects this variable.

- `_ << _` $\succ$ `anti(_)`: an *anti* should always be at the end, as it only verifies additional conditions, without instantiating new variables. Moreover, in the case of non-linear patterns it may perform some equality checks on variables that we need to had been instantiated before.

This ordering is implemented for the moment in a quite naive fashion: one rule is defined for each of the above conditions, and if it finds two constraints that do not respect the proper order, it simply switches their places. The process continues until no further switch is performed. There are no conflicts among these precedences (we do not have $c_1 \succ c_2$ and $c_2 \succ c_1$), which allows the ordering to terminate. A formal proof for termination, along with other ideas for improvement are envisaged as a future work and discussed more in detail at the end of this chapter.

Note D. Non-linear patterns are a particularity of Tom. They are not present in functional languages like ML or Haskell. To cope with this restriction, the usual technique is to use fresh variables instead of further occurrences of the same variable in the pattern. Then, after the instantiation of these variables by the matching, their equality is checked before executing the action. This method can be quite inefficient, because the pattern is completely matched before verifying the non-linearity conditions. Our approach ensures the verification of non-linear conditions as soon as it is possible in the matching process.

5.3.5. Pre-treatment of conditions

Several operations are performed on the conditions before they get to the propagator, both for optimization reasons and for simplifying the code of the next phases.

Abstraction of subjects

The subjects of the matching constraints are abstracted with fresh variables. For example, the input condition `p << s` is transformed into $\mathtt{v}_1$ `<< s` $\wedge$ `p <<` $\mathtt{v}_1$, where $\mathtt{v}_1$ is a fresh variable. It is an optimization avoiding that the subjects are constructed several times in the resulting code. Let's consider for instance the resulted `PIL` code for the compilation of the **%match** given on the left side:

```
%match(f(a(),b())) {
  g(x) -> { ... }
  y -> { ... }
}
```

```
if(is_fsym_g(f(a(),b()))) then
  assign t_1 = subterm_g(f(a(),b()),1) in
  assign x = t_1 in
    hostcode(...)
else nop endif ;
assign y = f(a(),b()) in
  hostcode(...)
```

As we can notice, the term `f(a(),b())` is constructed three times, instead of once. Abstracting the subject as we mentioned above gives us the following `PIL` code:

```
assign v1 = f(a(),b()) in
if(is_fsym_g(v1)) then
  assign t1 = subterm_g(v1,1) in
  assign x = t1 in
    hostcode(...)
else nop endif ;
assign y = v1 in
  hostcode(...)
```

Detaching of aliases

The aliases in TOM patterns, introduced by `@`, are used to assign names to certain subterms of the subject, in order to use them in the action part of the rules. In the compiled code, they are generated as variables and instantiated with the right subterms. For simplifying the way they are processed, we abstract the subterms that have aliases with fresh variables. For example, the match constraint $\texttt{f(a@b@g(x),y)} \ll \texttt{s}$ is transformed into $\texttt{f}(\texttt{v}_1,\texttt{y}) \ll \texttt{s} \wedge \texttt{g(x)} \ll \texttt{v}_1 \wedge \texttt{a} \ll \texttt{v}_1 \wedge \texttt{b} \ll \texttt{v}_1$, where $\texttt{v}_1$ denotes a fresh variable.

5.3.6. Code generation

The input of this phase consists in a tuple <*constraint,action*>, where *constraint* has the syntax presented in the Figure 5.9 on page 91. The goal is to transform this constraint into PIL code, such that if *constraint* is satisfied, the *action* is executed. Each of the basic output constraints corresponds to a sequence of tests, therefore given the constraints $c_1 \wedge c_2 \wedge \ldots \wedge c_n$, we want to generate a PIL code such that the constraint c_2 is evaluated if c_1 is satisfiable, c_3 if c_2 is satisfiable and so on (here *satisfiable* only means that the corresponding tests have positive results). We finally obtain a sequence of nested *if-then-else* instructions that execute the *action* if all the conditions are evaluated to *true*.

The generator respects exactly the same architecture as the propagator. Each plug-in handles the transformation of certain constraints, and they are launched one after the other until all input constraints are transformed into PIL code. We provide in Figure 5.10 on the facing page the correspondence between each type of constraint and PIL instructions. `x`, `y`, `b`, `e` denote variables, `f` a symbol, $\texttt{t}_\texttt{i}$ terms and c_i constraints. Please note that the *action* can be considered as the last constraint in a conjunction. The transformations described are performed on the whole AST, independently from the context.

Most of the correspondences between the constraints and the PIL code are straightforward. We detail in the following those that need some explanations.

The purpose of the `if` condition at the end of the `do`...`while`...instruction is to stop the execution of the loop if `e` is equal to the empty list. This is performed by assigning to `e` the value of `b`, otherwise an impossible situation.

The code generated for anti-patterns is a consequence of the algorithm presented as a conjecture in Section 3.5.3 on page 52. The reason why we do not have the

constraint	PIL
x << t $\wedge\, c$	assign x $=$ t in c
x << subterm$_{\mathtt{f}}$(t, i) $\wedge\, c$	assign x $=$ subterm$_{\mathtt{f}}$(t, i) in c
x << get_head$_{\mathtt{f}}$(t) $\wedge\, c$	if(not(is_empty$_{\mathtt{f}}$(t))) then assign x $=$ get_head$_{\mathtt{f}}$(t) in c else nop endif
x << get_tail$_{\mathtt{f}}$(t) $\wedge\, c$	assign x $=$ get_tail$_{\mathtt{f}}$(t) in c

x << variableHeadList$_{\mathtt{f}}$(b, e) $\wedge\, c$	do
	  assign x $=$ get_slice$_{\mathtt{f}}$(b, e) in c ;
	  if(is_empty$_{\mathtt{f}}$(e)) then
	    assign e $=$ b
	  else
	    assign e $=$ get_tail$_{\mathtt{f}}$(e)
	  endif
	while(not(eq(b, e)))

t$_1$ op t$_2$ $\wedge\, c$, where op is one of $>, \geq, <, \leq, ==, !=$	if(t$_1$ op t$_2$) then c else nop endif

anti(c$_1$) $\wedge\, c$	assign b $=$ false in
	if(c$_1$) then
	  assign b $=$ true
	else nop endif ;
	if(not(b)) then c else nop endif ;

(c$_1$ $\vee$ c$_2$) $\wedge\, c$	if(c$_1$) then c else nop endif; if(c$_2$) then c else nop endif
eq(t$_1$, t$_2$) $\wedge\, c$	if(eq(t$_1$, t$_2$)) then c else nop endif
is_fsym$_{\mathtt{f}}$(t) $\wedge\, c$	if(is_fsym$_{\mathtt{f}}$(t)) then c else nop endif

Figure 5.10.: The correspondence between constraints and PIL instructions. If the constraint being transformed is the last in the conjunction, we can consider c as being the action of the rule.

non-linearity problem is due to the order the generated code is interpreted. We are no longer placed in an environment where we cannot assume anything on the order in which the expressions are evaluated. For instance, using the algorithm from Section 3.5.3 on page 52, the expression $f(x, \urcorner x) \prec\!\!\prec f(a, b)$ would be decomposed in $x \prec\!\!\prec a \wedge \urcorner x \prec\!\!\prec b$. This has no solution, because $\mathcal{S}ol(x \prec\!\!\prec a \wedge \urcorner x \prec\!\!\prec b) = \mathcal{S}ol(x \prec\!\!\prec a) \cap \mathcal{S}ol(\urcorner x \prec\!\!\prec b) = \{x \mapsto a\} \cap \emptyset = \emptyset$. This is contrary to the semantics, and is why we introduced the $Pure\mathcal{FV}ars$ restriction. When compiling anti-patterns, we no longer have this problem. Decomposing the initial equation gives us the constraint is_fsym$_{\mathtt{f}}$(f(a, b)) $\wedge$ x << subterm$_{\mathtt{f}}$(f(a, b), 1) $\wedge$ anti(eq(x, subterm$_{\mathtt{f}}$(f(a, b), 2))). The generated PIL for this constraint respects the semantics of anti-patterns, because the

values of the variables are implicitly propagated in their scope.

As we may notice, the method employed for handling the disjunctions leads to code duplication, as if there was a distinct rule having the same action for each member of the disjunction. This is how disjunctions are treated in most of the business rules systems. We are forced to adopt this technique because of the associative patterns that are compiled into loops, and thus the rest of the code, including the action, has to be inside the loop. When only syntactic patterns are involved, we do have a more efficient method that avoids code duplication, similar to the one exposed in [FM01], although not based on labeled *exit* constructs.

5.4. Evaluation and future work

5.4.1. Flexibility et legibility

The new compiler, initially developed only to support the classical pattern matching of Tom (as presented in Chapter 1 on page 7), proved to be extremely flexible when we added the support for the new constructs exposed in Chapter 4 on page 63. For instance, in order to allow the use of conjunctions nothing had to be done in the compiler, as conjunctions were already used internally. It was only a parsing matter. Also adding the disjunctions only involved updates of the generation phase.

Using rewrite rules that apply wherever it is necessary on the AST, thanks to strategies, allows the implementation of new transformations by simply adding new rules, without any modifications on the previous ones.

The resulted code is also very close to its formal specification. For instance, the rule AntiMatch from Figure 5.7 on page 89 is implemented in the following way in the compiler:

```
MatchConstraint(AntiTerm(p),t) -> {
  return `AntiMatchConstraint(MatchConstraint(p,t));
}
```

Another interesting result that has an impact on legibility is related to the number of lines of code. Including the support for all the new extensions to the left hand sides of the rules, the new compiler has 2500 lines of code in 14 files, compared to 2700 in 7 files for the old version (it contained anti-patterns but not the other types of constraints detailed in Chapter 4 on page 63). The counted lines correspond only to those containing active code, ignoring the comments and the empty ones.

5.4.2. Performance analysis

In this section we provide some benchmarks for the compiler, concerning both compilation time and runtime performance of the generated code. Although generally the compilation time is not crucial for a system, it is interesting to have an idea of the performances in this area.

Compilation time

TOM is implemented in TOM, using a bootstrap technique. Since the system is quite large, a very interesting benchmark is the self-compilation. In the following table, the version `2.4` corresponds to the old compiler, and `2.5` to the new one. The system is bootstrapped, thus the compilation is performed in each case with the current version of the compiler.

	Compilation time	Files	Lines	Nb. of patterns
version 2.4	1 *min*	77	22139	1325
version 2.5	2 *min*, 33 *s*	86	25513	1616

The difference in the compilation time may come from two sources. The first one is the fact that the new compiler contains more patterns, and often more complicated, compared to the old one. This implicitly leads to increased compilation times when compiling the new compiler's sources. The second source may be the multiple traversals of the AST in version `2.5` compared to a single-traversal approach of the old compiler. In the new compiler, the AST is traversed in each phase.

We also benchmarked two other applications that use a lot of **%match** and **%strategy** constructs: lemuridae, a proof assistant for superdeduction [BHK07] (a dynamic extension of sequent calculus), and CCG, a tool for automata completion applied to JAVA program analysis. Both are available on the TOM source repository[1]. The results are in the following table:

	Compil. time 2.4	Compil. time 2.5	Files	Lines	Nb. of patterns
lemuridae	12 *s*	18 *s*	4	1971	461
CCG	35 *s*	38 *s*	1	920	680

Run-time performance

The run-time performance of the code generated by the new compiler didn't change, as for the same input it outputs the same `PIL` code as the previous one. Some small speed-ups may come from ordering the numeric constraints, as we detailed in Section 4.3.1.

On several classical benchmarks TOM is competitive with state of the art implementations like ASF+SDF, ELAN, or MAUDE[2]. In the following, `Fibonacci` computes 500 times the 18th Fibonacci number using a Peano representation. `Sieve` computes prime numbers up to 2000 using list matching to eliminate non-prime numbers:

$$(c_1*, x, c_2*, y, c_3*) \rightarrow (c_1*, x, c_2*, c_3*) \text{ if } x \text{ divides } y^3.$$

[1] `http://gforge.inria.fr/projects/tom/`

[2] note that MAUDE is an *interpreter*. The experimental results are extraordinarily good compared to the compiled and highly optimized low-level C implementations

[3] on this example, the performance of ASF+SDF may be explained by the lack of support for built-in integers

`Evalsym`, `Evalexp`, and `Evaltree` are three benchmarks based on the normalization of the expression 2^{22} mod 17 using different reduction strategies. These three benchmarks were first presented in [vdBKO99]. As the previous examples, all these programs are available in the TOM source repository. The measurements were done on a MacBook Pro 2.33 GHz, using Java 1.5.0 and gcc 4.0.

	Fibonacci	Sieve	Evalsym	Evalexp	Evaltree
ASF + SDF	0.4 s	24.1 s	1.7 s	2.0 s	1.6 s
ELAN	1.1 s	–	5.3 s	11.8 s	10.1 s
MAUDE	2.3 s	17.7 s	8.8 s	15.4 s	21.3 s
TOM C	0.6 s	0.2 s	1.9 s	2.0 s	2.2 s
TOM Java	1.9 s	2.2 s	7.8 s	8.4 s	8.2 s

5.4.3. Future work

There are several directions for further improvements of the compiler. An important one concerns the ordering of the constraints before transforming them into `PIL` — the scheduling phase. As we detailed in the Section 5.3.4, the ordering is implemented for the moment in a naive way, just by switching the places of the constraints that do not respect some pre-defined basic precedences. We think that a more formal analysis of the ordering is required. Ideally, we should have two types of precedences: one that is absolute and should be always respected, like declaring the variables before they are used, and others that could be just "preferences" (for optimization reasons). In other words, besides ordering the constraints such that they respect the absolute order, we should also try to order them as much as possible according to the "preferences". These priorities should be formally studied in order to insure that there are no conflicts and therefore the rules performing the ordering always terminate, and in the same time an optimal ordering is obtained. We feel that this would allow to implement some optimizations that are more difficult and more costly to obtain on the `PIL` code.

It would be also useful to have a phase in the compiler that performs some checks and report warnings to the user, as those presented by Maranget in [Mar07]. We refer in particular to the exhaustive analysis, *i.e.* if all the possible cases are considered in a **%match** construct. Other analysis, such as checking if some cases are subsumed by others, do not apply in the case of the pattern matching of TOM, because the semantics of **%match** is that of a *switch* construct where all applicable actions are executed in the absence of *break* instructions.

For ensuring that future extensions are performed in safe way, a statical analysis on the possible inputs and outputs of each phase should be performed. This would help to guarantee that any possible input constraint gets to be transformed, and that transformations are complete according to the expected output. This is not only valid for the compiler, but for the other phases of the system as well. For instance, using this technique for the typer would guarantee that at the end of this phase no symbol or variable in the AST is left un-typed.

6. Filtrage sur des structures complexes Java

Pour promouvoir l'utilisation du filtrage dans des projets complexes du milieu industriel, nous proposons dans ce chapitre une technique pour extraire de manière automatique des informations structurelles à partir d'une hiérarchie arbitraire de classes JAVA. Cela permettra l'intégration du filtrage offert par TOM dans n'importe quelle application JAVA, nouvelle ou déjà existante, en demandant peu d'effort de la part de l'utilisateur.

6.1. Motivation

Les problèmes qu'on résout avec des systèmes informatiques sont de plus en plus complexes de nos jours, et par conséquent, on écrit de plus en plus de code. Dans ce contexte, la maintenance des logiciels devient une préoccupation importante, accentuant le besoin de code qui soit facile à lire et à modifier. Dans des domaines d'application comme la simplification de formules, ou l'analyse et la transformation de requêtes par exemple, un pourcentage important du code est écrit pour décider si on est dans un cas ou dans un autre. Dans les langages impératifs, comme JAVA ou C, ces décisions sont implémentées la plupart du temps avec des constructions ad-hoc, qui sont habituellement des instructions imbriquées du type `if-then-else`. Les applications liées au *e-commerce* sont des exemples typiques où une partie importante du code est concernée avec l'analyse des données après leur extraction d'une base de données. Habituellement, elles utilisent des API pour la persistance, comme par exemple *Java Persistence API (JPA)* [KS06] pour représenter une base de données comme un modèle objet. Après, l'analyse et la transformation de ces données (l'adaptation d'un catalogue en ligne pour plusieurs dispositifs par exemple) sont faites sur des objets JAVA, en utilisant une combinaison de méthodes `get` et d'instructions `if-then-else`.

Dans ce cadre, l'utilisation du filtrage permettrait d'exprimer d'une manière beaucoup plus concise et lisible ces analyses. Par exemple, considérons une hiérarchie de classes JAVA composé de `CCAccount` (*credit card account*) et `SAccount` (*savings account*) héritant de `Account` et ayant chacune un champ `owner` du type `Account`. Étant donnés deux objets `s1` et `s2`, s'ils sont respectivement du type `CCAccount` et `SAccount`, et s'ils ont le propriétaire (*owner*), on veut imprimer ce nom. S'ils sont tout les deux du type `CCAccount`, on imprime simplement le texte `"CCAccount"`. En utilisant le filtrage, cela peut s'exprimer de la manière suivante :

```
%match(s1,s2) {
  CCAccount(Owner(name)),SAccount(Owner(name)) -> { print(name); }
```

```
  CCAccount(_),CCAccount(_) -> { print("CCAccount"); }
}
```

Le code JAVA équivalent serait le suivant :

```
if (s1 instanceof CCAccount) {
  if (s2 instanceof SAccount) {
    Owner o1=((CCAccount)s1).getOwner();
    Owner o2=((SAccount)s2).getOwner();
    if (o1 != null && o2 != null) {
      if ((o1.getName()).equals(o2.getName())) {
        print(o1.getName());
      }
    }
  } else {
    if (s2 instanceof CCAccount) { print("CCAccount"); }
  }
}
```

Les avantages du **%match** deviennent encore plus clairs quand il s'agit du filtrage associatif. On s'est alors posé la question de savoir comment on peut faciliter l'intégration du filtrage offert par TOM dans des applications complexes écrites en JAVA. Le tout en demandant un effort minimal de la part d'utilisateur.

Les applications JAVA pour lesquelles on voudrait fournir le filtrage sont à la fois des applications existantes et aussi des applications nouvelles. Cette distinction est très importante parce que dans le cadre d'applications existantes, les structures de données sont déjà définies, et il n'est pas possible de les changer. Si nous voulons offrir des constructions de filtrage pour ces applications, on doit pouvoir le faire sans modifier ces structures. Dans une application nouvelle, on peut envisager d'imposer quelques contraintes à respecter par le programmeur sur les classes qu'il aura besoin de filtrer, éventuellement l'utilisation de GOM.

Suite à ces remarques, la méthode que nous devons choisir doit s'appliquer de manière uniforme pour les deux catégories d'applications. Elle doit permettre à l'utilisateur d'utiliser des instructions **%match** dans toutes les étapes de développement d'une application, sans imposer des modifications du code existant.

6.2. Approches possibles

Comme nous l'avons déjà vu dans les sections précédentes, TOM permet de filtrer sur des objets pour lesquelles des *mappings* qui décrivent leur structure ont été définis. La définition de ces *mappings* n'est pas difficile, mais nécessite des connaissances sur la façon dont ils fonctionnent. Dans une application de taille importante il est fréquent d'avoir des dizaines, voir des centaines des classes sur lesquelles on veut pouvoir filtrer.

Par conséquent, écrire manuellement les *mappings* n'est pas une option viable, et c'est pourquoi nous devions trouver un façon pour les générer automatiquement.

Quand l'utilisateur écrit sa structure de données avec GOM, les *mappings* pour les objets définis sont automatiquement générés. Pour les utiliser dans TOM, on a juste à les inclure avec une instruction **%include** dans la classe où on veut faire du filtrage. À partir d'une hiérarchie arbitraire de classes JAVA, une première option serait de générer d'une manière automatique une signature GOM correspondant à cette hiérarchie. De cette façon, l'utilisateur aurait plusieurs avantages : il pourrait faire du filtrage sur ses classes, il pourrait écrire des stratégies, et il aurait une vue plus abstraite pour sa structure de données. Mais cette approche pose plusieurs problèmes :

1. GOM ne supporte que deux niveaux d'héritage. Par conséquence, les hiérarchies plus complexes ne peuvent pas avoir une correspondance directe dans GOM.
2. Souvent, les classes de l'utilisateur sont plus complexes, pouvant contenir plus d'informations que les champs, comme par exemple des annotations, ou des méthodes qui implémentent une certaine fonctionnalité.
3. Les structures générés par GOM sont non-mutables. Ça veut dire que le code dans une application existante qui utilise des méthodes du type `set` pour changer les objets et les utiliser après ne peut plus marcher, et des modifications sont nécessaires pour le mettre à jour.

Étant données ces restrictions du langage GOM, nous avons abandonné l'idée de générer une signature GOM et nous avons décidé de générer directement les *mappings* TOM pour une hiérarchie JAVA donnée. Avec cette approche, on ne modifie pas les structures de l'utilisateur, et on n'en crée pas d'autre en parallèle. C'est aussi une méthode très souple, parce que l'utilisateur peut apporter des modifications aux *mappings* après leur génération automatique, dans le cas où ils ne correspondent pas complètement à ses besoins.

Cela a aussi quelques inconvénients par rapport à la génération d'une signature GOM. Le plus important étant qu'on perd le support pour les stratégies. Avec les *mappings* générés, les structures de l'utilisateur peuvent seulement être utilisées dans des **%match**. Pour pouvoir utiliser une classe dans une stratégie, en plus d'avoir un *mapping* TOM pour cette classe, elle doit implémenter une interface qui se trouve dans la bibliothèque de stratégies, interface qui contient des méthodes donnant accès aux champs de la classe. Toutes les classes qui sont générées par GOM implémentent cette interface. Néanmoins, le filtrage en soit apporte des bénéfices considérables. Dans la section 6.4 page 108 nous présentons les perspectives d'avoir aussi le support pour les stratégies.

6.3. Extraction automatique des informations structurelles d'une hiérarchie Java

Nous présentons dans cette section les approches techniques que nous avons choisi pour la génération automatique des *mappings* TOM à partir des fichiers contenant des classes JAVA.

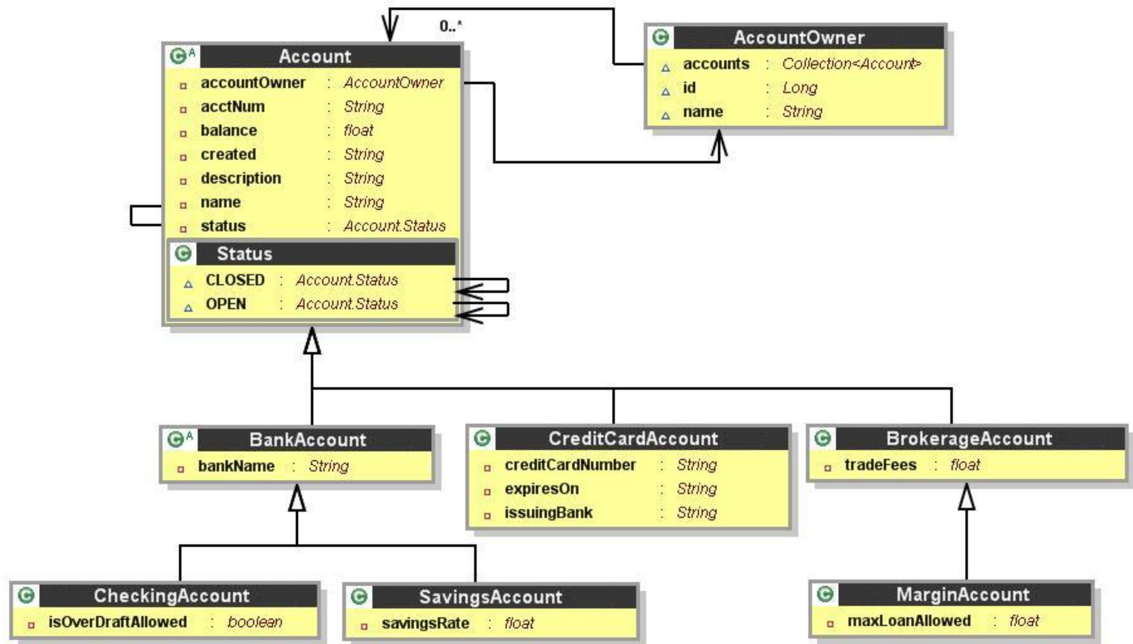


FIG. 6.1.: Exemple d'une hiérarchie de classes JAVA. `AccountOwner` contient une liste de `Account`. Chaque `Account` a un `AccountOwner`. `BankAccount`, `CreditCardAccount`, et `BrokerageAccount` héritent de `Account`.

6.3.1. La correspondance entre les classes Java et les *mappings*

Étant donnée une hiérarchie arbitraire JAVA, comme celle présentée dans la figure 6.1, le but est de pouvoir écrire des instructions **%match** en utilisant des objets qui sont des instances de ces classes. Nous voulons aussi avoir la possibilité de filtrer, pour chaque classe, vers des listes d'objets de cette classe.

Comme nous avons détaillé dans la section 1.1.3 page 11, pour utiliser un objet JAVA dans un **%match**, on doit avoir une construction **%typeterm**. Ensuite, pour pouvoir décomposer l'objet, une construction **%op** est nécessaire. Par exemple, pour la classe `Account`, on devrait avoir les constructions suivantes :

```
%typeterm Account {
  implement      { Account }
  is_sort(t)     { t instanceof Account }
  equals(t1,t2)  { t1.equals(t2) }
}

%op Account Account(name:String,status:Status,description:String,
    accountOwner:AccountOwner,created:String,balance:float,acctNum:String) {
  is_fsym(t)               { t instanceof Account }
  get_slot(name, t)        { t.getName() }
```

```
  get_slot(status, t)        { t.getStatus() }
  get_slot(description, t)   { t.getDescription() }
  get_slot(accountOwner, t)  { t.getAccountOwner() }
  get_slot(created, t)       { t.getCreated() }
  get_slot(balance, t)       { t.getBalance() }
  get_slot(acctNum, t)       { t.getAcctNum() }
}
```

Dans la figure 6.2 page 105, on donne sur la deuxième colonne la correspondance entre une classe JAVA de base[1] et ses *mappings* dans TOM. En ce qui concerne les sous-classes, les choses sont un peu plus compliquées. Supposons qu'étant donné un objet `s1`, on veut afficher le message `"an Account"`, `"a SavingsAccount"` et `"a CreditCardAccount"` si l'objet est du type `Account`, `SavingsAccount` et respectivement `CreditCardAccount`. La solution la plus intuitive serait d'écrire le **%match** suivant :

```
%match(s1) {
  Account[] -> { print("an␣Account"); }
  SavingsAccount[] -> { print("a␣SavingsAccount"); }
  CreditCardAccount[] -> { print("a␣CreditCardAccount"); }
}
```

Pour que le vérificateur de type du TOM puisse accepter ce **%match**, nous devons respecter les conditions suivantes : l'objet `s1` doit être d'un type déclaré avec **%typeterm** et les co-domaines des trois opérateurs doivent être les mêmes. En plus, comme chaque sous-classe peut aussi être vu comme une instance d'une de ses super-classes, il est normal de pouvoir accéder aux champs de toutes les super-classes dans les opérateurs. Par exemple, on devrait pouvoir écrire :

```
%match(s1) {
  Account[] -> { print("un␣Account"); }
  SavingsAccount[savingsRate=sr,bankName=bn,name=nm] -> {
    print("SavingsAccount,␣savingsRate=" + ‘sr +"␣bank␣name=" + ‘bn
           + "␣name=" + ‘nm);
  }
}
```

Pour pouvoir prendre en compte ces contraintes, une solution consiste à considérer le co-domaine d'un opérateur d'une sous-classe comme ayant le type de la classe la plus haute dans la hiérarchie (avant `Object`), et aussi les arguments de l'opérateur associe à une classe doivent contenir tous les champs de toutes ses super-classes. Ainsi, la définition de **%op** pour la classe `SavingsAccount` devrait être :

```
%typeterm SavingsAccount {
```

[1] qui n'hérite pas d'une autre classe

```
  implement        { SavingsAccount }
  is_sort(t)       { t instanceof SavingsAccount }
  equals(t1,t2)    { t1.equals(t2) }
}

%op Account SavingsAccount(savingsRate:float,bankName:String,name:String,
    status:Status,description:String,accountOwner:AccountOwner,created:String,
    balance:float,acctNum:String) {
  is_fsym(t)                    { t instanceof SavingsAccount }
  get_slot(savingsRate, t)      { ((SavingsAccount)t).getSavingsRate() }
  get_slot(bankName, t)         { ((SavingsAccount)t).getBankName() }
  get_slot(name, t)             { ((SavingsAccount)t).getName() }
  get_slot(status, t)           { ((SavingsAccount)t).getStatus() }
  get_slot(description, t)      { ((SavingsAccount)t).getDescription() }
  get_slot(accountOwner, t)     { ((SavingsAccount)t).getAccountOwner() }
  get_slot(created, t)          { ((SavingsAccount)t).getCreated() }
  get_slot(balance, t)          { ((SavingsAccount)t).getBalance() }
  get_slot(acctNum, t)          { ((SavingsAccount)t).getAcctNum() }
}
```

Le *cast* dans la partie droite des méthodes `get_slot` est nécessaire parce que le co-domaine de l'opérateur est `Account`, et toutes les instances intermédiaires des objets du type `SavingsAccount` qui sont créés dans la compilation d'un **%match** sont du type `Account`. Par conséquent, l'objet `t` est déclaré de type `Account` dans le code Java généré par Tom, et pour accéder aux méthodes appartenant aux classes plus bas dans sa hiérarchie (comme celles de `SavingsAccount` dans ce cas), on a besoin de faire le *cast*.

Dans la figure suivante nous présentons la correspondance entre les classes Java et les *mappings* Tom qui doivent être fournis. Le tableau résume les explications que nous avons données jusqu'à ce point.

Filtrage de liste

Quand on utilise des modèles objet comme *Java Persistence API* pour les bases de données, il est important d'avoir la possibilité de filtrer des listes d'objets aussi, parce que dans ce contexte, les enregistrements dans un tableau de la base de données correspondent à une liste d'objets dans le modèle objet. Par conséquent, une requête vers la base de données a comme réponse une liste d'objets sur laquelle l'utilisation du filtrage associatif de Tom peut être très utile. Pour cette raison, des *mappings* permettant le filtrage associatif pour tous les objets doivent aussi être fournis.

Comme on l'a déjà vu auparavant, tous les objets d'une hiérarchie sont supposés être des instances de la classe de base, parce que leur co-domaine déclaré avec **%op** est le type de la classe la plus haute dans la hiérarchie. De ce fait, ayant des *mappings* pour les listes d'objets des classes de base permet aussi de filtrer sur les listes des autres objets qui se trouves plus bas dans leur hiérarchies. Par exemple, les définitions suivantes devraient

mappings Tom		Java : classes de base	Java : sous-classes
%typeterm	la sorte algébrique (pour `is_sort` et `implements`)	le nom de la classe	le nom de la classe
%op	le nom de l'opérateur	le nom de la classe	le nom de la classe
	le domaine de l'opérateur	tous les champs de la classe	tout les champs de la classe + les champs de toutes les classes plus haut dans sa hiérarchie
	le co-domaine de l'opérateur	le nom de la classe	le nom de la classe qui est le plus haut dans sa hiérarchie (avant `Object`)
	les méthodes `get_slot`	pour chaque champ accessible de la classe (par une méthode `get` ou `is`)	pour chaque champ accessible de la classe (par une méthode `get` ou `is`) + tous les champs accessibles de toutes les classes plus haut dans sa hiérarchie

Fig. 6.2.: La correspondance entre les *mappings* Tom et les classes Java.

être fournies pour pouvoir filtrer sur des listes d'objets du type `Account`, `BankAccount`, `CheckingAccount`, `CreditCardAccount`,*etc* :

```
%oparray AccountList accountList(Account*) {
  is_fsym(t)            { t instanceof java.util.List }
  make_empty(n)         { new java.util.ArrayList<Account>(n) }
  make_append(e,l)      { (java.util.ArrayList<Account>)myAdd(e,l) }
  get_element(l,n)      { l.get(n) }
  get_size(l)           { l.size() }
}

private static java.util.List myAdd(Object e, java.util.List l) {
  // use a clone to avoid side effects on the list being filtered
  java.util.ArrayList tmp = ((java.util.ArrayList)l).clone();
  tmp.add(e);
  return tmp;
}
```

Tom fournit deux types de constructions pour déclarer des opérateurs associatifs : **%oplist** et **%oparray**, correspondant respectivement à une liste et à un tableau. Au niveau du filtrage et de l'utilisation de **%match**, il y a aucune différence entre les deux. Les particularités sont juste au niveau du moyen d'accès aux éléments, qui est fait en utilisant des indices pour les tableaux, contrairement aux méthodes `get_head` et `get_tail` pour les listes. Nous avons choisi ici l'opérateur **%oparray** pour pouvoir utiliser la classe Java `ArrayList`.

La première ligne spécifie le domaine et le co-domaine de l'opérateur, ainsi que son nom. Les opérations spécifiques aux tableaux doivent être définies. Le *mapping* précise ainsi comment construire un tableau vide, et comment insérer un nouvel élément. Il détaille aussi la manière d'accéder à un élément du tableau et de récupérer sa taille. La méthode `myAdd` est une méthode auxiliaire permettant d'exécuter plusieurs instructions dans la partie droite de `make_append`, opération interdite autrement. Elle doit être fournie une seule fois pour tous les *mappings*.

Nous présenterons par la suite la méthode utilisée pour la génération automatique de ces *mappings*.

6.3.2. La génération des *mappings* par réflexivité

Étant donné une hiérarchie de classes Java, le but est de générer automatiquement des *mappings* pour toutes les classes de la hiérarchie, en respectant les correspondances que nous avons décrit au début de cette section.

Nous avons développé un outil, appelé MappingGenerator[2], qui prend comme entrée de la part de l'utilisateur le chemin vers un fichier avec l'extension `.class`, ou bien le chemin vers un répertoire. Dans le premier cas, l'outil génère les *mappings* pour la classe contenue dans le fichier donné, et dans le deuxième cas, il parcourt récursivement le répertoire pour trouver tous les fichiers `.class`, et génère des *mappings* pour chacun d'eux.

Le MappingGenerator est disponible à la fois en ligne de commande, et à partir du *plug-in* Eclipse de Tom.

Ayant le *plug-in* Tom installé dans Eclipse, pour générer des *mappings* on fait un *click*-droit sur un fichier `.class` ou sur un répertoire, on choisit l'option *Generate Tom Mappings* et on donne le nom du fichier qui contiendra les *mappings*, ainsi que son répertoire destination dans la fenêtre qui est affiché (figure 6.3 page ci-contre).

Le but de cette intégration dans Eclipse est de permettre à n'importe quel stade de développement d'un projet Java l'intégration du code Tom dans certaines classes, tout en gardant les outils et l'état courant du projet de l'utilisateur inchangés.

La génération des *mappings* se fait dans un temps très court. Par exemple, pour les classes présentées dans la figure 6.1, le *plug-in* Eclipse génère les 213 lignes des *mappings* en moins de 200ms sur un ordinateur portable, processeur Intel Pentium M, 2GHz avec 1GB RAM.

Pour récupérer les informations dont on a besoin (le nom de la classe, les noms des champs, *etc.*) a partir d'un fichier `.class`, on a utilisé la réflexivité de Java [FF04]. Celle-ci permet, à partir d'un nom de classe ou d'un objet de type `Class`[3], de récupérer toutes les informations visibles de la classe en cause.

Pour un fichier `.class`, les *mappings* générés sont les suivants :

– une déclaration **%typeterm**,
– une déclaration **%op**,

[2] disponible dans le répertoire `https ://scm.gforge.inria.fr/svn/tom/applications/mappingGenerator` du dépôt Tom

[3] une instance de la classe `Class` représente des classes ou des interfaces dans une application Java

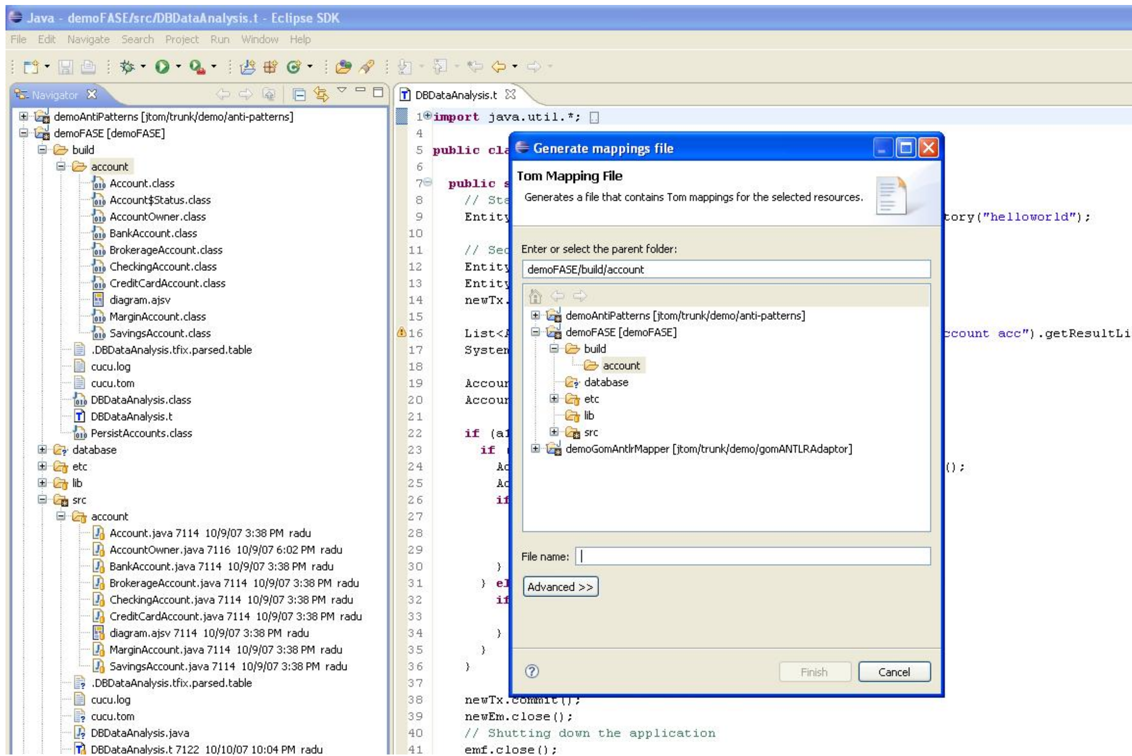


FIG. 6.3.: Le générateur de *mappings* dans ECLIPSE, lancé sur le répertoire `build/account` .

– une déclaration **%oparray** pour toutes les classes de base,
– une déclaration **%typeterm** pour chaque type d'un champ, type pour lequel un **%typeterm** n'a pas été crée auparavant.

À la fin de l'exécution, on dispose d'un fichier avec tous les *mappings* correspondant aux entrées, ainsi qu'un fichier de *log*, qui nous fournit, pour chaque fichier d'entrée, des informations sur l'état de la génération : si elle a réussi, ou, dans le cas contraire un message explicatif.

Difficultés rencontrées

La difficulté principale vient du fait que les champs d'une classe ne sont pas toujours détectables par réflexivité.

Pour générer les méthodes `get_slot`, MAPPINGGENERATOR récupère par réflexivité toutes les méthodes qui commencent par `get` ou `is` — on prend en compte les méthodes `is` aussi, parce que l'accès aux champs booléens est fait habituellement par ce type de méthodes. Ensuite, il considère que la suite de `get` ou `is` est le nom du champ correspondant, en transformant (si c'est le cas) le premier caractère après `get` et `is` de majuscule en minuscule (on suppose que les méthodes respectent les conventions standard de nommage). On est obligé de procéder de cette façon, parce que par réflexivité

on n'a pas accès aux informations concernant les champs privés de la classe. Alors, si les méthodes `get` et `is` ne sont pas correctement déclarées, les *mappings* générés ne sont pas corrects, et ils auront besoin de l'intervention manuelle de l'utilisateur pour les corriger. MappingGenerator prend aussi en compte les champs pour lesquels deux méthodes `get` (ou `is`) sont déclarées (par surcharge par exemple). Dans ce cas, un seul mapping est généré.

D'autres problèmes d'ordre technique peuvent apparaître lorsqu'une des classes analysées a un champ pour lequel la classe correspondant à son type ne se trouve pas dans le *classpath*. Le MappingGenerator cherche dans tout le *classpath* du système et aussi dans le *classpath* et toutes les références du projet Eclipse courant si le générateur a été lancé à partir de Eclipse. L'utilisateur peut aussi fournir, sous la forme de paramètres au moment du lancement du MappingGenerator, des chemins vers des répertoires contenant des classes ou des `jar` à inclure dans la recherche.

Nous avons aussi dû écrire un nouveau chargeur de classes (*ClassLoader*), pour pouvoir créer un objet de type `Class` à partir d'un fichier sur le disque-dur, méthode qui n'est pas supporté par les *ClassLoaders* standards de Java.

6.4. Bilan et perspectives

Nous avons présenté dans cette section un moyen d'intégrer le filtrage de Tom dans n'importe quelle application Java, nouvelle ou déjà existante, avec le moindre effort de la part de l'utilisateur. Pour utiliser Tom dans un projet Eclipse, il suffit de renommer l'extension du fichier de `.java` en `.t`, générer les *mappings*, et inclure le code généré avec une instruction `%include`. De cette façon, le code peut être simplifié en remplaçant par filtrage la majorité des constructions difficiles à lire et à maintenir comme les instructions `if-then-else` imbriquées, sans avoir aucun inconvénient (grâce au *plug-in* Eclipse de Tom, l'éditeur pour les fichiers Tom offre les mêmes fonctionnalités que l'éditeur Java — la même coloration, la complétion automatique, *etc.*).

Ce travail ouvre aussi une série de perspectives. La plus significative est la possibilité d'utiliser la bibliothèque de stratégies de Tom sur des structures Java arbitraires sans se restreindre aux structures construites avec Gom. La difficulté consiste à ne pas avoir à apporter des modifications aux structures existantes. Comme on l'a précédemment mentionné, pour pouvoir utiliser les objets d'une classe dans une stratégie, en plus d'avoir un *mapping* Tom pour cette classe, elle doit implémenter une interface (`Visitable`) qui se trouve dans la bibliothèque de stratégies. Cette interface contient des méthodes comme `getChildAt(int childNumber)` et `getChildrenCount()`, qui peuvent facilement être déduites à partir de méthodes `get_slot` définies dans le **%op** pour la classe respective. Récemment, nous avons travaillé dans l'équipe Tom sur un moyen d'inférer les méthodes de la classe `Visitable` à partir des *mappings* qui on été générés pour une classe arbitraire Java, ce qui va permettre l'utilisation de **%match** ainsi que des stratégies pour n'importe quels objets Java. Les applications possibles sont surtout dans le domaine de l'analyse des programmes Java, en parcourant les arbres de syntaxe abstraite (AST) produits par les *parsers* Java habituels. Modifier manuellement les classes qui sont utili-

sées pour construire ces AST pour qu'elles implémentent l'interface `Visitable` n'est pas une option viable, à cause de leur nombre très élevé, et du fait que souvent les sources ne sont pas disponibles.

En utilisant les *mappings* produits par MappingGenerator sur des hiérarchies Java avec plusieurs niveaux, comme celle dans la figure 6.1 page 102, on s'est rendu compte que le typage de Tom n'est pas assez souple. Par exemple, considérons le code suivant :

```
%match(s1) {
  t@Account[] -> { print("t=" + `t); }
  t@SavingsAccount[] -> { print("t=" + `((SavingsAccount)t).savingsRate); }
  t@CreditCardAccount[] -> { print("t=" + `t); }
}
```

Le type de la variable `t` dans la partie droite des règles est `Account`. Ceci est contraire à l'intuition, parce que, dans le cas de la deuxième règle, avec le motif `SavingsAccount` on a déjà vérifié que le sujet est du type `SavingsAccount` et pas `Account`. Par conséquent, on est obligé de faire un *cast* dans la partie droite pour pouvoir utiliser `t` avec son vrai type.

Cette situation vient du fait que le co-domaine déclaré est `Account` pour les trois opérateurs, pour pouvoir les mettre au même niveau dans un **%match** comme on a fait ci-dessus. Ce qu'on envisage, est de faire en sorte que Tom puisse connaître, en plus du co-domaine de l'opérateur, son type précis. Le MappingGenerator pourrait alors enrichir les **%op** avec cette information, qui sera par la suite utilisée par Tom. Ayant les deux types d'un opérateur dans les *mappings* (son type précis, qui correspond ici au type dynamique, et le type de la classe qui se trouve le plus haut dans sa hiérarchie), on pourrait ainsi envisager d'autoriser un mélange de plusieurs types dans une construction **%match**, tant qu'ils appartiennent tous à la même hiérarchie.

Conclusion

L'objectif principal de ce thèse est d'une part de développer des formalismes capables d'augmenter l'expressivité du filtrage, et d'autre part de concevoir un environnement capable d'assurer l'implémentation de ces formalismes de manière efficace et élégante.

Contributions

Nous résumons dans cette section les différentes contributions de cette thèse.

Anti-patterns. Le concept d'anti-pattern est très utile pour exprimer des conditions négatives de filtrage. Il offre la possibilité de représenter de manière naturelle et surtout compacte les ensembles de termes. La principale nouveauté consiste en l'utilisation des compléments directement ancrés dans les motifs, et de filtrer vers cette nouvelle construction. Cela donne une solution intuitive et appropriée pour le développement d'une extension de langage permettant d'exprimer des conditions négatives dans les langages qui offrent des fonctions de filtrage, soient-ils des langages fonctionnels ou basés sur la réécriture.

Les anti-patterns sont pratiques dans le cas syntaxique, mais leur pouvoir expressif est encore plus mis en valeur quand les symboles sont associés avec des théories équationnelles, en particulier avec l'associativité avec élément neutre et éventuellement avec la commutativité. Nous avons fourni la sémantique des anti-patterns de manière générale, modulo une théorie équationnelle arbitraire, et nous avons aussi proposé plusieurs algorithmes à base de règles pour résoudre les problèmes de filtrage entre les anti-patterns et les termes clos. Nous avons montré que tous ces algorithmes sont corrects et complets. Dans le cas syntaxique, ils sont aussi unitaires.

Les anti-patterns ont été complètement intégrés dans le filtrage de Tom, pour le filtrage syntaxique mais aussi pour le cas $\mathcal{AU}$.

Contraintes non-atomiques de filtrage. En s'inspirant de l'expressivité des règles de production, nous avons étendu le filtrage de Tom afin de pouvoir encoder toutes les conditions pour l'application d'une règle dans la partie gauche. En plus d'obtenir un code plus clair et plus concis, ces extensions nous permettront aussi d'effectuer plus d'optimisations et des vérifications.

Compilation par réécriture. Nous avons proposé une nouvelle méthode de compilation pour les combinaisons complexes de conditions de filtrage. C'est une méthode basée sur des systèmes de réécriture, dont l'application est contrôlée par des stratégies. Chaque

système de réécriture est un *plug-in* qui s'intègre à une plate-forme. De cette façon, on obtient un environnement de compilation modulaire et extensible, où chaque système de réécriture est en charge du traitement d'un seul type de contrainte (contrainte de filtrage syntaxique, contrainte de filtrage associatif, *etc*).

Promouvoir le filtrage. Bien que le filtrage soit une construction qui augmente la concision et la lisibilité du code, son utilisation dans les projets du milieu industriel est souvent limitée. Nous avons proposé une méthode permettant d'intégrer le filtrage de TOM dans n'importe quelle application JAVA, nouvelle ou déjà existante, en demandant peu d'effort de la part de l'utilisateur.

Perspectives

Plus d'anti-patterns. En plus de promouvoir l'utilisation des anti-patterns dans les langages et les recherches à base de motifs en général, nous envisageons aussi d'étudier plusieurs propriétés théoriques pour les systèmes incluant les anti-patterns. Il serait intéressant de savoir quel est l'impact de l'utilisation des anti-patterns dans les parties gauches d'un système de réécriture sur sa terminaison, sa complétude ou bien sa confluence.

Les anti-patterns représentent de manière compacte les ensembles des termes. Alors ils pourraient très bien être intégrés dans des outils comme GOM par exemple, permettant de déclarer des termes qui *ne sont pas* d'une certaine forme. Dans ce cas, les problèmes de filtrage classiques évolueraient vers des équations de filtrage ayant potentiellement les parties gauches et droites des anti-patterns. Comme le membre droit d'une contrainte de filtrage ne serait plus un terme clos, mais plutôt un ensemble de termes, cela donnerait un type spécifique de problèmes d'unification.

Vers plus d'expressivité. Une direction future assez naturelle est d'augmenter toujours l'expressivité du filtrage, et du langage TOM en particulier. L'étude et l'intégration de nouvelles contraintes permettra à TOM de résoudre des problèmes comme ceux résolus aujourd'hui par les systèmes de règles de production, ou de se constituer dans un environnent puissant pour l'analyse de programmes.

Une meilleur intégration. Nous avons présenté une approche pour l'intégration du filtrage de TOM dans des applications JAVA existantes. Une perspective très souhaitable serait d'avoir aussi la possibilité d'utiliser la bibliothèque de stratégies de TOM, sans se restreindre aux structures construites avec GOM. La difficulté consiste à ne pas avoir à apporter des modifications aux structures de données existantes.

Les applications possibles sont surtout dans le domaine de l'analyse des programmes JAVA, en parcourant les arbres de syntaxe abstraite (AST) produits par les parseurs JAVA habituels. Modifier manuellement les classes qui sont utilisées pour construire ces

AST pour qu'elles implémentent l'interface exigée par Tom n'est pas une option viable, à cause de leur nombre très élevé, et du fait que souvent les sources ne sont pas disponibles.

Bibliographie

[AKPS94] Hassan Ait-Kaci, Andreas Podelski et Gert Smolka – « A feature constraint system for logic programming with entailment », *Theoretical Computer Science* **122** (1994), no. 1–2, p. 263–283.

[AKW95] Alexander Aiken, Dexter Kozen et Edward Wimmers – « Decidability of systems of set constraints with negative constraints », **122** (1995), no. 1, p. 30–44.

[AM87] Andrew W. Appel et David MacQueen – « A standard ML compiler », *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Portland, Oregon, USA) (G. Kahn, éd.), Lecture Notes in Computer Science, vol. 274, Springer, Berlin, 1987, p. 301–324.

[AM91] Andrew W. Appel et David B. MacQueen – « Standard ML of new jersey », *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming* (J. Maluszyński et M. Wirsing, éds.), Lecture Notes in Computer Science, vol. 528, Springer Verlag, 1991, p. 1–13.

[Aug85] Lennart Augustsson – « Compiling pattern matching », *Proceedings of Functional Programming Languages and Computer Architecture* (New York, NY, USA), Lecture Notes in Computer Science, vol. 201, Springer-Verlag, 1985, p. 368–381.

[AW92] Alexander Aiken et Edward L. Wimmers – « Solving systems of set constraints (extended abstract) », *LICS*, IEEE Computer Society, 1992, p. 329–340.

[BBK+07] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau et Antoine Reilles – « Tom : Piggybacking rewriting on java », *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science, vol. 4533, Springer-Verlag, 2007, p. 36–47.

[BBK+08] — , « The Tom manual », 2008, `http://tom.loria.fr/docs.php`.

[BBN+93] Franz Baader, Hans-Jürgen Bürckert, Bernhard Nebel, Werner Nutt et Gert Smolka – « On the expressivity of feature logics with negation, functional uncertainty, and sort equations », *Journal of Logic, Language and Information* **2** (1993), p. 1–18.

[BFB07] Clara Bertolissi, Maribel Fernández et Steve Barker – « Dynamic event-based access control as term rewriting », *DBSec* (S. Barker et G.-J. Ahn, éds.), Lecture Notes in Computer Science, vol. 4602, Springer, 2007, p. 195–210.

[BHK07] Paul Brauner, Clement Houtmann et Claude Kirchner – « Principles of superdeduction », *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science (LICS'07)* (Washington, DC, USA), IEEE Computer Society, 2007, p. 41–50.

[BKK+98] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau et Christophe Ringeissen – « An overview of ELAN », *Proceedings of WRLA 1998* (C. Kirchner et H. Kirchner, éds.), ENTCS, vol. 15, Elsevier Science Publishers, 1998.

[BKM06] Emilie Balland, Claude Kirchner et Pierre-Etienne Moreau – « Formal islands », *11th International Conference on Algebraic Methodology and Software Technology (AMAST 2006)*, Lecture Notes in Computer Science, vol. 4019, Springer, 2006, p. 51–65.

[BKN87] Dan Benanav, Deepak Kapur et Paliath Narendran – « Complexity of matching problems », *Journal of Symbolic Computation* **3** (1987), no. 1-2, p. 203–216.

[BM85] Marianne Baudinet et David MacQueen – « Tree pattern matching for ML », Unpublished, 1985.

[BM06] Emilie Balland et Pierre-Etienne Moreau – « Optimizing pattern matching compilation by program transformation », *3rd Workshop on Software Evolution through Transformations (SeTra'06)* (J.-M. Favre, R. Heckel et T. Mens, éds.), vol. 3, Electronic Communications of EASST, 2006.

[BN98] Franz Baader et Tobias Nipkow – *Term rewriting and all that*, Cambridge University Press, 1998.

[BS95] Rolf Backofen et Gert Smolka – « A complete and recursive feature theory », *Theoretical Computer Science* **146** (1995), no. 1–2, p. 243–268.

[Bür90] Hans-Jürgen Bürckert – « Matching — A special case of unification ? », Unification (C. Kirchner, éd.), London, 1990, p. 125–138.

[Bur07] Emir Burak – « Object-oriented pattern matching », Thèse de Doctorat d'Université, IIF Institut d'informatique fondamentale, Switzerland, 2007.

[Car84] Luca Cardelli – « Compiling a functional language », *Proceedings of the 1984 ACM Symposium on LISP and functional programming (LFP '84)* (New York, NY, USA), ACM, 1984, p. 208–217.

[CD89] Charles Consel et Olivier Danvy – « Partial evaluation of pattern matching in strings », *Information Processing Letters* **30** (1989), no. 2, p. 79–86.

[CDE+03] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer et Carolyn Talcott – « The maude 2.0 system », *Proceedings of RTA 2003* (R. Nieuwenhuis, éd.), Lecture Notes in Computer Science, vol. 2706, Springer-Verlag, 2003, p. 76–87.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer et Carolyn L. Talcott – *All about maude - a high-performance logical framework, how to specify, program and verify systems in rewriting logic*, Lecture Notes in Computer Science, vol. 4350, Springer, 2007.

[CG04] Luca Cardelli et Giorgio Ghelli – « Tql : a query language for semistructured data based on the ambient logic. », *Mathematical Structures in Computer Science* **14** (2004), no. 3, p. 285–327.

[CJL99] Yves Caseau, François-Xavier Josset et François Laburthe – « Claire : Combining sets, search, and rules to better express algorithms », *16th International Conference on Logic Programming*, 1999, p. 245–259.

[CK01] Hubert Comon et Claude Kirchner – « Constraint solving on terms », *Lecture Notes in Computer Science* **2002** (2001), p. 47–103.

[CKMM04a] Horatiu Cirstea, Claude Kirchner, Michael Moossen et Pierre-Etienne Moreau – « Production and rewrite systems », Report, INRIA Grand Est, 2004, http ://hal.inria.fr/inria-00280939/fr/.

[CKMM04b] — , « Production systems and rete algorithm formalisation », Report, INRIA Grand Est, 2004, http ://hal.inria.fr/inria-00280938/fr/.

[CL90] Hubert Comon et Pierre Lescanne – « Equational problems and disunification », Unification (C. Kirchner, éd.), London, 1990, p. 297–352.

[Com88] Hubert Comon – « Unification et disunification. Théories et applications », Thèse, Institut Polytechnique de Grenoble (France), 1988.

[Com91] — , « Disunification : a survey », Computational Logic. Essays in honor of Alan Robinson (J.-L. Lassez et G. Plotkin, éds.), Cambridge (MA, USA), 1991, p. 322–359.

[CP94] Witold Charatonik et Leszek Pacholski – « Negative set constraints with equality », *LICS*, IEEE Computer Society, 1994, p. 128–136.

[Der85] Nachum Dershowitz – « Computing with rewrite systems », *Information and Control* **65** (1985), no. 2/3, p. 122–157.

[dJO04] Hayco de Jong et Pieter Olivier – « Generation of abstract programming interfaces from syntax definitions », *Journal of Logic and Algebraic Programming* **59** (2004), no. 1-2, p. 35–61.

[DKKdO07] Daniel J. Dougherty, Claude Kirchner, Hélène Kirchner et Anderson Santana de Oliveira – « Modular access control via strategic rewriting », *12th European Symposium On Research In Computer Security (ESORICS 2007)*, Lecture Notes in Computer Science, vol. 4734, Springer, 2007, p. 578–593.

[Eke92] Steven M. Eker – « Associative matching for linear terms », Report CS-R9224, CWI, 1992, ISSN 0169-118X.

[Eke03] Steven EKER – « Associative-commutative rewriting on large terms », *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA 2003)*, Lecture Notes in Computer Science, vol. 2706, 2003, p. 14–29.

[FF04] Ira R. FORMAN et Nate FORMAN – *Java reflection in action (in action series)*, Manning Publications Co., Greenwich, CT, USA, 2004.

[FM01] Fabrice Le FESSANT et Luc MARANGET – « Optimizing pattern matching », *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming (ICFP'01)* (New York, NY, USA), ACM, 2001, p. 26–37.

[Fos96] Jeffrey S. FOSTER – « CLP(SC) : Implementation and efficiency considerations », *Proceedings of the Workshop on Set Constraints, held in Conjunction with CP'96, Boston, Massachusetts*, 1996.

[GKK+87] Joseph A. GOGUEN, Claude KIRCHNER, Hélène KIRCHNER, Aristide MÉGRELIS, José MESEGUER et Timothy WINKLER – « An introduction to OBJ-3 », *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)* (J.-P. JOUANNAUD et S. KAPLAN, éds.), Lecture Notes in Computer Science, vol. 308, Springer-Verlag, July 1987, Also as internal report CRIN : 88-R-001, p. 258–263.

[Grä91] Albert GRÄF – « Left-to-right tree pattern matching », *Proceedings of the 4th international conference on Rewriting techniques and applications (RTA-91)* (New York, NY, USA), Lecture Notes in Computer Science, vol. 488, Springer-Verlag, 1991, p. 323–334.

[has08] « Haskell » – 2008, `http://www.haskell.org/`.

[HK95] Miki HERMANN et Phokion G. KOLAITIS – « The complexity of counting problems in equational matching », *Journal of Symbolic Computation* **20** (1995), no. 3, p. 343–362.

[HL79] Gérard HUET et Jean-Jacques LÉVY – « Call by need computations in non-ambiguous linear term rewriting systems », Research report 359, INRIA, 1979.

[HO82] Christoph M. HOFFMANN et Michael J. O'DONNELL – « Pattern matching in trees », *Journal of the ACM* **29** (1982), no. 1, p. 68–95.

[Hue76] Gérard HUET – « Résolution d'equations dans les langages d'ordre 1, 2, . . . , ω », Thèse, Université de Paris 7 (France), 1976.

[Hue80] Gérard HUET – « Confluent reductions : Abstract properties and applications to term rewriting systems », **27** (1980), no. 4, p. 797–821, Preliminary version in 18th Symposium on Foundations of Computer Science, IEEE, 1977.

[ILO08] ILOG – « Ilog jrules », 2008, `http://www.ilog.com/products/jrules/`.

[Jac99] Peter JACKSON – *Introduction to expert systems (3rd edition)*, Addison-Wesley Longman Publishing Co., Inc., Harlow, England, 1999.

[JBo08] JBoss – « Drools », 2008, `http://www.jboss.org/drools/`.

[JK86a] Jean-Pierre Jouannaud et Hélène Kirchner – « Completion of a set of rules modulo a set of equations », *SIAM Journal of Computing* **15** (1986), no. 4, p. 1155–1194, Preliminary version in Proceedings 11th ACM Symposium on Principles of Programming Languages, Salt Lake City (USA), 1984.

[JK86b] Jean-Pierre Jouannaud et Hélène Kirchner – « Completion of a set of rules modulo a set of equations », *SIAM Journal on Computing* **15** (1986), no. 4, p. 1155–1194.

[JK91] Jean-Pierre Jouannaud et Claude Kirchner – « Solving equations in abstract algebras : a rule-based survey of unification », Computational Logic. Essays in honor of Alan Robinson (J.-L. Lassez et G. Plotkin, éds.), Cambridge (MA, USA), 1991, p. 257–321.

[Kah87] Gilles Kahn – « Natural semantics », Tech. Report 601, INRIA Sophia-Antipolis, February 1987.

[Kir86] Claude Kirchner – « Computing unification algorithms », *Symposium on Logic in Computer Science (LICS)*, IEEE Computer Society, 1986, p. 206–216.

[KK90] Claude Kirchner et Francis Klay – « Syntactic theories and unification », *Proceedings of the 5th Symposium on Logic in Computer Science (LICS)*, 1990, p. 270–277.

[KK99] Claude Kirchner et Hélène Kirchner – « Rewriting, solving, proving », A preliminary version of a book available at `http://www.loria.fr/~ckirchne/rsp.ps.gz`, 1999.

[KKM07a] Claude Kirchner, Radu Kopetz et Pierre-Etienne Moreau – « Anti-pattern matching », *Proceedings of the 16th European Symposium on Programming*, Lecture Notes in Computer Science, vol. 4421, Springer Verlag, 2007, p. 110–124.

[KKM07b] Claude Kirchner, Radu Kopetz et Pierre-Etienne Moreau – « Anti-pattern matching modulo », *21th International Workshop on Unification*, 2007.

[KKM08] — , « Anti-pattern matching modulo », *Proceedings of the 2nd International Conference on Language and Automata Theory and Applications (LATA)*, volume to appear in Lecture Notes in Computer Science, Springer-Verlag, 2008.

[Kla92] Francis Klay – « Unification dans les théories syntaxiques », Thèse de doctorat, Université Henri Poincaré - Nancy I, Oct 1992.

[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.

[KM08] Radu Kopetz et Pierre-Etienne Moreau – « Software quality improvement via pattern matching », *Proceedings of the 11th Conference on Fundamental Approaches to Software Engineering (FASE)*, Lecture Notes in Computer Science, vol. 4961, Springer-Verlag, 2008, p. 296–300.

[KMR05] Claude Kirchner, Pierre-Etienne Moreau et Antoine Reilles – « Formal validation of pattern matching code », *Proceedings of the 7th ACM SIGPLAN PPDP* (P. Barahona et A. Felty, éds.), ACM, 2005, p. 187–197.

[KS06] Mike Keith et Merrick Schincariol – *Pro ejb 3 : Java persistence api (pro)*, Apress, Berkely, CA, USA, 2006.

[Lab08] Ernest Friedman-Hill (Sandia National Laboratories) – « Jess », 2008, `http://herzberg.ca.sandia.gov/`.

[Ler08] Xavier Leroy – « The objective caml system release 3.10.2 », 2008, (`http ://caml.inria.fr/`).

[Liq06] Luigi Liquori – « iRho : the software [system description] », *DCM : International Workshop on Development in Computational Models. Electr. Notes Theor. Comput. Sci.* **135** (2006), no. 3, p. 85–94.

[LM87] Jean-Louis Lassez et Kim Marriott – « Explicit representation of terms defined by counter examples », *Journal of Automated Reasoning* **3** (1987), no. 3, p. 301–317.

[Mak77] Gennady S. Makanin – « The problem of solvability of equations in a free semigroup », *Math. USSR Sbornik* **32** (1977), no. 2, p. 129–198.

[Mar96] Claude Marché – « Normalized rewriting : an alternative to rewriting modulo a set of equations », *Journal of Symbolic Computation* **21** (1996), no. 3, p. 253–288.

[Mar07] Luc Maranget – « Warnings for pattern matching », *Journal of Functional Programming* (2007), no. 17(3), p. 647–656.

[MM82] Alberto Martelli et Ugo Montanari – « An efficient unification algorithm », **4** (1982), no. 2, p. 258–282.

[MNP97] Martin Müller, Joachim Niehren et Andreas Podelski – « Inclusion constraints over non-empty sets of trees », *Theory and Practice of Software Development, International Joint Conference CAAP/FASE/-TOOLS* (M. Dauchet, éd.), Lecture Notes in Computer Science, vol. 1214, Springer Verlag, 1997, p. 217–231.

[Mom00] Alberto Momigliano – « Elimination of negation in a logical framework », *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic* (London, UK), Lecture Notes in Computer Science, vol. 1862, Springer Verlag, 2000, p. 411–426.

[Mor99] Pierre-Etienne Moreau – « Compilation de règles de réécritures et de stratégies non-déterministes », Thèse, Université Henri Poincaré - Nancy 1 (France), 1999.

[MRV03] Pierre-Etienne MOREAU, Christophe RINGEISSEN et Marian VITTEK – « A Pattern Matching Compiler for Multiple Target Languages », *12th Conference on Compiler Construction, Warsaw (Poland)* (G. HEDIN, éd.), Lecture Notes in Computer Science, vol. 2622, Springer-Verlag, 2003, p. 61–76.

[New42] Maxwell Herman Alexander NEWMAN – « On theories with a combinatorial definition of equivalence », *Annals of Math*, vol. 43, 1942, p. 223–243.

[Nip90] Tobias NIPKOW – « Proof transformations for equational theories », *Proceedings of the 5th Symposium on Logic in Computer Science (LICS)*, 1990, p. 278–288.

[Nip91] Tobias NIPKOW – « Combining matching algorithms : The regular case », *Journal of Symbolic Computation* **12** (1991), no. 6, p. 633–653.

[OW97] Martin ODERSKY et Philip WADLER – « Pizza into java : translating theory into practice », *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'97)* (New York, NY, USA), 1997, p. 146–159.

[Plo72] Gordon PLOTKIN – « Building-in equational theories », *Machine Intelligence* **7** (1972), p. 73–90.

[PS81] Gerald E. PETERSON et Mark E. STICKEL – « Complete sets of reduction for some equational theories », *Journal of the ACM* **28** (1981), p. 233–264.

[PS93] Laurence PUEL et Ascander SUAREZ – « Compiling pattern matching by term decomposition », *Journal of Symbolic Computation* **15** (1993), no. 1, p. 1–26.

[Rei06a] Antoine REILLES – « Canonical abstract syntax trees », *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications*, vol. 176, Electronic Notes in Theoretical Computer Science, 2006, p. 165–179.

[Rei06b] — , « Réécriture et compilation de confiance », Thèse de Doctorat d'Université, Institut National Polytechnique de Lorraine, France, 2006.

[Rin96] Christophe RINGEISSEN – « Combining decision algorithms for matching in the union of disjoint equational theories », *Information and Computation* **126** (1996), no. 2, p. 144–160, Journal version of the technical report 93-R-249.

[Ses96] Peter SESTOFT – « ML pattern match compilation and partial evaluation », *Partial Evaluation*, Lecture Notes in Computer Science, vol. 1110, Springer-Verlag, 1996, p. 446–464.

[SGC08] Don SYME, Adam GRANICZ et Antonio CISTERNINO – *Expert F# (expert's voice in .net)*, APRESS, 2008.

[SNM07] Don SYME, Gregory NEVEROV et James MARGETSON – « Extensible pattern matching via a lightweight language extension », *Proceedings of the*

12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007), ACM, 2007, p. 29–40.

[SR00] Kevin Scott et Norman Ramsey – « When do match-compilation heuristics matter ? », Tech. report, Charlottesville, VA, USA, 2000.

[SS96] Wayne Snyder et James G. Schmolze – « Rewrite semantics for production rule systems : Theory and applications », *Proceedings of the 13th International Conference on Automated Deduction (CADE-13)* (London, UK), Lecture Notes in Computer Science, vol. 1104, Springer-Verlag, 1996, p. 508–522.

[Tre92] Ralf Treinen – « A new method for undecidability proofs of first order theories », *Journal of Symbolic Computation* **14** (1992), no. 5, p. 437–457.

[VBT98] Eelco Visser, Zine-el-Abidine Benaissa et Andrew Tolmach – « Building program optimizers with rewriting strategies », *Proceedings of the third ACM SIGPLAN international conference on Functional programming* (NY, USA), ACM Press, 1998, p. 13–26.

[vdBHdJ+01] Mark van den Brand, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter Olivier, Jeroen Scheerder, Jurgen Vinju, Eelco Visser et Joost Visser – « The ASF+SDF Meta-Environment : a Component-Based Language Development Environment », *Compiler Construction* (R. Wilhelm, éd.), Lecture Notes in Computer Science, vol. 2027, Springer-Verlag, 2001, p. 365–370.

[vdBHKO02] Mark van den Brand, Jan Heering, Paul Klint et Pieter Olivier – « Compiling language definitions : the asf+sdf compiler », *ACM Transactions on Programming Languages and Systems (TOPLAS)* **24** (2002), no. 4, p. 334–368.

[vdBKO99] Mark van den Brand, Paul Klint et Pieter Olivier – « Compilation and Memory Management for ASF+SDF », *Compiler Construction*, Lecture Notes in Computer Science, vol. 1575, Springer-Verlag, 1999, p. 198–213.

[vdBMV05] Mark van den Brand, Pierre-Etienne Moreau et Jurgen Vinju – « A generator of efficient strongly typed abstract syntax trees in java », *IEE Proceedings - Software Engineering* **152** (2005), no. 2, p. 70–78.

[vdBvDK+96] Mark van den Brand, Arie van Deursen, Paul Klint, Steven Klusener et Emma van der Meulen – « Industrial applications of ASF+SDF », *AMAST '96* (M. Wirsing et M. Nivat, éds.), Lecture Notes in Computer Science, vol. 1101, Springer-Verlag, 1996, p. 9–18.

[Vis01] Joost Visser – « Visitor combination and traversal control », *Proceedings of the 16th ACM SIGPLAN conference on OOPSLA* (NY, USA), ACM Press, 2001, p. 270–282.

[Vit94] Marian Vittek – « ELAN : Un cadre logique pour le prototypage de langages de programmation avec contraintes », Thèse, 1994.

[Wol99] Stephen Wolfram – « The mathematica book », ch. Patterns, Transformation Rules and Definitions, Cambridge University Press, 1999, ISBN 0-521-64314-7.

[ZLCD05] Wei Zhang, Tok Wang Ling, Zhuo Chen et Gillian Dobbie – « Xdo2 : A deductive object-oriented query language for xml. », *DASFAA* (L. Zhou, B. C. Ooi et X. Meng, éds.), Lecture Notes in Computer Science, vol. 3453, Springer, 2005, p. 311–322.

Table des figures

1.1. Fonctionnement global du système TOM 13
1.2. Aperçu des contributions 16

3.1. $\mathcal{A}$-Match: p_i are patterns, t_i are ground terms, and S is any conjunction of matching equations. Mutate is the most interesting rule, and it is a direct consequence of the fact that associativity is a syntactic theory. $\wedge, \vee$ are classical boolean connectors. 41
3.2. Simplified presentation of the disunification rules: AntiMatchDisunif 49
3.3. The complete set of rules for solving anti-pattern matching problems via disunification: we first normalize with ElimAnti, second we replace the symbol $\prec\!\!\!\prec$ with $=$ and we normalize with DeMorgan rules. We finally normalize with the disunification rules. 53
3.4. $\mathcal{AU}$-AntiMatch 54

4.1. La syntaxe de nouveau **%match**. 66
(a). Le nouveau **%match**. 66
(b). La syntaxe simplifiée des conditions. 66
4.2. Élimination des parenthèses dans une condition 68
4.3. Un exemple d'utilisation des contraintes booléenes pour augmenter l'efficacité par rapport au code JAVA équivalent. 72
(a). Un **%match** avec contraintes booléenes 72
(b). Le codage des contraintes booléenes en JAVA 72

5.1. A general view of the way TOM system works 78
5.2. Inside view of the TOM system 79
5.3. The syntax of the intermediate language `PIL`. 82
5.4. The general architecture of the TOM compiler 86
5.5. A simple example of the compilation process. The propagator decomposes the initial problem in smaller constraints, which are then passed to the scheduler that arranges them in the good order. The generator further translates these constraints into `PIL` code. 87
5.6. The simplified abstract syntax of the compiler's input conditions. $\langle pattern \rangle$ is a TOM pattern, and $\langle term \rangle$ is a term built with constructors, JAVA variables, JAVA function calls and variables that are used in TOM patterns. . 88
5.7. Transformation rules of the syntactic propagator 89
5.8. Transformation rule of the associative propagator 90
5.9. Propagator's output syntax. $\mathtt{f} \in \mathcal{F}$, $\mathtt{n} \in \mathbb{N}$. 91

5.10. The correspondence between constraints and `PIL` instructions. If the constraint being transformed is the last in the conjunction, we can consider c as being the action of the rule. 95

6.1. Exemple d'une hiérarchie de classes JAVA. `AccountOwner` contient une liste de `Account`. Chaque `Account` a un `AccountOwner`. `BankAccount`, `CreditCardAccount`, et `BrokerageAccount` héritent de `Account`. 102
6.2. La correspondance entre les *mappings* TOM et les classes JAVA. 105
6.3. Le générateur de *mappings* dans ECLIPSE, lancé sur le répertoire `build/account` 107