Optimisation du filtrage par transformations de programmes

Emilie Balland

Optimisation du filtrage par transformations de programmes

Rapport

Stage soutenu le 27 Juin 2005

dans le cadre de la

Maitrise IUP GMI de l'Université Henri Poincaré – Nancy I

par

Emilie Balland

Membres du Jury

Encadrant: Pierre-Etienne Moreau

Membres: N. Carbonnell
D. Galmiche
D. Méry

Mis en page avec la classe thloria.

Table des matières

Introduction

Il y a vingt ans, la première nécessité du programmeur était d'écrire du code efficace et minimisant la quantité de mémoire utilisée. Aujourd'hui, ses priorités ont changées. Plus affranchi des contraintes matérielles, il privilégie la sécurité et l'évolutivité : le code doit être lisible, modulaire et sûr. Pour cela il utilise des langages de haut niveau, permettant de mélanger les instructions avec des notions plus formelles que sont les conditions, les invariants, ou plus généralement les propriétés sémantiques.

La tâche d'efficacité est laissée au compilateur qui se doit aussi de garantir la correction du code produit.

Au centre des méthodes de compilation et d'optimisation, le concept de *transformations de programmes* tient un rôle important. Il consiste à écrire une première version correcte du programme source, sans forcément se préoccuper de son efficacité et ensuite à le dériver par pas de transformation successifs en un programme plus efficace. L'objectif étant évidemment de préserver la sémantique du programme source à chaque pas de transformation.

On peut classer les transformations de programmes en deux grandes catégories :

- les reformulations pour lesquelles le langage du programme cible est identique au langage du programme source. Il s'agit en particulier de normalisation, d'optimisation et de re-engineering de programmes.
- les traductions pour lesquelles le langage du programme cible est différent du langage du programme source. On parle alors de compilation ou de reverse-engineering.

Le processus de *compilation* peut être mis en œuvre par une suite de transformations de programmes. Les avantages de cette approche sont de décomposer et de structurer la compilation, de simplifier les preuves de correction et de permettre des comparaisons formelles en étudiant chaque transformation ou leur composition. Les langages basés sur la réécriture tels que Maude [CELM96],ELAN [BHKMR98] ou Tom [MRV03] sont adaptés pour des transformations locales et la correction de la compilation de ces langages est primordiale puisqu'ils sont souvent à la base de spécification formelle. L'approche par transformations de programmes est très appropriée à ce type de langages. C'est pourquoi on souhaite l'appliquer au langage Tom.

La *réécriture* est un concept permettant de modéliser les principes de transformation et de simplification. Elle sert en particulier de fondement au calcul symbolique, à la sémantique opérationnelle des langages de programmation notamment fonctionnelle et à la déduction automatique. La notion de base en réécriture est celle d'égalité orientée, appelée *règle* (souvent notée $g \to d$). Un terme peut se réécrire avec une telle règle si g filtre, c'est à dire s'il contient une instance du membre gauche. Un système de réécriture correspond à un ensemble de règles dont l'application repose sur les opérations de sélection et de filtrage. En pratique, ces deux opérations s'implantent par une suite de tests et d'affectations permettant de calculer une substitution (i.e. les valeurs des variables du membre gauche g).

Dans les langages basés sur la réécriture, le filtrage est omniprésent et son implantation doit être la plus efficace possible. Dans la littérature, il existe de nombreux algorithmes de compilation de filtrage [Grä91, SRR95] qui permettent de factoriser les tests communs entre les différentes règles du système de réécriture. Ces algorithmes permettent de construire des programmes de filtrage efficaces mais ils sont difficiles à faire évoluer ou à adapter à des situations nouvelles, telles que le filtrage dans les langages à objets par exemple.

Ce stage a pour but d'étudier une autre approche : *l'optimisation du filtrage par transformations de programmes*. Au lieu de compiler directement le filtrage en un programme efficace, en utilisant un algorithme de compilation complexe et peu souple, notre démarche est de séparer la phase de compilation

de la phase d'optimisation. Cette approche permet en particulier de prouver la correction de l'algorithme de filtrage.

L'étape d'optimisation consiste à améliorer les performances en temps d'exécution ou en espace mémoire tout en préservant la sémantique du programme initial. Comme la compilation, elle peut être réalisée par transformations de programmes.

Durant ce stage, nous nous sommes inspirés des techniques d'optimisations par transformations de programmes et nous avons proposé une solution adaptée aux programmes de filtrage du logiciel Tom, développé dans l'équipe Protheo. Après avoir présenté ces techniques ainsi que le logiciel Tom, nous définirons un système de transformations applicable aux programmes de filtrage syntaxique et nous l'étendrons aux programmes de filtrage équationnel. Enfin, nous présenterons l'implantation de ce système de règles en Tom et les gains de performance obtenus.

Chapitre 1

Problématique

Dans cette étude, nous allons nous intéresser aux langages basés sur la réécriture (section 1.1), dont le mécanisme principal d'exécution est le filtrage. Il est ainsi primordial que son implantation soit la plus efficace possible. Les méthodes existantes reposent généralement sur l'utilisation d'automates (section 1.2). L'inconvénient principal de ces méthodes est le manque de modularité. Pour y remédier, nous avons choisi une autre approche qui consiste à compiler le filtrage sans se préoccuper de l'efficacité et ensuite d'optimiser le code généré. Cela nous a naturellement amené à étudier les méthodes de transformation source-à-source, et plus particulièrement les travaux d'E. Visser [VeABT98] et de D. Lacey [Lac03]. Ces deux approches seront à la base de cette étude.

1.1 Réécriture

La réécriture est basée sur des ensembles d'égalités orientées appelées règles. Il existe deux grands domaines d'utilisation de la réécriture :

- la déduction automatique : prouveur de théorèmes basé sur des règles, résolution de contraintes, résolution d'équations,
- les langages de programmation : la réécriture peut être un moyen de définir la sémantique d'un langage de programmation ou encore être directement à la base de langages de programmation, comme par exemple ELAN [BHKMR98], Maude [CELM96], ou encore d'une certaine façon les langages fonctionnels tels que ML, Caml ou Haskell.

Nous présentons d'abord les notions de base de la réécriture puis les langages basés sur ce paradigme.

1.1.1 Systèmes de réécriture

Signature. Une *signature* $\mathcal{F}$ est un ensemble de symboles de fonction. On définit l'application retournant l'arité d'un symbole de fonction $ar : \mathcal{F} \mapsto \mathbb{N}$ et on note $\mathcal{F}_0$ l'ensemble des symboles de fonctions d'arité 0, autrement dit l'ensemble des constantes.

Terme. Les symboles définis dans la signature peuvent être utilisés pour construire des termes.

Étant donné un ensemble de symboles de fonctions $\mathcal{F}$ et un ensemble de variables $\mathcal{X}$, l'ensemble des termes $\mathcal{T}(\mathcal{F}, \mathcal{X})$ correspond au plus petit ensemble contenant $\mathcal{X}$ et $f(t_1, \ldots, t_n)$ pour toute fonction f et $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, pour $i \in [1..n]$.

On définit l'application $Var : \mathcal{T}(\mathcal{F}, \mathcal{X}) \mapsto \mathcal{P}(\mathcal{X})$ qui renvoie l'ensemble des variables contenues dans un terme.

$\mathcal{S}ymb(t)$ est une fonction partielle de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ dans $\mathcal{F}$, qui associe à chaque terme t son symbole de tête $f \in \mathcal{F}$.

Un terme est dit *clos* s'il ne contient pas de variable et l'ensemble des termes clos se note $\mathcal{T}(\mathcal{F})$.

Les termes clos t et u de $\mathcal{T}(\mathcal{F})$ sont égaux (noté $t = u$), si il existe un symbole de fonction f et des termes $t_1, \ldots, t_n$, $u_1, \ldots, u_n$ tels que $\mathcal{S}ymb(t) = \mathcal{S}ymb(u) = f$, $f \in \mathcal{F}_n$, $t = f(t_1, \ldots, t_n)$, $u = f(u_1, \ldots, u_n)$, et $\forall i \in [1..n]$, $t_i = u_i$.

Position. Il est souvent nécessaire en réécriture de pouvoir désigner un sous-terme s d'un terme t. A cette fin, on introduit le concept de position. Une position sous un terme t est représentée par une séquence ω d'entiers positifs décrivant le chemin de la racine de t au sous-terme à cette position. Par exemple dans le terme $f(g(t_1), t_2)$, t_1 a la position 1.1 et t_2 a la position 2.

La position vide, qui correspond à la racine est notée ϵ.

Le sous-terme de t à la position ω est noté $t_{|\omega}$.

Le remplacement dans t, de $t_{|\omega}$ par t' est noté $t[\omega \hookleftarrow t']$.

Substitution. Une substitution σ sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$ est un endomorphisme de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ qui s'écrit $\sigma = (x_1 \mapsto t_1, \cdots, x_n \mapsto t_n)$ lorsque les images de x_i pour $i = 1, \ldots, n$ sont des $t_i \neq x_i$.

Une des propriétés fondamentales des substitutions est que pour tous termes $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et pour tout symbole $f \in \mathcal{F}$:

$$f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma) \quad \text{(propriété d'endomorphisme)}$$

L'application d'une substitution σ au terme t est notée $t\sigma$ ou $\sigma(t)$.

Le simple remplacement dans un terme t_1 d'une variable v par un terme t_2 est noté $t_1[v/t_2]$.

Règle de réécriture. Une règle de réécriture est une paire ordonnée de termes $< l, r >$, respectivement appelés membre gauche et membre droit, et notée $l \rightarrow r$. Une règle conditionnelle est une règle de réécriture gardée par une condition c. Une condition c est définie par une conjonction de prédicats portant sur $Var(l)$. On la note $l \rightarrow r \; if \; c$. Un système de réécriture est un ensemble de règles.

En pratique, une règle de réécriture conditionnelle doit vérifier les contraintes suivantes :
- l n'est pas une variable,
- $Var(r) \subseteq Var(l)$,
- $Var(c) \subseteq Var(l)$.

Application de règles et filtrage. Pour pouvoir appliquer une règle sur un terme clos, appelé sujet, il faut que l'on puisse remplacer les variables de son membre gauche par des termes clos, de telle sorte que ce nouveau membre gauche soit égal au sujet. On dit alors que le membre gauche de la règle *filtre* vers le sujet qui devient un *radical*.

Un terme $t \in \mathcal{T}(\mathcal{F})$ se réécrit en t' dans le système R si il existe :
- une règle $l \rightarrow r \in R$,
- une position ω,
- une substitution σ telle que $t_{|\omega} = \sigma(l)$ et $t' = t[\sigma(r)]_\omega$

Lorsqu'il existe une règle dont le membre gauche filtre vers le sujet, celle-ci peut s'appliquer et réduire le sujet. Le mécanisme d'application d'une règle consiste simplement à remplacer le sujet par le membre droit de la règle sur lequel est appliqué le filtre par substitution.

Pour un système R, lorsqu'aucune règle n'est applicable sur un terme t, on dit que ce terme n'est plus réductible et qu'il est en *forme normale*. On le note $t_{\downarrow R}$.

On peut faire référence par l'étiquette `ref` un sous-terme t d'une règle.

Exemple :

`let`$(var, term,$ `ref`@(`while`$(e, instr)))$ permet de définir une référence `ref` au sous-terme `while`$(e, instr)$.

Stratégie d'application. L'aspect non-déterministe de la réécriture (les règles de réécriture peuvent s'appliquer dans n'importe quel ordre et à n'importe quelle position du terme à réduire) a nécessité l'introduction de la notion de *stratégie* d'application pour mieux contrôler l'application des règles de réécriture.

L'une des stratégies classiques qui nous intéresse est la stratégie de *parcours intérieur gauche* (*leftmost-innermost*) qui sélectionne le radical le plus à gauche et le plus interne à chaque étape de réécriture. Elle est souvent utilisée pour la normalisation.

1.1.2 Langages basés sur la réécriture

Bien que longtemps utilisée comme outil pour raisonner, la réécriture peut également constituer un bon support pour développer un langage de programmation. On peut citer les systèmes de spécification comme ELAN [BHKMR98], Maude [CELM96], ASF+SDF [vdBHdJ+01], ou encore le système Tom [MRV03] permettant d'intégrer de la réécriture dans des langages impératifs.

Le système ELAN, développé dans l'équipe Protheo, est un environnement de spécification et de prototypage. Il est fondé sur l'application de règles de réécriture contrôlée au moyen de stratégies et permet de combiner les paradigmes de calcul et de déduction. ELAN possède un langage de stratégies dont les primitives (opérateurs de séquence, de répétition, ou encore de choix) permettent une gestion fine de l'application des règles et de l'exploration de l'espace de recherche. Maude, initialement développé au SRI, est un langage permettant de définir des règles de réécriture et de contrôler leur application. Contrairement à ELAN, la stratégie d'application des règles se définie au méta-niveau, ce qui rend leur écriture plus difficile. ASF+SDF, développé au CWI, est également un langage basé sur la notion de règle de réécriture. Une de ses originalités est de fournir un formalisme de définition de syntaxes (SDF) très expressif ainsi que des opérateurs de parcours d'arbre, qui font d'ASF+SDF un outil bien adapté à la transformation et la rénovation de programmes.

Dans la suite de ce travail nous nous intéresserons particulièrement au système Tom, qui peut être vu comme une évolution d'ELAN. La particularité de Tom est d'être un pré-compilateur permettant d'ajouter des instructions de filtrage dans des langages hôtes comme Java ou C. Similairement aux trois systèmes mentionnés précédemment, Tom offre un langage permettant de contrôler l'application des règles de réécriture, tout en définissant des stratégies de parcours de termes.

Nous pouvons également noter que les langages fonctionnels, comme Haskell ou Caml, reposent également sur des concepts liés à la réécriture : ils possèdent tous deux une instruction de filtrage.

Pour tous ces systèmes, l'efficacité dépend largement de l'implantation du filtrage. Il est donc primordial d'assurer soit au niveau de la compilation, soit lors d'une phase d'optimisation, un filtrage performant.

1.2 Compilation du filtrage

Une première approche naïve consiste à considérer les règles une à une pour savoir si elles peuvent s'appliquer. Ces méthodes sont dites *one-to-one* parce que les problèmes de filtrage ne font intervenir à la fois qu'une seule règle de réécriture et un seul sujet. Cette approche est peu efficace parce que sa complexité est proportionnelle au nombre de règles composant le système. C'est pourquoi des méthodes dites *many-to-one* ont été développées [Grä91, SRR95] : elles permettent de sélectionner efficacement une règle du système afin de réduire le sujet. La compilation du filtrage consiste alors à représenter les règles sous forme d'automate, d'optimiser cet automate puis de générer le code correspondant.

1.2.1 Automates de filtrage déterministes

On peut traduire les règles sous forme d'un automate de type DFA (Deterministic Finite Automaton). Cet automate déterministe est basé sur une traversée de gauche à droite des sujets. Il permet de sélectionner les règles qui filtrent en évaluant chaque position du terme une seule fois car il ne remet jamais en question les transitions effectuées. On peut citer les travaux de L.Cardelli [Car84] et de A.Graf [Grä91]. L'implantation du filtrage dans ELAN est basée sur un automate de type DFA. Cependant, le principal inconvénient de cette méthode est que l'espace mémoire nécessaire à la représentation de l'automate peut être exponentiel en la taille des règles.

R.C. Sekar s'est intéressé à ce type d'algorithmes et a proposé dans [SRR95] une méthode orthogonale permettant de construire un automate plus petit au niveau de l'espace. Ces améliorations sont basées sur différentes stratégies permettant de choisir un ordre de traversée adapté aux règles, et non plus

systématiquement une évaluation de gauche à droite. Cependant, il a aussi démontré que si l'on souhaite conserver la linéarité en temps (chaque position du terme n'est évaluée qu'une seule fois), la borne inférieure de complexité en espace est forcément exponentielle.

1.2.2 Automates de filtrage non déterministes

Une autre catégorie d'algorithmes est basée sur des automates avec retour arrière (*backtracking*). Cette fois-ci, on s'autorise à évaluer plusieurs fois une même position du terme, à revenir sur nos choix. On n'est donc plus linéaire en temps à cause du backtracking. Mais en contrepartie, l'espace nécessaire à la construction de l'automate peut être linéaire par rapport au nombre de règles.

L'algorithme défini par L. Augustsson [Aug85] initialement utilisé dans le compilateur d'Objective-Caml est basé sur des automates avec backtracking. Les travaux récents de L. Maranget et de F. Le Fessant portent sur l'optimisation du filtrage dans le cadre du compilateur d'Objective-Caml [FM01]. Ils optimisent l'automate avec backtracking produit en factorisant au maximum les tests élémentaires. Pour cela, des suppositions sont faites sur le pouvoir d'expression du langage cible. En particulier, il faut que ce langage ait une instruction `switch` ainsi qu'une instruction de saut. Ces deux instructions permettent de discriminer et de transférer le flot de contrôle en fonction d'un entier : cela permet d'implanter efficacement les transitions d'un automate.

Les deux approches (déterministes et non-déterministes) sont complètement orthogonales. Elles montrent bien qu'il faut forcément trouver un compromis entre linéarité en espace et linéarité en temps. Le principe est ensuite de s'imposer la linéarité sur l'un de ces deux axes et de minimiser la complexité de l'autre tout en sachant qu'on ne peut pas atteindre la linéarité pour les deux.

Quelle que soit l'approche choisie, dans les deux cas il existe de très bons algorithmes permettant de compiler efficacement le filtrage. Il faut cependant noter que ces algorithmes sont adaptés à des formes bien particulières de filtrage et supposent de nombreuses propriétés sur le langage cible ainsi que sur la représentation des termes manipulés. Parce que très pointus, les principaux algorithmes connus s'adaptent mal à des variations telles que la prise en compte de la non-linéarité des motifs ou de contraintes plus complexes au moment du filtrage. D'un point de vue pratique, il faut également que le langage cible ait une instruction de `switch`, ou encore que les symboles apparaissant dans les termes puissent se représenter par un entier. Bien que raisonnables, ces suppositions ne sont pas adaptées au cas où les objets d'un langages à objets sont considérés comme des termes. En effet, dans ce cas, la notion de symbole correspond à la notion de type dynamique, qui n'a pas forcément de représentation (lorsque le langage ne supporte pas une certaine forme de réflexivité).

Ces différentes restrictions nous amènent à étudier une autre approche permettant de concevoir des algorithmes plus modulaires et évolutifs, tout en conservant une certaine efficacité : l'optimisation, a posteriori, du code généré par un algorithme de filtrage simple, supposant moins de propriétés.

1.3 Optimisation par transformations

Optimiser un programme consiste à rendre son implantation la plus efficace possible. Cette efficacité se mesure souvent en temps, mais peut également se mesurer en espace utilisé par exemple. C'est une opération qui se fait en général conjointement à une opération de compilation, qui transforme le programme initial dans une représentation plus bas niveau. Pour cela, on utilise fréquemment des structures annexes, tel que le *graphe de flot de contrôle* (CFG). Ce dernier est une représentation sous forme de graphe du code source : chaque nœud correspond à une instruction du programme. On ne considère pas ici les graphes de flot de contrôle dont les nœuds correspondent à des blocs d'instructions contigus car nous avons besoin d'une représentation avec une granularité plus fine. Les arêtes entre les nœuds représentent les pas possibles dans le programme entre les instructions.

L'opération d'optimisation peut également être vue indépendamment de toute étape de compilation. On parle alors d'optimisation source-à-source parce que le langage de départ et le langage cible sont les mêmes. Pour cela, il est souvent pratique de considérer une représentation abstraite du programme, appelée *arbre syntaxique abstrait* (AST). C'est un arbre où chaque instruction est représentée par un

nœud dont les fils correspondent à des arguments ou de l'information locale à cette instruction. C'est une structure proche du code, les détails de syntaxe en moins. Ces deux types de représentations sont illustrés sur la figure 1.1.

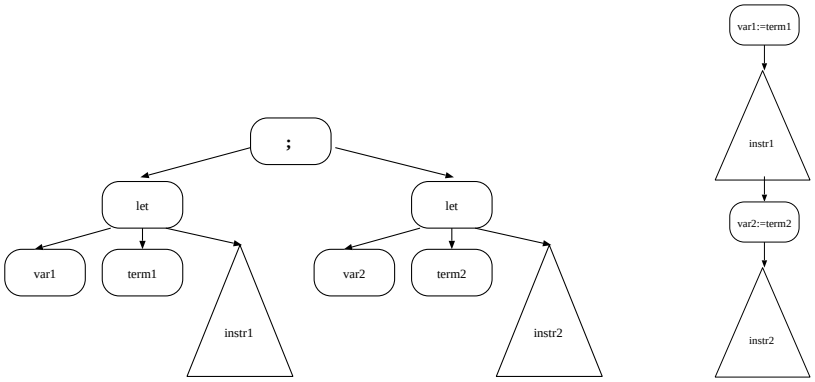

FIG. 1.1 – AST et CFG représentant l'instruction `let`$(var1, term1, instr1)$; `let`$(var2, term2, instr2)$

Dans le cas qui nous intéresse, nous souhaitons réaliser de l'optimisation source-à-source sur des programmes sans effet de bord : le langage cible que nous considérons ne permet pas de modifier la valeur d'une variable (cela correspond au Single Assignement Statement). Dans ce cas, l'optimisation par transformations (réécriture) est un mécanisme parfaitement adapté parce qu'il permet de décrire facilement la recherche et le remplacement de motifs complexes dans un arbre de syntaxe abstraite. Par ailleurs, l'optimisation par transformations est particulièrement modulaire et extensible car elle permet d'implanter, de tester, et d'activer chaque optimisation séparément. L'ordonnancement et la combinaison des différentes règles d'optimisation peut se décrire en utilisant un langage de stratégies.

L'optimisation par transformations a surtout été utilisée pour les langages fonctionnels. En effet, en limitant la possibilité de faire des effets de bords, la structure syntaxique d'un programme fonctionnel est souvent suffisante et permet de faire des optimisations sans avoir recours à des méthodes plus complexes, faisant intervenir des structures d'analyse auxiliaires. Les optimisations souvent effectués sur des langages fonctionnels sont par exemple :

- l'évaluation partielle qui permet de spécialiser un programme suivant ses entrées [DGT96],
- la déforestation qui permet d'éliminer des structures de données intermédiaires et de composer des programmes [Wad88],
- le tupling qui permet d'éliminer des traversées parallèles de structures de données (en quelque sorte la version parallèle de la déforestation) et d'éliminer des sous-expressions communes [Chi93],
- la super-compilation qui permet de minimiser le nombre de traversées de structures de données [Tur86]

Dans notre cas, bien que nous considérions un langage cible fonctionnel (au sens "sans effet de bord"), le code généré par la phase de compilation du filtrage ne se prête pas à ce type d'optimisations (du fait de la présence de nombreuses séquences d'instructions). Cette étude s'est donc autant portée sur les formalismes de définitions de règles (implantation par réécriture des transformations de programmes) que sur les règles proprement dites (optimisations spécifiques aux types de programmes étudiés). Afin de définir un formalisme de définition de règles le plus adapté à notre problème, nous nous sommes naturellement intéressés à la définition de stratégie de réécriture pour l'optimisation [VeABT98] ainsi qu'aux méthodes reposant sur l'utilisation de conditions exprimées en logique temporelle pour contrôler l'application de règles de transformations de programmes [Lac03].

1.3.1 Stratégies de réécriture pour l'optimisation

E. Visser a beaucoup étudié les transformations de programmes. Il a par exemple proposé une classification[1]. Mais il a surtout approfondi l'étude des stratégies de réécriture pour leur application à l'optimi-

[1] Le site http ://www.program-transformation.org répertorie les différentes transformations existantes

sation. En effet, la définition de règles n'est pas suffisante pour garantir la terminaison de l'optimisation et encore moins son efficacité. La stratégie doit ainsi permettre de contrôler l'application des règles.

Dans [Vis01a], Stratego est présenté comme un langage modulaire pour la spécification de systèmes de transformations de programmes complètement automatiques basé sur le paradigme des stratégies de réécriture. À partir de ce langage, il est possible de définir indépendamment les règles en s'assurant de leur correction. Ensuite, on peut utiliser différentes stratégies sur ces règles en utilisant le langage de spécification et ainsi obtenir un optimiseur qui garantit la terminaison au niveau de la stratégie et qui soit plus efficace. Cette méthode permet aussi une meilleure lisibilité et une plus grande souplesse dans la construction d'optimiseurs.

[OV02] présente une définition simple de la propagation de constantes à l'aide de Stratego. Il s'agit d'une optimisation classique consistant à remplacer toutes les occurrences d'une variable lorsqu'elle est définie par une constante.

Dans le cas des langages impératifs, les optimisations de flot de données comme la propagation de constantes, l'inlining ou l'élimination de variable morte sont généralement effectuées sur des représentations intermédiaires bas-niveau proches du programme cible. Ce choix est justifié car leur implantation peut être réutilisée pour tous les langages sources traduits dans ce langage cible. La représentation intermédiaire est donc proche du code machine et définit le programme en terme de sauts, d'accès mémoire et de registres par exemple. La propagation de constantes s'exprime alors sous forme d'information propagée sur un graphe de flot de contrôle. Lorsque l'on souhaite réaliser de l'optimisation source-à-source, cette méthode n'est pas adaptée. C'est pourquoi dans ce document, la propagation de constantes est définie sur une représentation intermédiaire haut-niveau proche du programme source : l'arbre syntaxique abstrait.

Les travaux d'E. Visser sont très intéressants pour notre étude car ils portent aussi sur des transformations de programmes source-à-source et sur une représentation intermédiaire de type arbre syntaxique abstrait. Cependant, l'écriture de règles ne portant que sur l'arbre syntaxique abstrait peut devenir limitative car cette représentation du programme est trop statique. En effet, l'arbre ne représente pas les instructions dans l'ordre de leur exécution mais suivant la syntaxe du langage. Nous nous sommes alors intéressés aux travaux de D. Lacey qui porte cette fois-ci sur une représentation plus dynamique du programme : le graphe de flot de contrôle.

1.3.2 Spécification des conditions des règles à l'aide de logique temporelle

La thèse de D. Lacey présente un langage pour définir des règles de transformations de programmes. Ces règles portent sur le graphe de flot de contrôle de programmes écrits dans un langage impératif. L'aspect novateur de ce travail est l'utilisation dans les conditions des règles de transformations de formules de logique temporelle.

En effet, au lieu d'utiliser des structures d'analyse complexes en plus du graphe de flot de contrôle ou alors de décorer le graphe, l'information est directement calculée à partir des formules temporelles et seulement lorsqu'elle est nécessaire.

Par exemple, une condition pour considérer l'affectation $x := t$ comme du code inutile (x variable morte), et donc l'éliminer, est qu'à partir de l'affectation $x := t$, la variable x n'est plus utilisée. Lorsque le programme est représenté par graphe de flot de contrôle, cela revient à déterminer si le noeud correspondant à $x := t$ vérifie la condition temporelle suivante : $AG(\neg use(x))$. Cette expression est vraie lorsque tous les noeuds atteignables depuis le noeud de $x := t$ vérifient $\neg use(x)$, où $use(x)$ est un prédicat indiquant si la variable x est utilisée par le nœud considéré. Un nœud n vérifie $n \models AG(c)$ si n et tous les nœuds atteignables depuis n vérifient c.

L'inconvénient principal de cette méthode est que toute la règle de transformations de programmes est écrite sur le graphe de flot de contrôle et aucun lien avec le programme de départ n'est donné.

Dans la suite de ce travail nous définissons un formalisme combinant les avantages des deux méthodes présentées précédemment : la règle porte sur l'arbre syntaxique abstrait, mais suivant les cas, la condition peut porter aussi bien sur l'arbre syntaxique abstrait que sur le graphe de flot de contrôle. Avant de définir le système de règles de transformations, nous présentons le logiciel Tom afin de mieux comprendre le contexte et les contraintes liées aux choix de conception de cet outil.

Chapitre 2

Le système Tom

Le principe de notre approche consiste à définir des règles de transformations de programmes dans le but de les appliquer sur le langage intermédiaire utilisé par Tom [2]. Tom est un pré-compilateur permettant d'ajouter des instructions de filtrage dans des langages hôtes. L'intérêt est de pouvoir combiner un environnement de programmation impératif et la puissance des langages de filtrage. Dans cette partie sont présentées le langage et le logiciel Tom puis les motivations de l'approche par transformations compte-tenu des contraintes de Tom.

2.1 Présentation du langage Tom

Un programme Tom est un programme écrit dans un langage hôte comme par exemple C, Java ou Caml et qui comporte de nouvelles constructions comme `%match`, `%rule`, ou `%include`. Du point de vue de Tom, un programme est une liste de blocs correspondant à des constructions Tom et des séquences de caractères. Une fois compilés, les blocs de constructions Tom se combinent avec le code hôte pour former un programme hôte.

2.1.1 Exemple

En JTom (Java +Tom), on peut écrire le programme suivant :

```
public class HelloWorld {//classe java

%include { string.tom } //code tom
//inclut des définitions de fonctions sur les chaînes de caractères

  public String getWord(String t) { // fonction java
    %match(String t) { //code tom de filtrage d'une chaîne de caractères
      "World"  ->  { return "World";} //règle de filtrage du mot ``World'' et code java correspondant
       _       ->  { return "Unknown";} //règle de filtrage qui filtre quelque soit le sujet
  }
  }

  public final static void main(String[] args) {//fonction java principale
    HelloWorld o = new HelloWorld();
    System.out.println(``Hello ''+o.getWord("World"));
    //exécution de la méthode de filtrage avec le sujet ``World'',affichage de ``Hello World''
  }
}
```

[2] http ://tom.loria.fr

Les constructions `%include` et `%match` sont remplacées, après compilation, par des définitions de fonctions et des instructions Java.

2.1.2 Construction %match

C'est la construction la plus importante de Tom. On peut la voir comme une extension du `switch/case` de Java ou de C. La différence étant que les motifs ne sont plus simplement des constantes (caractères ou entiers). Elle correspond à l'opération de filtrage des systèmes de réécriture comme le `match` de Caml.

Une construction `%match` est composée de deux parties : une liste de variables du langage hôte (appelées *sujets*) et une une liste de règles (`PatternAction`) correspondant à des couples (*pattern*,*action*), où une *action* est un ensemble d'instructions du langage hôte qui est exécuté à chaque fois que le motif *pattern* filtre avec les sujets. La construction est définie ainsi :

```
MatchConstruct   : := '%match' '(' MatchArguments ')' '{' ( PatternAction )* '}'
MatchArguments   : := SubjectName ( ',' SubjectName )*
PatternAction    : := TermList '->' '{' HostCode '}'
TermList         : := Term ( ',' Term )*
```

À l'exécution les sujets sont des termes clos. La construction `%match` est évaluée de la manière suivante :
- le contrôle de l'exécution est transféré au premier `PatternAction` pour lequel le motif filtre avec la liste des sujets. Les variables du motif sont alors instanciées, en fonction de la substitution calculée, et le code hôte correspondant à l'action est exécuté,
- si lors de l'exécution de l'action, le contrôle est transféré en dehors du `%match` (par un `goto`,`break` ou `return` par exemple), le filtrage est terminé. Sinon, le contrôle est transféré au prochain `PatternAction`, et la règle suivante est essayée,
- lorsqu'il n'y a plus de `PatternAction` dont le motif filtre le sujet, l'instruction `%match` est terminée et le contrôle est transférée à l'instruction suivante.

Informellement, un motif (`Term`) peut être une variable telle que `x` ou `y`, une variable anonyme (`_`), une constante telle que `a()` ou `"b"`, ou un terme tel que `h(a(),_,x)`. Les variables n'ont pas besoin d'être déclarées, leur type est automatiquement déduit du contexte dans lequel elles apparaissent.

2.1.3 Construction %rule

La construction `%rule` permet de définir un ensemble de règles de réécriture conditionnelles dont les membres gauches commencent par un même symbole :

```
<RuleConstruct>   : := '%rule' '{' ( <Rule> )* '}'
<Rule>            : :=  <Term> '->' <Term> [ 'if' <Term> '==' <Term> ]
```

Le code généré pour cette instruction correspond à une fonction avec un nombre d'arguments égal à l'arité du symbole racine et dont le nom correspond au nom de ce symbole. L'application de cette fonction sur un terme retourne le membre droit (instancié par la substitution) de la première règle (du haut vers le bas) dont le membre gauche filtre, et dont la condition est satisfaite. Lorsque aucune règle ne s'applique, c'est le sujet qui est retourné par la fonction.

2.2 Architecture du logiciel Tom

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

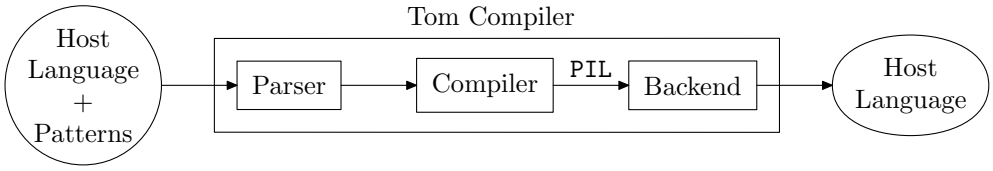

FIG. 2.1 – Chaîne de compilation de Tom

Les différents arbres syntaxiques abstraits sont représentés sous forme d'ATerm [vdBdJKO00] Cette implantation des termes permet un partage maximal des structures. En effet, si un sous-terme apparaît plusieurs fois dans un terme, il n'est représenté qu'une seule fois et partagé pour les différentes positions. C'est cette structure qui est manipulée tout le long de la chaîne de compilation.

L'intégration d'un optimiseur dans le compilateur se réalise facilement en intégrant un nouveau composant dans la chaîne de compilation, entre le *compiler* et le *backend*.

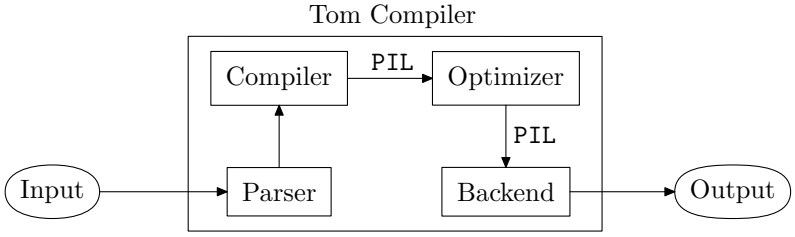

FIG. 2.2 – Intégration de l'optimiseur dans la chaîne de compilation de Tom

Dans la chaîne de compilation de Tom, l'optimiseur transforme le programme `PIL` compilé en un programme `PIL` optimisé. Ce programme est ensuite traduit en programme hôte par le *backend*. Il s'inscrit donc bien dans le cadre de transformations source-à-source.

2.3 Contraintes et motivations

Tom étant un compilateur multi-langages, `PIL` doit être le plus générique possible afin que chacune de ses instructions puissent être traduites dans n'importe quel langage. Par exemple, le filtrage est souvent compilé par une instruction `switch/case` (c'est le cas pour Objective-Caml) mais pour Tom, c'est l'instruction `IfThenElse` qui a été choisie car elle est présente dans n'importe quel langage de programmation. Le mécanisme d'évaluation du `switch/case` est très spécifique au langage, il consiste généralement à représenter le sujet par à un nombre et par dichotomie à retrouver les règles qui filtrent celui-ci. Certaines optimisations réalisées dans [FM01] ne peuvent pas être adaptées pour Tom car elles sont fortement liées à l'utilisation du `switch/case`. Puisque ce sont des règles de filtrage qui sont compilées et que le code est généré automatiquement, il diffère grandement de ce qu'un programmeur serait susceptible d'écrire. On a ainsi besoin d'optimisations très spécifiques comme par exemple la fusion ou la permutation de blocs. Un programmeur n'écrira pas deux blocs conditionnels contigus gardés par la même condition par exemple. Ce type d'optimisations n'étant pas beaucoup étudié dans la littérature, on souhaite s'inspirer des formalismes utilisés dans l'optimisation de programmes tout en définissant des règles spécifiques au problème du filtrage et des programmes `PIL`. De plus, on s'interdit de dupliquer du code pour conserver la linéarité en espace de la compilation. Comme il nous est impossible d'être aussi linéaire en temps, le rôle de l'optimiseur est de limiter au mieux la durée d'exécution. Enfin, la compilation des instructions `%match` et `%rule` conserve l'ordre d'application des règles. L'optimisation doit aussi préserver cet ordre.

Les contraintes sur la définition des règles sont de plusieurs types :

- théoriques (généricité du langage intermédiaire et conservation de la linéarité en espace)
- pratiques (langage intermédiaire fixé pour l'outil, préservation de l'ordre lors du filtrage)

Pour définir un formalisme de règles d'optimisations de programmes permettant de raisonner à la fois sur l'arbre syntaxique abstrait (donc directement sur les programmes `PIL`) comme dans les travaux d'E.Visser (définition de règles et d'une stratégie efficace et terminante sur l'arbre syntaxique abstrait) et sur le graphe de flot de contrôle comme D.Lacey (définition de règles avec conditions temporelles sur le graphe de flot de contrôle), il va falloir définir la construction du graphe de flot de contrôle à partir du programme `PIL` et le lien entre les deux structures. L'objectif est de réussir à obtenir un algorithme de filtrage efficace en séparant complètement les phases de compilation et d'optimisation pour qu'elles restent simples, modulaires et extensibles.

Chapitre 3

Définition des optimisations de programmes PIL

Dans ce chapitre, nous allons présenter un ensemble d'optimisations sur les programmes écrits en PIL, le langage intermédiaire de Tom. Les premières optimisations présentées sont des optimisations classiques comme l'inlining et l'élimination de code mort. Dans un second temps, d'autres optimisations plus spécifiques au code généré sont définies comme par exemple la fusion ou l'entrelacement de blocs d'instructions. Pour l'instant, les conditions des règles ne portent que sur l'arbre syntaxique abstrait. Une partie est ensuite consacrée à la correction des règles de transformation et un exemple de preuves de correction est donnée. À partir de cet ensemble correct de règles de transformations de programmes, nous définissons une stratégie permettant de rendre terminante et efficace l'optimisation et ainsi obtenir des programmes de filtrage syntaxique efficaces à partir de programmes générés simplement.

3.1 Définition du langage intermédiaire

Le langage intermédiaire utilisé dans l'arbre syntaxique abstrait dispose à la fois de traits fonctionnels (instruction d'affectation : $\texttt{let}(variable, term, instr)$) et impératifs (instruction de séquence : $instr; instr$).

3.1.1 Syntaxe de PIL

Étant donnés $\mathcal{F}$, l'ensemble des symboles de fonction, $\mathcal{X}$, l'ensemble des variables, $\mathbb{B}$, les booléens (`true` et `false`), et $\mathbb{N}$, l'ensemble des entiers, la syntaxe de PIL est définie sur la Figure 3.1.

$$
\begin{array}{lcl}
\texttt{PIL} & ::= & \langle instr \rangle \\
\textbf{symbol} & ::= & f \in \mathcal{F} \\
\textbf{variable} & ::= & x \in \mathcal{X} \\
\langle term \rangle & ::= & t \in \mathcal{T}(\mathcal{F}) \\
& | & \textbf{variable} \\
& | & \texttt{subterm}_f(\langle term \rangle, n) \\
& & (f \in \mathcal{F} \wedge n \in \mathbb{N})
\end{array}
\qquad
\begin{array}{lcl}
\langle expr \rangle & ::= & b \in \mathbb{B} \\
& | & \texttt{eq}(\langle term \rangle, \langle term \rangle) \\
& | & \texttt{is\_fsym}(\langle term \rangle, \textbf{symbol}) \\
& | & \texttt{or}(\langle expr \rangle, \langle expr \rangle) \\
& | & \texttt{and}(\langle expr \rangle, \langle expr \rangle) \\
& | & \neg(\langle expr \rangle) \\
\langle instr \rangle & ::= & \texttt{let}(\textbf{variable}, \langle term \rangle, \langle instr \rangle) \\
& | & \texttt{if}(\langle expr \rangle, \langle instr \rangle, \langle instr \rangle) \\
& | & \langle instr \rangle; \langle instr \rangle \\
& | & \texttt{hostcode}(\textbf{variable}*) \\
& | & \texttt{nop}
\end{array}
$$

FIG. 3.1 – Syntaxe de PIL

La Figure 3.2 montre un exemple de programme écrit en Tom ainsi qu'un exemple de code PIL qui aurait put être généré par Tom.

Exemple de code Tom :

```
%match(Term t) {
      f(a) ⇒ {println(hostcode1});}
      g(b)⇒ {println(hostcode2);}
      f(b) ⇒ {println(hostcode3);}
}
```

Exemple de code généré :

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

FIG. 3.2 – Exemple de code PIL généré par Tom

3.1.2 Correction d'un programme PIL

Parmi les programmes PIL, nous allons considérer les programmes dont l'exécution se terminent toujours par l'exécution de l'instruction `hostcode` ou `nop`.

Définition 1. *Un programme $\pi \in$ PIL est dit bien formé s'il satisfait toutes les propriétés suivantes :*

- *une variable apparaissant dans une sous-expression a été auparavant initialisée par la construction* `let`,
- *une variable ne peut être définie que si elle se nomme différemment des variables définies auparavant dans l'arbre syntaxique abstrait,*
- *une expression* $\texttt{subterm}_f(t, n)$ *est toujours encapsulée par un* $\texttt{if}(\texttt{is\_fsym}(t, f), \ldots)$*, et telle que* $n \in [1..ar(f)]$.

Les programmes bien formés correspondent aux programmes dont l'évaluation ne mènera pas à une erreur (accès à une variable non initialisée, ou accès à un sous-terme n'existant pas par exemple). Déterminer si un programme PIL est bien formé est linéairement décidable. La preuve se fait comme dans [KMR05], en utilisant un système d'inférence tel que tout programme PIL pouvant être inféré par ce système corresponde à un programme PIL bien formé.

La Figure 3.3 décrit une version étendue du système de [KMR05] où nous avons ajouté les opérateurs `or`, `and` et $\neg$. L'opérateur $\texttt{Gen}(\Delta, e)$ est défini par :

$$\texttt{Gen}(\Delta, e) = \begin{cases} \Delta; (t, f) \text{ if } e = \texttt{is\_fsym}(t, f) \\ \Delta \text{ sinon} \end{cases}$$

Les contextes Γ, Δ permettent de stocker les informations relatives aux variables. Γ contient toutes les variables définies par un `let` ou un `letRef` (cette information est en prévision de l'extension du système présentée Chapitre 4). Δ contient des couples (terme/symbole de tête) quand on connaît le symbole de tête du terme. Cette information est inférée lorsqu'on se trouve dans une branche gardée par un $\texttt{is\_fsym}(t, f)$. Cela permet de vérifier que $\texttt{subterm}_f$ est uniquement utilisé sur un terme ayant f comme symbole de tête.

3.1.3 Sémantique de PIL

Nous définissons la sémantique de PIL de la même manière que dans [KMR05] par une sémantique à grand pas *à la Kahn* [Kah87]. La relation de réduction d'une sémantique à grand pas est exprimée par des couples formés d'un environnement et d'une instruction et notés $\langle \epsilon, i \rangle$.

$$\frac{}{\Gamma, \Delta \vdash t \ (t \in \mathcal{T}(\mathcal{F}))} \qquad \frac{\Gamma, \Delta \vdash \mathtt{t}}{\Gamma, \Delta \vdash \mathtt{subterm}_f(\mathtt{t}, i)} \text{ if } (\mathtt{t}, f) \in \Delta \text{ and } i \in [1..\mathrm{ar}(f)]$$

$$\frac{}{\Gamma, \Delta \vdash x \ (x \in \mathcal{X})} \qquad \frac{\Gamma, \Delta \vdash \mathtt{t}_1 \quad \Gamma, \Delta \vdash \mathtt{t}_2}{\Gamma, \Delta \vdash \mathtt{eq}(\mathtt{t}_1, \mathtt{t}_2)}$$

$$\frac{}{\Gamma, \Delta \vdash n \ (n \in \mathbb{N})} \qquad \frac{\Gamma, \Delta \vdash \mathtt{t} \quad \Gamma, \Delta \vdash f}{\Gamma, \Delta \vdash \mathtt{is\_fsym}(\mathtt{t}, f)}$$

$$\frac{}{\Gamma, \Delta \vdash b \ (b \in \mathbb{B})} \qquad \frac{\Gamma, \Delta \vdash e \quad \Gamma, \Delta :: \mathtt{Gen}(\Delta, e) \vdash i_1 \quad \Gamma, \Delta \vdash i_2}{\Gamma, \Delta \vdash \mathtt{if}(e, i_1, i_2)}$$

$$\frac{\Gamma, \Delta \vdash e}{\Gamma, \Delta \vdash \neg e} \qquad \frac{\Gamma, \Delta \vdash i_1 \quad \Gamma, \Delta \vdash i_2}{\Gamma, \Delta \vdash i_1; i_2}$$

$$\frac{\Gamma, \Delta \vdash e_1 \quad \Gamma, \Delta \vdash e_2}{\Gamma, \Delta \vdash \mathtt{or}(e_1, e_2)} \qquad \frac{\Gamma, \Delta \vdash \mathtt{t} \quad \Gamma :: v, \Delta \vdash i}{\Gamma, \Delta \vdash \mathtt{let}(v, \mathtt{t}, i)} \text{ if } (v, \_) \notin \Gamma$$

$$\frac{\Gamma, \Delta \vdash e_1 \quad \Gamma, \Delta \vdash e_2}{\Gamma, \Delta \vdash \mathtt{and}(e_1, e_2)} \qquad \frac{}{\Gamma, \Delta \vdash \mathtt{hostcode}(\mathtt{list})} \text{ if } \forall v \in \mathtt{list} : v \in \Delta$$

FIG. 3.3 – Système d'inférence pour vérifier la validité d'un programme PIL

Un environnement représente l'état de la mémoire d'un programme durant son évaluation. Nous représentons donc ici un environnement par une pile d'assignements de termes à des variables. Un environnement ϵ correspond à une composition d'assignements de $\mathcal{X}$ vers $\mathcal{T}(\mathcal{F})$, $[x_1 \leftarrow t_1][x_2 \leftarrow t_2] \cdots [x_k \leftarrow t_k]$. Son application est telle que :

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

La relation de réduction est définie de la manière suivante :

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

Nous avons souhaité garder la même relation de réduction que [KMR05] même si dans notre cas, on remarque que $i' = \mathtt{nop}$.

Pour définir la sémantique des instructions, nous avons besoin de définir l'évaluation d'une expression $e \in \langle expr \rangle$ notée $\lceil e \rceil$. Chaque expression (de type $\langle expr \rangle$) s'évalue par `false` ou `true`.

$$\begin{array}{rclrcl} \lceil \mathtt{true} \rceil & = & \mathtt{true} & \lceil \mathtt{false} \rceil & = & \mathtt{false} \\ \lceil \mathtt{eq}(t_1, t_2) \rceil & = & t_1 = t_2 & \lceil \mathtt{is\_fsym}(t, f) \rceil & = & \mathcal{S}ymb(t) = f \\ \lceil \mathtt{subterm}_f(t, i) \rceil & = & t_{|i} \text{ si } \mathcal{S}ymb(t) = f & \lceil \mathtt{and}(e_1, e_2) \rceil & = & \lceil e_1 \rceil \wedge \lceil e_2 \rceil \\ \lceil \mathtt{or}(e_1, e_2) \rceil & = & \lceil e_1 \rceil \vee \lceil e_2 \rceil & \lceil \neg e \rceil & = & \neg_{\mathbb{B}} \lceil e \rceil \end{array}$$

Les symboles $\wedge$, $\vee$ et $\neg_{\mathbb{B}}$ correspondent aux connecteurs classiques de logique des prédicats. Pour simplifier, on les applique directement aux expressions `true` et `false` de PIL afin d'éviter de les redéfinir pour PIL.

La sémantique de PIL est définie par les règles de la figure 3.4

La sémantique de ce langage est assez naturelle. Il faut tout de même noter certaines particularités utiles pour la définition des règles de transformations.

Variable. Les variables sont définies par l'instruction `let` et ne peuvent jamais être modifiées. La valeur donnée en paramètre est définitive. Leur portée correspond au bloc d'instructions donné en paramètre du `let`.

$$\frac{}{\langle \epsilon, \texttt{nop}\rangle \mapsto_{bs} \langle \epsilon, \texttt{nop}\rangle} \quad (nop) \qquad \frac{}{\langle \epsilon, \texttt{hostcode(list)}\rangle \mapsto_{bs} \langle \epsilon, \texttt{nop}\rangle} \quad (hostcode)$$

$$\frac{\langle \epsilon, i_1\rangle \mapsto_{bs} \langle \epsilon', i\rangle}{\langle \epsilon, \texttt{if}(e, i_1, i_2)\rangle \mapsto_{bs} \langle \epsilon', i\rangle} \text{ if } \lceil \epsilon(e)\rceil = \texttt{true} \quad (iftrue) \qquad \frac{\langle \epsilon, i_2\rangle \mapsto_{bs} \langle \epsilon', i\rangle}{\langle \epsilon, \texttt{if}(e, i_1, i_2)\rangle \mapsto_{bs} \langle \epsilon', i\rangle} \text{ if } \lceil \epsilon(e)\rceil = \texttt{false} \quad (iffalse)$$

$$\frac{\langle \epsilon[x \leftarrow t], i_1\rangle \mapsto_{bs} \langle \epsilon', i\rangle}{\langle \epsilon, \texttt{let}(x, u, i_1)\rangle \mapsto_{bs} \langle \epsilon', i\rangle} \text{ if } \epsilon(u) = t \quad (let) \qquad \frac{\langle \epsilon, i_1\rangle \mapsto_{bs} \langle \epsilon', \texttt{nop}\rangle \quad \langle \epsilon', i_2\rangle \mapsto_{bs} \langle \epsilon'', i\rangle}{\langle \epsilon, i_1 \ ; \ i_2\rangle \mapsto_{bs} \langle \epsilon'', i\rangle} \quad (seq)$$

FIG. 3.4 – Sémantique opérationnelle à grand pas pour PIL

Expression. Chaque expression de $\langle expr\rangle$ se réduit lors de son évaluation en un élément $b \in \mathbb{B}$, autrement dit en `false` ou `true`. Il faut noter aussi que `or` et `and` sont des opérateurs associatifs-commutatifs. Cette propriété sera importante par la suite car on considérera un système de normalisation modulo cette théorie.

Représentation du code hôte. Les blocs d'instructions de code hôte n'étant pas concernés par l'optimisation, on considère une abstraction et ils sont représentés par le constructeur `hostcode`. Ce dernier est paramétré par la liste des variables PIL apparaissant dans ce bloc d'instructions. Une variable apparaît autant de fois qu'elle est utilisée. Il n'y a cependant pas d'effet de bord sur ces variables car le code hôte ne peut pas les modifier.

Instruction de séquence. L'instruction de séquence ";" est un opérateur associatif.

Instruction `nop`. Elle correspond à l'instruction vide et ne modifie donc pas l'environnement du programme.

Instruction `if`. Les conditions des blocs `if` portent sur les $e \in \langle expr\rangle$. On peut tester l'égalité de $t_1, t_2 \in \langle term\rangle$ avec l'expression `eq` ou encore si un $t \in \langle term\rangle$ a f comme symbole de tête avec $\texttt{is\_fsym}(t, f)$.

3.2 Définition des transformations de programmes

Les transformations de programmes sont définies par des règles de réécriture. Le concept consiste à remplacer, sous certaines conditions, les sous-termes représentant des instructions par des sous-termes représentant des instructions équivalentes mais plus efficaces. En reprenant l'exemple de la Figure 3.2, l'optimiseur devra pouvoir factoriser les tests et les affectations mais aussi minimiser les évaluations d'expressions.

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

FIG. 3.5 – Code PIL optimisé

Pour optimiser les programmes PIL, les règles de transformations doivent permettre de réaliser des optimisations classiques (comme l'inlining) mais aussi de déplacer, de fusionner et d'entrelacer des blocs. Il nous faut réfléchir aux conditions d'application de ces règles. Par exemple, l'application de l'inlining nécessite de connaître le nombre d'*utilisations* des variables. La fusion de blocs conditionnels requiert que les deux expressions conditionnelles soient *équivalentes* et au contraire, la permutation nécessite

l'*incompatibilité* des expressions pour s'assurer que si un bloc est exécuté, l'autre ne peut pas l'être. Ces notions correspondent à des prédicats qui vont être nécessaires pour la définition des conditions des règles. Dans la suite, nous introduisons ces prédicats avant de définir les règles de transformations.

3.2.1 Définition des prédicats

Le premier prédicat dont nous avons besoin est celui concernant l'utilisation des variables. On le définit par $\texttt{use}(v, i) = n$ où n est un entier, v une variable, i une instruction et `use` une fonction qui calcule le nombre d'utilisations de v dans i.

Définition 2. *On définit récursivement la fonction* `use` *qui calcule un majorant du nombre d'utilisations d'une variable v dans un terme t.* $\texttt{use}(v, t)$ *est défini par :*

$$
\begin{array}{l}
\texttt{use}(v,t) = \textit{match } t \textit{ with} \\
\quad | \;\; t_1 \rightarrow 0 \;\texttt{if}\; t_1 \in \mathcal{T}(\mathcal{F}) \\
\quad | \;\; v_1 \rightarrow 1 \; \textit{if } v_1 = v \\
\qquad\qquad 0 \; \textit{else} \\
\quad | \;\; \texttt{subterm}_f(t_1, \_) \rightarrow \texttt{use}(v, t_1) \\
\quad | \;\; b \in \mathbb{B} \rightarrow 0 \\
\quad | \;\; \texttt{eq}(t_1, t_2) \rightarrow \texttt{use}(v, t_1) + \texttt{use}(v, t_2) \\
\quad | \;\; \texttt{is\_fsym}(t, \_) \rightarrow \texttt{use}(v, t) \\
\quad | \;\; \texttt{or}(e_1, e_2) \rightarrow \texttt{use}(v, e_1) + \texttt{use}(v, e_2) \\
\quad | \;\; \texttt{and}(e_1, e_2) \rightarrow \texttt{use}(v, e_1) + \texttt{use}(v, e_2) \\
\quad | \;\; \neg(e_1) \rightarrow \texttt{use}(v, e_1) \\
\quad | \;\; \texttt{let}(\_, t, i) \rightarrow \texttt{use}(v, t) + \texttt{use}(v, i) \\
\quad | \;\; \texttt{if}(e, i_1, i_2) \rightarrow \texttt{use}(v, e) + max(\texttt{use}(v, i_1), \texttt{use}(v, i_2)) \\
\quad | \;\; i_1 \; ; \; i_2 \rightarrow \texttt{use}(v, i_1) + \texttt{use}(v, i_2) \\
\quad | \;\; \texttt{hostcode}(\texttt{list}) \rightarrow count(\texttt{list}, v) \\
\quad | \;\; \texttt{nop} \rightarrow 0
\end{array}
$$

où $count(l, v)$ compte le nombre d'apparitions de v dans la liste l.

Afin de pouvoir identifier des tests ou des affectations susceptibles d'être fusionnés, la notion d'équivalence entre deux termes est nécessaire.

Définition 3. *Soit un programme* `PIL` *π donné, deux sous-termes $t_1, t_2 \in \langle term \rangle \cup \langle expr \rangle$ sont équivalents ssi leur évaluation est identique dans π. On le note $t_1 \sim t_2$.*

Comme l'évaluation est sous-spécifiée, on ne peut pas déterminer en général si deux termes sont équivalents. On ne connaît pas par exemple la valeur du sujet t que l'on souhaite filtrer au niveau du programme Tom (en général, c'est un paramètre du programme hôte). Si une expression dépend du sujet, on ne peut pas l'évaluer statiquement et on ne peut pas déduire d'équivalence entre elle et une autre expression. Pour obtenir une notion décidable d'équivalence, on peut en donner une définition syntaxique.

Proposition 1. *Dans un programme* `PIL` *π, deux sous-termes $t_1, t_2 \in \langle term \rangle \cup \langle expr \rangle$ appartenant au même bloc et identiques syntaxiquement (noté $\equiv$) sont équivalents.*

$$t_1 \equiv t_2 \Rightarrow t_1 \sim t_2$$

Preuve : grâce aux restrictions du langage, on peut prouver cette proposition. En effet, les variables ne pouvant pas être modifiées et puisque t_1 et t_2 appartiennent au même bloc, leurs environnements sont égaux. De plus, comme ils sont syntaxiquement identiques, ils ont même évaluation.

Définition 4. *Un système de règles de réécriture préserve la sémantique d'un langage si chaque règle $t_i \rightarrow t'_i$ de R vérifie $t_i \sim t'_i$.*

`and`$(expr, \texttt{true})$	$\rightarrow$	$expr$	
`and`$(expr, \texttt{false})$	$\rightarrow$	`false`	
`or`$(expr, \texttt{true})$	$\rightarrow$	`true`	
`or`$(expr, \texttt{false})$	$\rightarrow$	$expr$	
`and`(`is_fsym`$(term, \textbf{symbol1})$, `is_fsym`$(term, \textbf{symbol2})$)	$\rightarrow$	`false`	if **symbol1** $\neq$ **symbol2**
`eq`$(\mathbf{t}, \mathbf{t})$	$\rightarrow$	`true`	

FIG. 3.6 – Système de normalisation d'expressions

On définit dans la Figure 3.2.1 un système de règles R permettant de réduire une expression $e \in \langle expr \rangle$ en sa forme normale $e_{\downarrow R}$. Ce système n'est pas complet au sens où il ne permet pas d'évaluer une expression e en `true` ou `false` mais simplement de simplifier statiquement e. Contrairement à l'évaluation, aucune connaissance de l'environnement du programme n'est nécessaire.

Propriété 1. *Le système R préserve la sémantique du langage* `PIL`

Preuve : en se basant sur la définition de la sémantique, on peut prouver que chacune des règles la préserve. La cinquième règle préserve la sémantique car un terme ne peut pas avoir deux symboles de tête différents.

Proposition 2. *Soit un système R préservant la sémantique de* `PIL`*, deux termes* $t_1, t_2 \in \langle expr \rangle \cup \langle term \rangle$ *sont équivalents ssi* $t_{1\downarrow R}$ *et* $t_{2\downarrow R}$ *sont équivalents.*

$$t_1 \sim t_2 \Leftrightarrow t_{1\downarrow R} \sim t_{2\downarrow R}$$

Preuve :

$\rightarrow$: comme le système R préserve la sémantique, on a $t_1 \sim t_{1\downarrow R}$ et $t_2 \sim t_{2\downarrow R}$ et comme $t_1 \sim t_2$, on peut en déduire par transitivité que $t_{1\downarrow R} \sim t_{2\downarrow R}$.

$\leftarrow$: de la même manière, comme R préserve la sémantique, on a $t_1 \sim t_{1\downarrow R}$ et $t_2 \sim t_{2\downarrow R}$. De plus, sachant que $t_{1\downarrow R} \sim t_{2\downarrow R}$, par transitivité, $t_1 \sim t_2$.

Corollaire 1. *Soient deux sous-termes* $t_1, t_2 \in \langle term \rangle \cup \langle expr \rangle$ *d'un programme* `PIL` *et un système de règle R préservant la sémantique de* `PIL`*,* $t_{1\downarrow R} \equiv t_{2\downarrow R} \Rightarrow t_1 \sim t_2$*.*

Preuve : c'est une conséquence directe des propositions 1 et 2.

En pratique, c'est ce corollaire qui sera utilisé pour déterminer si deux expressions sont équivalentes.

Orthogonalité d'expressions

Définition 5. *Deux expressions booléennes* $e_1, e_2 \in \langle expr \rangle$ *sont incompatibles si à chacune de leur évaluation, leurs valeurs ne sont pas vraies en même temps. On le note* $e_1 \perp e_2$*.*

On ne peut pas déterminer si deux conditions sont incompatibles, dans le cas général, à cause de l'évaluation. En effet, c'est le même problème que pour l'équivalence. Le sujet est souvent un paramètre du programme et correspond ainsi à une variable libre. On ne connaît donc pas statiquement sa valeur. Toute expression dépendant du sujet ne peut pas être évaluée statiquement. Pour pouvoir tout de même décider dans certains cas, on utilise le système de règles R pour normaliser les expressions.

Proposition 3. *Soient R le système de règles défini dans la partie 3.2.1 et* $e_1, e_2 \in \langle expr \rangle$*,*

$$\texttt{and}(e_1, e_2)_{\downarrow R} = \texttt{false} \Rightarrow e_1 \perp e_2$$

Preuve : de part les restrictions imposées par le langage, les variables utilisées dans e_1 et e_2 ne peuvent pas être modifiées. La valeur des expressions est donc toujours identique à chaque évaluation.

En pratique, lorsque la condition d'une règle est de la forme $c_1 \perp c_2$, on utilise la proposition 3 et on évalue si $\texttt{and}(e_1, e_2)_{\downarrow R} = \texttt{false}$.

3.2.2 Optimisations des instructions d'affectation

Excepté la fusion, les optimisations suivantes sont classiques. Leur but est de minimiser le nombre d'affectations sans augmenter le nombre d'évaluations.

Élimination de variable morte

Le compilateur de Tom est construit de manière à rester le plus générique possible par rapport aux différents types de patterns qu'il peut avoir à compiler. Beaucoup de variables sont générées par Tom et dans certains cas, elles peuvent ne pas être utilisées. Une première optimisation consiste à supprimer ces variables.

Lorsqu'une variable n'est pas utilisée, on considère que l'instruction `let` la définissant correspond à la création d'une variable morte (variable qui ne sera jamais utilisée) et elle est supprimée. En effet, la définition du langage entraîne l'absence d'effet de bord et permet de définir simplement l'utilisation d'une variable (Définition 2).

Puisque la portée de la variable est limitée au bloc *let*, l'utilisation de cette variable se limite donc aux instructions *instr* du bloc *let*. Nous pouvons donc transformer un programme PIL de la manière suivante :

$$\texttt{ElimVarMorte}: \texttt{let}(var, term, instr) \rightarrow instr \ \texttt{IF use}(var, instr) = 0$$

Propagation de constante

Dans le cas où une variable est définie par une constante, on peut la remplacer directement par sa valeur. En effet, l'évaluation d'une constante ne nécessite pas de calcul, on peut donc remplacer partout la variable par sa valeur constante et éliminer ainsi l'instruction `let`.

$$\texttt{PropConstant}: \texttt{let}(var, term, instr) \rightarrow instr[var/term] \ \texttt{IF}\ term \in \mathcal{T}(\mathcal{F}_0)$$

Inlining

Lorsqu'une variable n'est utilisée qu'une seule fois, on peut la remplacer par le terme qui la définit. Comme précédemment, la restriction du langage assure la correction de cette transformation. En effet, les variables ne peuvent pas être modifiées. Ainsi, entre la définition de la variable et son utilisation, la valeur du terme la définissant reste inchangée.

$$\texttt{Inlining}: \texttt{let}(var, term, instr) \rightarrow instr[var/term] \ \texttt{IF use}(var, instr) = 1$$

Fusion

Pour pouvoir fusionner deux blocs `let` contigus, il faut tout d'abord que les deux termes définissant chacune des deux variables soient équivalents. Dans notre cas, on se limite à une équivalence syntaxique donc ces deux termes doivent être strictement identiques. Comme notre langage interdit toute modification des variables, l'équivalence syntaxique assure l'équivalence sémantique (Théorème 1).

$$\begin{aligned}\texttt{FusionLet}: \texttt{let}(var_1, term_1, instr_1); \texttt{let}(var_2, term_2, instr_2) \rightarrow \\ \texttt{let}(var_1, term_1, instr_1; instr_2[var_2/var_1]) \ \texttt{IF}\ term_1 \sim term_2\end{aligned}$$

On suppose ici que $\texttt{use}(var_1, instr_2) = 0$ car sinon il faudrait remplacer var_1 par une nouvelle variable "fraîche" dans $instr_2$.

3.2.3 Optimisation des instructions conditionnelles

Les optimisations relatives aux instructions conditionnelles sont plus spécifiques aux programmes de filtrage. Leur but est de minimiser le nombre de tests à l'exécution sans augmenter la taille du code.

Fusion

La fusion de deux instructions conditionnelles contiguës permet de factoriser les tests des deux blocs. Elle nécessite que les conditions soient équivalentes. Puisque nous nous intéressons pour l'instant à des critères syntaxiques, on utilisera en pratique la définition d'équivalence à base de formes normales (Corollaire 1).

$$\texttt{FusionIf}: \texttt{if}(c_1, sucInstr_1, failInstr_1); \texttt{if}(c_2, sucInstr_2, failInstr_2) \rightarrow \texttt{if}(c_1, sucInstr_1; sucInstr_2, failInstr_1; failInstr_2) \texttt{ IF } c_1 \sim c_2$$

Entrelacement

On souhaite optimiser le code de deux instructions `if` dont les conditions sont incompatibles. On utilisera en pratique la proposition 3 pour déterminer l'orthogonalité. Certaines parties du code ne peuvent donc pas s'exécuter ensemble et il est possible de les réordonnancer statiquement, sans que cela ne modifie le comportement observable du programme.

On ne veut conserver qu'une seule des deux instructions `if`. A l'aide d'une étude de cas, on détermine quelles instructions doivent être exécutées en cas de succès ou d'échec de la condition.

On obtient les deux règles de transformation suivantes :

$$\texttt{if}(c_1, sucInstr_1, failInstr_1); \texttt{if}(c_2, sucInstr_2, failInstr_2) \rightarrow \texttt{if}(c_1, sucInstr_1; failInstr_2, failInstr_1; \texttt{if}(c_2, sucInstr_2, failInstr_2)) \texttt{ IF } c_1 \perp c_2$$

$$\texttt{if}(c_1, sucInstr_1, failInstr_1); \texttt{if}(c_2, sucInstr_2, failInstr_2) \rightarrow \texttt{if}(c_2, failInstr_1; sucInstr_2, \texttt{if}(c_1, sucInstr_1, failInstr_1), failInstr_2) \texttt{ IF } c_1 \perp c_2$$

Dans ces règles, une partie du code est dupliquée et pour conserver la linéarité sur la taille du code, on se l'interdit. Pour tout de même utiliser ces règles, on les instancie avec respectivement $failInstr_2$ et $failInstr_1$ à `nop` et nous pouvons éliminer certains `nop` avec la règle suivante : $\texttt{nop}; instr \rightarrow instr$. On évite ainsi la duplication de code.

$$\texttt{EntrelaceIf}: \texttt{if}(c_1, sucInstr_1, failInstr_1); \texttt{if}(c_2, sucInstr_2, \texttt{nop}) \rightarrow \texttt{if}(c_1, sucInstr_1; \texttt{nop}, failInstr_1; \texttt{if}(c_2, sucInstr_2, \texttt{nop})) \texttt{ IF } c_1 \perp c_2$$

$$\texttt{if}(c_1, sucInstr_1, \texttt{nop}); \texttt{if}(c_2, sucInstr_2, failInstr_2) \rightarrow \texttt{if}(c_2, \texttt{nop}; sucInstr_2, \texttt{if}(c_1, sucInstr_1, \texttt{nop}); failInstr_2) \texttt{ IF } c_1 \perp c_2$$

Ces deux règles permettent de réduire le nombre de tests à l'exécution en déplaçant un test dans la branche "else". Leur effet est équivalent, nous garderons donc uniquement la première pour notre système.

La deuxième règle peut être instanciée et utilisée pour la permutation de blocs. En effet, dans le cas où les instructions $failInstr_1$ et $failInstr_2$ sont réduites à l'instruction `nop`, on peut simplifier la deuxième règle en :

$$\texttt{if}(c_1, sucInstr_1, \texttt{nop}); \texttt{if}(c2, sucInstr_2, \texttt{nop}) \rightarrow \texttt{if}(c2, sucInstr_2, \texttt{if}(c_1, sucInstr_1, \texttt{nop})) \texttt{ IF } c_1 \perp c2$$

Étant donné que les deux conditions sont incompatibles, on a l'équivalence suivante :

$$\texttt{if}(c2, sucInstr_2, \texttt{if}(c_1, sucInstr_1, \texttt{nop})) \equiv \texttt{if}(c2, sucInstr_2, \texttt{nop}); \texttt{if}(c_1, sucInstr_1, \texttt{nop}) \texttt{ IF } c_1 \perp c2$$

On obtient finalement la règle de transformation de programme suivante qui correspond à la permutation de deux blocs conditionnels contigus :

$$\texttt{PermutIf}: \texttt{if}(c_1, sucInstr_1, \texttt{nop}); \texttt{if}(c2, sucInstr_2, \texttt{nop}) \rightarrow \texttt{if}(c2, sucInstr_2, \texttt{nop}); \texttt{if}(c_1, sucInstr_1, \texttt{nop}) \texttt{ IF } c_1 \perp c2$$

Cette dernière règle va servir à déplacer des blocs afin de rapprocher les blocs susceptibles de fusionner. Elle doit donc être utilisée pour rapprocher les tests identiques ce qui nécessite des conditions d'application supplémentaires par la stratégie.

Nous allons maintenant définir des critères de correction pour les règles de transformations afin d'assurer que l'optimisation préserve la sémantique du programme source.

3.3 Correction des règles de transformations

3.3.1 Extension de la sémantique de PIL

Pour prouver que deux programmes sont équivalents, il faut tenir compte des instructions du langage hôte, même si leur sémantique en PIL est toujours égale à celle de `nop`, on doit avoir une trace de leur exécution. Pour cela, dans la définition de la sémantique, on associe à l'environnement un liste δ des instructions hôtes exécutées. La seule règle de la sémantique pour laquelle δ est modifiée est celle de `hostcode`.

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

$$\frac{}{\langle \epsilon, \delta, \texttt{hostcode(list)}\rangle \mapsto_{bs} \langle \epsilon, \delta :: \texttt{hostcode(list)}, \texttt{nop}\rangle} \quad (hostcode)$$

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

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

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

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

FIG. 3.7 – Sémantique opérationnelle à grand pas pour PIL

A partir de cette sémantique, on peut prouver que tout programme PIL bien formé se réduit en `nop` et que cette dérivation est unique. La preuve se fait très naturellement par induction sur la structure du programme.

3.3.2 Définition de la correction

Definition 1. $\pi_1, \pi_2 \in$ PIL *sont sémantiquement équivalents si et seulement si :*

$$\forall \epsilon, \epsilon', \delta, \delta' \ \langle \epsilon, \delta, \pi_1\rangle \mapsto_{bs} \langle \epsilon', \delta', \texttt{nop}\rangle ssi \langle \epsilon, \delta, \pi_2\rangle \mapsto_{bs} \langle \epsilon', \delta', \texttt{nop}\rangle$$

Comme la dérivation est unique, ϵ' et δ' sont en fait fonction de π_1, ϵ et δ.

Definition 2. *Une règle de transformation* $i_1 \rightarrow i_2$ IF c *préserve la sémantique de* PIL *si et seulement si :*

$$c \Rightarrow i_1 \sim i_2$$

La relation de réduction étant définie par récurence, on peut raisonner par induction sur la structure du programme. Chacune des règles de transformations peut être prouvée de cette manière. Nous allons seulement détailler la preuve de la règle de permutation.

3.3.3 Preuve pour la permutation

La règle de transformation pour la permutation est la suivante :

$$\texttt{if}(c_1, s_1, \texttt{nop}); \texttt{if}(c_2, s_2, \texttt{nop}) \rightarrow \\ \texttt{if}(c_2, s_2, \texttt{nop}); \texttt{if}(c_1, s_1, \texttt{nop}) \; if \, c_1 \perp c_2$$

On note i_1 le premier bloc conditionnel $\texttt{if}(c_1, s_1, \texttt{nop})$ et i_2 $\texttt{if}(c_2, s_2, \texttt{nop})$.

Etant donné les spécificités du langage PIL et la condition de la règle, nous disposons des deux hypothèses suivantes :

- hyp1 : les variables contenues dans c_2 ne sont pas modifiées par i_1 car aucune variable ne peut être modifiée : pour tout ϵ, $\lceil \epsilon c_2 \rceil = \lceil \epsilon' c_2 \rceil$ avec $\langle \epsilon, \delta, i_1 \rangle \rightarrow \langle \epsilon', \delta', \texttt{nop} \rangle$,
- hyp2 : c_1 et c_2 sont incompatibles : $\texttt{and}(c_1, c_2) = \texttt{false}$

On souhaite prouver que $i_1; i_2 \sim i_2; i_1$, autrement dit :

$$\forall \epsilon, \epsilon', \delta, \delta' \; \langle \epsilon, \delta, i_1; i_2 \rangle \mapsto_{bs} \langle \epsilon', \delta', \texttt{nop} \rangle ssi \langle \epsilon, \delta, i_2; i_1 \rangle \mapsto_{bs} \langle \epsilon', \delta', \texttt{nop} \rangle$$

Soit ϵ, δ, nous allons faire un raisonnement par cas suivant l'évaluation de c_1 et c_2 :

- cas1 : $\lceil \epsilon(c_1) \rceil = \texttt{true}$ et $\lceil \epsilon(c_2) \rceil = \texttt{false}$
- cas2 : $\lceil \epsilon(c_1) \rceil = \texttt{false}$ et $\lceil \epsilon(c_2) \rceil = \texttt{true}$
- cas3 : $\lceil \epsilon(c_1) \rceil = \texttt{false}$ et $\lceil \epsilon(c_2) \rceil = \texttt{false}$

Comme les deux conditions sont incompatibles, nous n'avons pas besoin de traiter le cas $\lceil \epsilon(c_1) \rceil = \texttt{true}$ et $\lceil \epsilon(c_2) \rceil = \texttt{true}$ car alors c_1 et c_2 sont compatibles.

Cas1. On veut prouver que pour δ et ϵ, on obtient la même dérivation de $i_1; i_2$ et $i_2; i_1$.

On calcule la dérivation de $i_1; i_2$:

$$\dfrac{\dfrac{\dfrac{}{\langle \epsilon, \delta, s_1 \rangle \mapsto_{bs} \langle \epsilon', \delta', \texttt{nop} \rangle}}{\langle \epsilon, \delta, \texttt{if}(c_1, s_1, \texttt{nop}) \rangle \mapsto_{bs} \langle \epsilon', \delta', \texttt{nop} \rangle} (iftrue) \quad \dfrac{\dfrac{}{\langle \epsilon', \delta', \texttt{nop} \rangle \mapsto_{bs} \langle \epsilon', \delta', \texttt{nop} \rangle} (nop)}{\langle \epsilon', \delta', \texttt{if}(c_2, s_2, \texttt{nop}) \rangle \mapsto_{bs} \langle \epsilon', \delta', \texttt{nop} \rangle} (iffalse)}{\langle \epsilon, \delta, \texttt{if}(c_1, s_1, \texttt{nop}); \texttt{if}(c_2, s_2, \texttt{nop}) \rangle \mapsto_{bs} \langle \epsilon', \delta', \texttt{nop} \rangle} (seq)$$

Par l'hypothèse hyp1, $\epsilon(c2) = \epsilon'(c2)$ et comme $\epsilon(c2) = \texttt{false}$ alors $\epsilon'(c2) = \texttt{false}$. C'est pourquoi on peut utiliser la règle $(iffalse)$. On a choisi ϵ' et δ' tels qu'ils correspondent à l'environnement et à la liste de la dérivation de $\texttt{if}(c_1, s_1, \texttt{nop})$. Il ne peut pas y avoir d'autres solutions car la dérivation est unique.

De la même facon, on calcule la dérivation de $i_2; i_1$:

$$\dfrac{\dfrac{\dfrac{}{\langle \epsilon, \delta, \texttt{nop} \rangle \mapsto_{bs} \langle \epsilon, \delta, \texttt{nop} \rangle} (nop)}{\langle \epsilon, \delta, \texttt{if}(c_2, s_2, \texttt{nop}) \rangle \mapsto_{bs} \langle \epsilon, \delta, \texttt{nop} \rangle} (iffalse) \quad \dfrac{\dfrac{}{\langle \epsilon, \delta, s_1 \rangle \mapsto_{bs} \langle \epsilon', \delta', \texttt{nop} \rangle}}{\langle \epsilon, \delta, \texttt{if}(c_1, s_1, \texttt{nop}) \rangle \mapsto_{bs} \langle \epsilon', \delta', \texttt{nop} \rangle} (iftrue)}{\langle \epsilon, \delta, \texttt{if}(c_2, s_2, \texttt{nop}); \texttt{if}(c_1, s_1, \texttt{nop}) \rangle \mapsto_{bs} \langle \epsilon', \delta', \texttt{nop} \rangle} (seq)$$

ϵ' et δ' sont les mêmes que dans la dérivation de $i_1; i_2$ car la dérivation de $\texttt{if}(c_1, s_1, \texttt{nop})$ est unique.□

Cas2 et cas3 La preuve se fait d'une manière très similaire à celle du cas 1.□

On a finalement prouver que la règle de permutation conserve la sémantique du langage PIL.

3.4 Système de réécriture et stratégie d'application

À partir des règles définies dans la section précédente, on obtient le système de règles de transformations suivant et noté Trans1 :

$$\{ \texttt{ElimVarMorte}, \texttt{PropConstante}, \texttt{Inlining}, \texttt{FusionLet}, \texttt{FusionIf}, \texttt{EntrelaceIf}, \texttt{PermutIf} \}$$

Le système proposé est clairement non confluent et non terminant. En effet, sans condition, la règle de permutation peut s'appliquer indéfiniment car l'incompatibilité est symétrique. Si deux blocs sont incompatibles, on peut les permuter une fois et comme ils sont toujours incompatibles, la règle de permutation peut encore s'appliquer. On ne souhaite pas assurer la confluence car quel que soit l'ordre dans lequel sont appliquées les règles, le programme obtenu est sémantiquement équivalent au programme source. Par contre, il est impératif de définir une stratégie qui d'une part permettra de réaliser efficacement les optimisations et d'autre part assure la terminaison.

3.4.1 Ordre d'application des règles d'optimisation

Un premier problème concerne l'ordre d'application des règles. Considérons par exemple le programme Tom suivant :

```
%match(Term t) {
      f(x,b) ⇒ {println(hostcode1});}
      g(b)   ⇒ {println(hostcode2);}
      f(a,b) ⇒ {println(hostcode3);}
}
```

On remarque que si l'on entrelace les deux dernières règles, il n'est plus possible ensuite de permuter afin de pouvoir fusionner les 2 tests $is\_fsym(t, f)$. Cet ordre d'application n'est pas intéressant.

```
if(is_fsym(t,f),let(t1,subterm_f(t,1),let(t2,subterm_f(t,2),let(x,t1,
      if(is_fsym(t2,b),hostcode(),nop)))),
      nop);
if(is_fsym(t,g),let(t1,subterm_g(t,1),
      if(is_fsym(t1,b),hostcode(),nop)),
      if(is_fsym(t,f),let(t1,subterm_f(t,1),let(t2,subterm_f(t,2),
            if(is_fsym(t1,a),
                  if(is_fsym(t2,b),hostcode(),nop)
                  nop))),
            nop))
```

Afin de pouvoir favoriser au mieux les fusions, la règle d'entrelacement ne doit être appliquée qu'après toutes les fusions.

3.4.2 Condition d'application de la permutation

La règle de permutation rend le système non terminant puisqu'elle peut être appliquée indéfiniment. Il faut donc une condition qui limite son application aux situations intéressantes, c'est-à-dire lorsque cette permutation peut permettre une future fusion. Pour détecter les situations de fusion, on peut définir dans la règle un contexte qui indique s'il existe dans la séquence un bloc pouvant être fusionné avec celui qu'on permute. Comme on ne fait qu'une seule passe d'application des règles, cette condition suffit à éviter tout blocage par les permutations.

Cette condition assure la terminaison du système. En effet, le nombre de fusions de blocs dans un programme est fini et comme une permutation n'est réalisée qu'en prévision d'une future fusion, le nombre de permutations est fini.

3.4.3 Normalisation de l'arbre syntaxique abstrait

Pour optimiser notre programme, plusieurs parcours de l'arbre syntaxique abstrait sont nécessaires. Il faut donc définir un point fixe sur l'application *leftmost-innermost* (à gauche et en profondeur d'abord) du système de règles.

On obtient une mise sous forme normale du programme PIL par le système de règles `Trans1`. Cette forme irréductible n'est pas unique car notre système de règles n'est pas confluent. Cependant, les différents programmes obtenus sont sémantiquement équivalents.

Finalement, la stratégie de `Trans1` peut être définie de la manière suivante :

Innermost(*repeat*(`PropConstante` | `ElimVarMorte` | `Inlining` | `FusionLet` | `FusionIf` | `PermutIf`) ;
repeat(`EntrelaceIf`))

où les opérateurs de stratégie sont définis ainsi :
– $s1 \mid s2$ signifie qu'on applique $s1$ ou $s2$ indifféremment,
– $repeat(s)$ signifie qu'on applique s tant qu'on peut,
– $r1$; $r2$ signifie qu'on applique $s1$ puis $s2$ si $s1$ n'a pas échoué.

L'application d'une règle r peut être vue comme une stratégie atomique et r peut ainsi être utilisée avec les opérateurs définis précédemment.

Chapitre 4

Extension du langage et nouvelles optimisations

Jusqu'à maintenant, nous nous sommes uniquement intéressés à l'optimisation de programmes correspondant à la compilation du filtrage syntaxique. Afin d'étendre cette méthode à des programmes correspondant à la compilation du filtrage équationnel, de nouvelles instructions doivent être ajoutées au langage intermédiaire, comme par exemple les boucles `while` ou encore les `letRef`. Ces instructions compliquent les optimisations car l'équivalence syntaxique utilisée jusqu'à présent n'est plus suffisante pour assurer une équivalence sémantique. En effet, maintenant, la valeur des variables définies par `letRef` peut être modifiée (instruction `letAssign`). La forme des règles de transformations de programmes jusqu'à présent ne faisait appel qu'à l'arbre syntaxique abstrait or pour mieux représenter la dynamique des programmes `PIL`, nous devons maintenant tenir compte dans les conditions des règles du graphe de flot de contrôle. Cette nouvelle forme de règle nécessite de définir formellement le lien entre l'arbre syntaxique abstrait et le graphe de flot de contrôle.

4.1 Extension du langage intermédiaire

4.1.1 Nouvelle syntaxe

Afin de se rapprocher du langage intermédiaire manipulé par Tom, on enrichit le langage `PIL` de nouvelles instructions et d'une signature pour les listes. Cette extension du langage intermédiaire permet d'écrire des programmes effectuant du filtrage équationnel.

Nouvelles instructions. Le filtrage équationnel correspond à du filtrage sur des listes. Le sujet et les membres des règles peuvent être composés de listes. Par exemple, si l'on veut filtrer les listes contenant un élément a, le membre droit de la règle sera $(\_^*, a, \_^*)$, autrement dit a peut être à n'importe quelle position dans la liste. Les programmes `PIL` correspondant sont alors composés de boucles `while` qui permettent de parcourir des listes. La manipulation de listes nécessite de pouvoir modifier la valeur de variables définissant des listes. En effet, le parcours d'une liste consiste à récupérer la queue d'une liste à chaque itération dans la boucle, jusqu'à ce que la liste soit vide. On a donc besoin de variables modifiables, contrairement aux variables définies par l'instruction `let`. On introduit une nouvelle instruction `letRef` qui permet de définir des variables modifiables et une instruction `letAssign` pour modifier une variable définie par une instruction `letRef`.

$$
\begin{array}{rcl}
\langle instr \rangle & ::= & \texttt{letRef}(\textbf{variable}, \langle term \rangle, \langle instr \rangle) \\
& | & \texttt{letAssign}(\textbf{variable}, \langle term \rangle, \langle instr \rangle) \\
& | & \texttt{while}(\langle expr \rangle, \langle instr \rangle)
\end{array}
$$

Manipulation de listes. Le filtrage équationnel nécessite de définir une nouvelle signature pour manipuler des listes de termes, $\mathbb{L}$. Nous avons aussi besoin de récupérer la queue et le premier élément d'une liste (respectivement `getTail` et `getHead`) et de tester si une liste est vide (`isEmpty`).

$$\begin{array}{lcl} \langle list \rangle & ::= & l \in \mathbb{L} \\ & | & \texttt{getTail}(\langle list \rangle) \\ \langle term \rangle & ::= & \texttt{getHead}(\langle list \rangle) \\ \langle expr \rangle & ::= & \texttt{isEmpty}(\langle list \rangle) \end{array}$$

Cette extension du langage intermédiaire a plusieurs conséquences. L'arbre syntaxique abstrait ne représente plus aussi bien l'exécution du programme. Pour l'instruction `letAssign`, la portée de la nouvelle valeur ne correspond pas au bloc du `letAssign`. Sa portée va jusqu'au prochain assignement ou à la fin du bloc `letRef`. Prenons par exemple, le code `PIL` suivant :

$$\texttt{letRef}(x, a, \texttt{hostcode(x)}; \texttt{letAssign}(x, b, \texttt{nop}); \texttt{hostcode(x)}; \texttt{letAssign}(x, a, \texttt{nop}); \texttt{hostcode(x)})$$

La variable x utilisée dans les blocs `hostcode` prend successivement les valeurs a, b puis a.

La modification possible des variables à l'intérieur de leur bloc de définition complique aussi les conditions des règles de transformations de programmes. Par exemple, à cause des boucles, le nombre d'utilisations d'une variable ne correspond plus à son nombre d'apparitions dans le programme. Il est toujours possible d'exprimer nos conditions sur l'arbre syntaxique abstrait mais cela nécessite des définitions complexes de prédicats pour compenser l'aspect statique de l'arbre syntaxique abstrait. Pour conserver des règles simples et lisibles, il est préférable de définir les conditions sur une structure plus dynamique du programme, le graphe de flot de contrôle. De plus, pour définir des conditions les plus explicites possibles, l'utilisation de la logique temporelle permet de manipuler des prédicats simples (pas de définition récursive sur le graphe comme pour `use`). Ce sont les opérateurs temporels qui en quelque sorte contrôlent le parcours du graphe.

4.1.2 Extension de la correction de programme

Pour déterminer si un programme `PIL` est correct, dans ce langage étendu, on ajoute des nouvelles règles d'inférence pour ces instructions et on modifie la règle de la séquence pour tenir compte des possibles modifications de variables.

$$\frac{\Gamma, \Delta \vdash i_1 \quad \Gamma, \texttt{Simpl}(\Delta, i_1) \vdash i_2}{\Gamma, \Delta \vdash i_1 \,;\, i_2} \qquad \text{(seq)}$$

$$\frac{\Gamma, \Delta \vdash t \quad \Gamma; (v, ref), \Delta \vdash i}{\Gamma, \Delta \vdash \texttt{letRef}(v, t, i)} \text{ if } (v, \_) \notin \Gamma \qquad \text{(letref)}$$

$$\frac{\Gamma, \Delta \vdash t \quad \Gamma, \texttt{Red}(\Delta, v) \vdash i}{\Gamma, \Delta \vdash \texttt{letAssign}(v, t, i)} \qquad \text{(letassign)}$$

$$\frac{\Gamma, \texttt{Simpl}(\Delta, i) \vdash e \quad \Gamma, \texttt{Gen}(\Delta, e) \vdash i}{\Gamma, \Delta \vdash \texttt{while}(e, i)} \qquad \text{(while)}$$

$\texttt{Red}(\Delta, t)$ est défini par :

$$\texttt{Red}(\Delta, t) = \Delta \backslash (t, \_)$$

et $\texttt{Simpl}(\Delta, i)$ est défini par :

$$\begin{array}{l} \texttt{Simpl}(\Delta, i) = \text{match } i \text{ with} \\ \quad | \ \texttt{let}(\_, \_, i) \rightarrow \texttt{Simpl}(\Delta, i) \\ \quad | \ \texttt{letRef}(\_, \_, i) \rightarrow \texttt{Simpl}(\Delta, i) \\ \quad | \ \texttt{letAssign}(v, \_, i) \rightarrow \texttt{Simpl}(\texttt{Red}(\Delta, v), i) \\ \quad | \ \texttt{if}(e, i_1, i_2) \rightarrow \texttt{Simpl}(\texttt{Simpl}(\Delta, i_1), i_2) \end{array}$$

$$
\begin{array}{l}
\mid\ i_1\ ;\ i_2 \rightarrow \texttt{Simpl}(\texttt{Simpl}(\Delta, i_1), i_2) \\
\mid\ \texttt{while}(e, i) \rightarrow \texttt{Simpl}(\Delta, i) \\
\mid\ \texttt{nop} \rightarrow \Delta \\
\mid\ \texttt{hostcode}(\_) \rightarrow \Delta
\end{array}
$$

`Gen` ajoute au contexte Δ le couple (t, f) s'il est connu que t a comme symbole de tête f. `Red` et `Simpl` assurent qu'on enlève un couple aussitôt qu'un terme $t \in \langle term \rangle$ change.

La nouvelle règle d'inférence de la séquence permet de gérer les possibles changements de valeurs des variables. Par exemple, si la séquence se situe sous un $\texttt{is\_fsym}(v, f)$ et que la valeur de v est modifiée par un `letAssign` dans i_1, on n'est plus assuré dans i_2 que le symbole de tête de v est f. De même, pour une boucle, l'inférence de l'expression e ne doit pas se servir d'informations sur les variables qui sont modifiées dans le cœur de la boucle i car il est possible qu'elles aient changé de symbole de tête.

4.2 Nouvelle définition des transformations de programmes

Pour pouvoir tenir compte de ces nouvelles instructions, les conditions de nos règles de transformations ne porteront plus sur l'arbre syntaxique abstrait mais sur le graphe de flot de contrôle. Comme nous souhaitons utiliser des conditions temporelles sur le graphe de flot de contrôle, nous avons besoin de quelques notions sur les graphes et sur la logique temporelle CTL. Nous devons aussi définir formellement la construction du graphe de flot de contrôle à partir de l'arbre syntaxique abstrait afin d'obtenir une représentation intermédiaire fine du programme mais sans dupliquer de l'information. On peut alors définir de nouvelles règles de transformations de programmes pour les nouvelles instructions mais aussi adapter les règles du chapitre précédent car dans ce nouveau langage, elle ne sont plus valides.

4.3 Notions de graphes et de logique temporelle

Avant de définir la construction du graphe de flot de contrôle, nous introduisons quelques notions classiques sur les graphes ainsi que les opérateurs de logique temporelle dont nous aurons besoin par la suite.

4.3.1 Définitions sur les graphes

Définition 6 (Graphe orienté)**.** *Un* graphe orienté *g est un couple (V, E) où*

– *V un ensemble fini d'objets appelés sommets du graphe.*

– *$E \subseteq V \times V$ un ensemble fini de couples appelés arcs du graphe.*

Soit un arc (x,y) du graphe, les sommets x et y sont les extrémités de l'arc et x est le prédécesseur de y. Les arcs sont donc orientés.

Définition 7 (Chaîne)**.** *Une chaîne est une suite d'arcs telle que chaque arc de la suite a une extrémité en commun avec l'arc précédent. La direction n'a pas d'importance.*

Définition 8 (Connexité)**.** *Un graphe est dit* connexe *si pour toute paire de sommets distincts il existe une chaîne les reliant.*

Définition 9 (Sommet de sortie)**.** *Un sommet x est un* **sommet de sortie** *s'il n'a pas de successeur c'est-à-dire qu'il n'existe pas d'arc (x,_).*

Définition 10 (Sommet d'entrée)**.** *Un sommet x est un* **sommet d'entrée** *s'il n'a pas de prédécesseur c'est-à-dire qu'il n'existe pas d'arc (_,x).*

Définition 11. *La* **fonction** `edge` *construit un arc du graphe de flot de contrôle. Elle prend comme paramètre les deux sommets reliés par cet arc, le premier étant le prédécesseur du second.*

Définition 12. *La* **fonction** `conc` *concatène un graphe $g \in G$ avec une liste l de graphes $g_i \in G$. Autrement dit, les sommets de sortie de g deviennent les prédécesseurs des sommets d'entrée des graphes de l. Soit un graphe $g \in G$, $out(g)$ renvoie l'ensemble des sommets de sortie et $in(g)$ renvoie l'ensemble des sommets d'entrée. $\times$ correspond au produit cartésien. On obtient la définition suivante de conc*

$conc : G \times list[G] \rightarrow G$
$conc((V, E), [(V', E'), tail]) \Rightarrow conc((V \cup V', E \cup E' \cup E''), tail)$
$avec\ E'' = out((V, E)) \times in((V', E'))$

Définition 13. *La* **fonction** $\cup$ *fusionne deux graphes $g, g' \in G$ en un seul graphe.*

$\cup : G \times G \rightarrow G$
$(V, E) \cup (V', E') \Rightarrow (V \cup V', E \cup E')$

4.3.2 Définition d'opérateurs temporels CTL

Nous utilisons des formules de logique CTL (logique temporelle arborescente). Cette logique temporelle est basée sur la notion de chemins dans des arbres infinis correspondants à des graphes. Un chemin dans un arbre correspond à une suite de nœuds reliés par des arcs. Un chemin contient un nœud initial, correspondant au premier nœud du chemin.

Nous définissons ici uniquement les opérateurs temporels dont nous aurons besoin par la suite :

- $AX : n \models AX(cond)$ ssi tous les fils n_i de n vérifient $n_i \models cond$. L'opérateur A (comme All) signifie que la formule doit être vérifiée pour tous les chemins dont le nœud initial est n. L'opérateur X fait référence au deuxième nœud du chemin et finalement l'association AX signifie que tous les successeurs de n doivent vérifier la condition.
- $A(c_1\ U\ c_2) : n \models A(c_1\ U\ c_2)$ ssi $n \models c_2$ ou ($n \models c_1$ et tous les fils n_i vérifient $n_i \models A(c_1\ U\ c_2)$). L'opérateur U (comme $Until$) associé à A signifie que la condition c_1 doit être vérifiée pour tous les chemins partant de n jusqu'à ce que la condition c_2 soit vérifiée.

4.4 Définition du graphe de flot de contrôle à partir de l'arbre syntaxique abstrait

Un graphe de flot de contrôle correspond à un graphe *orienté et connexe*. On note G l'ensemble des graphes de flot de contrôle. Avant de définir la fonction de construction du graphe de flot de contrôle, nous avons besoin de la fonction `node` qui construit un nœud du graphe de flot de contrôle.

Définition 14. *La* **fonction** `node` *construit un graphe composé d'un seul sommet à partir d'un sous-terme t de l'arbre syntaxique abstrait et de sa position $p \in Pos$. Ce graphe ne possède pas d'arcs. C'est le plus petit graphe possible.*

On définit récursivement le constructeur `cfg` qui construit à partir d'un programme `PIL` π et d'une position ω le graphe de flot de contrôle correspondant.

Afin de faire le lien entre l'arbre syntaxique abstrait et le graphe de flot de contrôle, les nœuds du graphe de flot de contrôle font référence aux sous termes de l'arbre syntaxique abstrait correspondants. Pour définir le constructeur `cfg`, on enrichit le langage `PIL` par de nouvelles instructions permettant d'expliciter la portée des blocs au niveau du graphe de flot de contrôle. Ils correspondent en quelque sorte à des marqueurs et leur sémantique correspond à celle de `nop` :

- $free(x)$ qui prend en paramètre une variable x et qui délimite la fin de la portée de celle-ci,
- $endIf$ qui délimite la fin d'un bloc conditionnel,
- $beginWhile$ qui délimite le début d'une boucle,
- $endWhile$ qui délimite la fin des instructions d'une boucle,
- $failWhile$ qui délimite la sortie d'une boucle lorsque la condition a échouée

La fonction `cfg` est définie par :

$$
\begin{array}{lcll}
\texttt{cfg}(\pi,\omega) & = & \text{match } \pi \text{ with} & \\
& | & \texttt{hostcode}(\_) & \rightarrow \quad \texttt{node}(\pi,\omega) \\
& | & \texttt{nop} & \rightarrow \quad \texttt{node}(\pi,\omega) \\
& | & \texttt{let}(x,t,i) & \rightarrow \quad \texttt{conc}(\texttt{conc}(\texttt{node}(\pi,\omega),[\texttt{cfg}(i,\omega.3)]),[\texttt{node}(free(x),\omega)]) \\
& | & \texttt{letRef}(x,t,i) & \rightarrow \quad \texttt{conc}(\texttt{conc}(\texttt{node}(\pi,\omega),[\texttt{cfg}(i,\omega.3)]),[\texttt{node}(free(x),\omega)]) \\
& | & \texttt{letAssign}(x,t,i) & \rightarrow \quad \texttt{conc}(\texttt{node}(\pi,\omega),[\texttt{cfg}(i,\omega.3)]) \\
& | & \texttt{if}(e,i_1,i_2) & \rightarrow \quad \texttt{conc}(\texttt{conc}(\texttt{node}(\pi,\omega),[\texttt{cfg}(i_1,\omega.2),\texttt{cfg}(i_2,\omega.3))]),[\texttt{node}(endIf,\omega)]) \\
& | & i_1 \,;\, i_2 & \rightarrow \quad \texttt{conc}(\texttt{cfg}(i_1,\omega.1),[\texttt{cfg}(i_2,\omega.2)]) \\
& | & \texttt{while}(e,i) & \rightarrow \quad \texttt{conc}(nodebeginWhile\omega,\texttt{conc}(\texttt{node}(\pi,\omega), \\
& & & \quad [\texttt{conc}(\texttt{cfg}(i,\omega.2),\texttt{node}(endWhile,\omega)),\texttt{node}(failWhile,\omega)])) \\
& & & \cup \quad (\{\texttt{node}(endWhile,\omega),\texttt{node}(\pi,\omega)\}, \\
& & & \quad \{\texttt{edge}(\texttt{node}(endWhile,\omega),\texttt{node}(\pi,\omega))\})
\end{array}
$$

On définit dans la figure 4.4 un exemple de graphe de flot de contrôle construit avec `cfg`. On remarque qu'aucune information n'est dupliquée car les sommets du graphe de flot de contrôle font référence à des nœuds de l'arbre syntaxique abstrait. Chaque sommet du graphe de flot de contrôle stocke aussi la position du sous-terme de l'arbre syntaxique abstrait auquel il correspond. Cette information permet de distinguer les sommets correspondant à des sous-termes identiques mais de position différente comme par exemple pour l'instruction *beginWhile* si le programme avait contenu deux boucles.

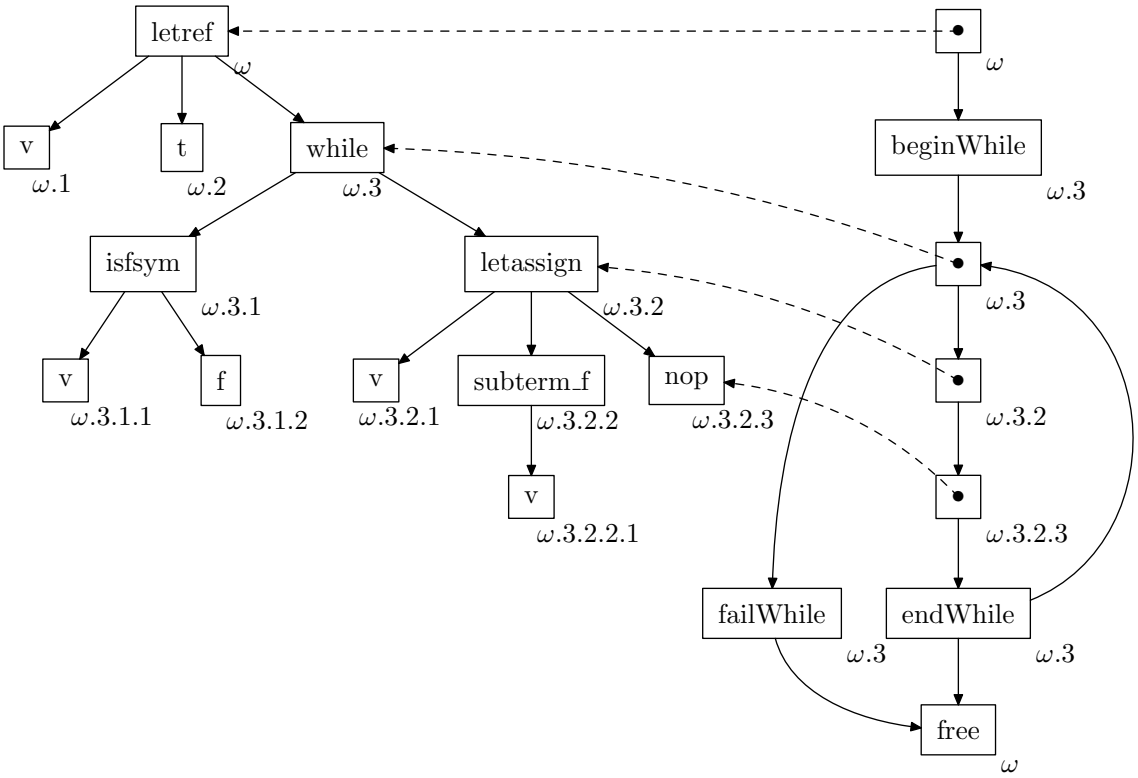

Fig. 4.1 – Exemple de graphe de flot de contrôle construit avec `cfg`

On souhaiterait pouvoir exprimer des conditions de règles sur le graphe de flot de contrôle sous forme de formules de logique temporelle. En effet, comme l'a présenté D. Lacey dans sa thèse [Lac03], la logique temporelle permet d'exprimer facilement des informations dynamiques sur le graphe de flot de contrôle. L'utilisation couplée du graphe de flot de contrôle et de la logique temporelle permet ainsi d'éviter des structures d'analyse complexes sur l'arbre syntaxique abstrait. Par exemple, si l'on souhaite connaître

le nombre d'utilisations d'une variable à l'exécution et non plus statiquement, l'information est très compliquée à obtenir à partir de l'arbre syntaxique abstrait puisque la dynamique du programme n'est pas représentée directement.

Pour pouvoir exprimer des conditions temporelles sur le graphe de flot de contrôle, il faut pouvoir associer à n'importe quel sous-terme (i.e. un sous-terme de l'arbre syntaxique abstrait) un sommet du graphe de flot de contrôle. Or nous avons construit les sommets du graphe de flot de contrôle à partir d'un sous-terme de l'arbre syntaxique abstrait et de la position de ce sous-terme. On définit donc une fonction $\texttt{getNode} : \texttt{PIL} \times Pos \rightarrow V$ qui permet à partir d'un sous-terme et d'une position dans l'arbre syntaxique abstrait de retrouver le premier sommet de ce sous-terme au niveau du graphe de flot de contrôle.

Lorsque le sous-terme t de l'arbre syntaxique abstrait est tel que $t \in \textbf{symbol} \cup \langle term \rangle \cup \langle expr \rangle$, on détermine l'instruction $i \in \langle instr \rangle$ englobante de ce sous-terme et on renvoie $\texttt{getNode}(i, w_i)$. Les règles de transformation de programmes sont de la forme :

$$term_1 \rightarrow term_2 \ \texttt{IF getNode}(term_3, \omega) \models temporalCond$$

avec $term_1$, $term_2$ et $term_3$ des sous-termes de l'arbre syntaxique abstrait, ω une position dans l'arbre syntaxique abstrait et $temporalCond$ une condition temporelle sur $\texttt{getNode}(term_3, \omega)$ dans le graphe de flot de contrôle.

4.5 Règles d'optimisation

À partir de cette définition de règles de transformations, nous redéfinissons les optimisations du chapitre 3 ainsi que de nouvelles transformations pour les instructions `letRef` et `letAssign`.

4.5.1 Optimisation de l'instruction let

L'élimination de variable morte et la propagation de constante peuvent toujours être définies de manière statique. Parmi les optimisations précédentes du `let`, seul l'inlining et la fusion doivent être redéfinis pour tenir compte de la modification possible des variables contenues dans les termes.

Inlining

Notre précédente définition de l'inlining dans le chapitre 3 n'était valable que si le terme affecté à la variable ne changeait pas. Ce n'est plus le cas maintenant puisqu'il y a l'instruction `letAssign`. On doit donc définir une condition sur le graphe de flot de contrôle.

On définit les prédicats suivants qui seront utilisés dans les conditions temporelles :

- `isModified` prend un ensemble de variables et teste si l'une des variables est modifiée (par une instruction `letAssign`) au sommet courant de la condition temporelle à laquelle il appartient,
- `isUsed` teste si une variable est utilisée au sommet courant,
- `isNode` teste si un sommet correspond au sous-terme et à la position données en paramètre,
- `isAssigned` teste si une variable est assignée au sommet courant.

$$\texttt{ref@}(\texttt{let}(var, term, instr)) \ \rightarrow \ instr[var/term]$$
$$\texttt{IF getNode}(\texttt{ref}, \omega) \models$$
$$A(\neg\texttt{isModified}(Var(term)) \ U \ (\texttt{isUsed}(var) \ \wedge \ AX(A(\neg\texttt{isUsed}(var) \ U \ \texttt{isNode}(free(var), \omega)))))$$

Cette condition temporelle teste au niveau du graphe de flot de contrôle si à partir du début du bloc `let`, on ne modifie pas les variables de $term$ jusqu'à un sommet n. On obtient le début de formule suivant : $\texttt{getNode}(\texttt{ref}, \omega) \models A(\neg\texttt{isModified}(Var(term)) \ U \ \ldots$ Ce sommet n correspond à une utilisation de var (d'où $\texttt{isUsed}(var)$) et comme on souhaite qu'il n'y ait qu'une seule utilisation de var, on doit vérifier qu'à partir de ce sommet il n'y a plus d'utilisation jusqu'à la fin du bloc `let`. Tous les successeurs du sommet n (noté AX) doivent vérifier que $A(\neg\texttt{isUsed}(var) \ U \ \texttt{isNode}(free(var), \omega))$.

Fusion

Comme précédemment, on ne pourra fusionner deux instructions `let` que si les variables du $t \in \langle term \rangle$ ne sont pas modifiées dans le bloc du premier `let`. On assure ainsi que les deux variables sont bien définies avec la même valeur.

$$\texttt{ref@}(\texttt{let}(var_1, term_1, instr_1)); \texttt{let}(var_2, term_2, instr_2) \rightarrow \texttt{let}(var_1, term1, instr_1; instr_2[var_2/var_1])$$
$$\texttt{IF getNode}(\texttt{ref}, \omega) \models term_1 \sim term_2 \wedge A(\neg \texttt{isModified}(Var(term)) \; U \; \texttt{isNode}(free(var_1), \omega))$$

Par exemple, dans l'instruction `PIL` suivante, on ne peut pas fusionner car x est modifiée dans le premier bloc :

$$\texttt{letRef}(x, a, \texttt{let}(y_1, x, \texttt{hostcode}(\texttt{y}_1); \texttt{letAssign}(x, b, \texttt{nop})); \texttt{let}(y_2, x, \texttt{hostcode}(\texttt{y}_2)))$$

4.5.2 Optimisation de l'instruction if

Les optimisations de l'instruction `if` doivent aussi être adaptées pour tenir compte des modifications possibles de variables.

Fusion

Les variables contenues dans la condition ne doivent pas être modifiées dans le premier bloc conditionnel car sinon l'équivalence syntaxique ne suffira pas à assurer l'équivalence sémantique. La condition assure donc que le premier bloc conditionnel ne modifie pas les variables de la condition.

$$\texttt{ref@}(\texttt{if}(c_1, sucInstr_1, failInstr_1); \texttt{if}(c_2, sucInstr_2, failInstr_2)) \rightarrow$$
$$\texttt{if}(c_1, sucInstr_1; sucInstr_2, failInstr_1; failInstr_2)$$
$$\texttt{IF getNode}(\texttt{ref}, \omega) \models c_1 \sim c_2 \; \wedge \; A(\neg \texttt{isModified}(Var(c_2)) \; U \; \texttt{isNode}(endIf, \omega.1))$$

$\omega.1$ correspond à la position du premier fils (le plus à gauche) du nœud à la position ω dans l'arbre syntaxique abstrait. Comme le nœud à la position ω correspond à la séquence, son premier fils correspond au premier bloc conditionnel.

Entrelacement

La première règle d'entrelacement nécessite une condition similaire à celle de la fusion. Cette fois-ci, ce sont les variables de la deuxième condition qui ne doivent pas être modifiées par le premier bloc conditionnel afin que l'orthogonalité des conditions soit toujours vérifiée.

$$\texttt{ref@}(\texttt{if}(c_1, sucInstr_1, failInstr_1); \texttt{if}(c_2, sucInstr_2, failInstr_2)) \rightarrow$$
$$\texttt{if}(c_1, sucInstr_1; failInstr_2, failInstr_1; \texttt{if}(c_2, sucInstr_2, failinstr_2))$$
$$\texttt{IF getNode}(\texttt{ref}, \omega) \models c_1 \perp c_2 \; \wedge \; A(\neg \texttt{isModified}(Var(c_2)) \; U \; \texttt{isNode}(endIf, \omega.1))$$

Pour la deuxième règle d'entrelacement, la condition est la même d'autant plus que c'est la deuxième condition qui est évaluée dès le début.

$$\texttt{ref@}(\texttt{if}(c_1, sucInstr_1, failInstr_1); \texttt{if}(c_2, sucInstr_2, failInstr_2)) \rightarrow$$
$$\texttt{if}(c_2, failInstr_1; sucInstr_2, \texttt{if}(c_1, sucInstr_1, failinstr_1); failInstr_2)$$
$$\texttt{IF getNode}(\texttt{ref}, \omega) \models c_1 \perp c_2 \; \wedge \; A(\neg \texttt{isModified}(Var(c_2)) \; U \; \texttt{isNode}(endIf, \omega.1))$$

Permutation

Pour la règle de permutation, la condition sur les variables de c_2 doit toujours être vérifiée mais en plus les variables de c_1 ne doivent pas être modifiées par le deuxième bloc car cette fois-ci les instructions sont inversées. Elles sont donc susceptibles d'annuler l'orthogonalité des deux conditions.

$$\texttt{ref}_1\texttt{@}(\texttt{if}(c_1, sucInstr_1, \texttt{nop})); \texttt{ref}_2\texttt{@}(\texttt{if}(c_2, sucInstr_2, \texttt{nop})) \rightarrow$$
$$\texttt{if}(c_2, sucInstr_2, \texttt{nop}); \texttt{if}(c_1, sucInstr_1, \texttt{nop})$$
$$\texttt{IF getNode}(\texttt{ref}_1, \omega.1) \models c_1 \; \perp \; c_2 \; \wedge \; A(\neg \texttt{isModified}(Var(c_2)) \; U \; \texttt{isNode}(endIf, \omega.1)) \; \wedge$$
$$\texttt{getNode}(\texttt{ref}_2, \omega.2) \; \models \; A(\neg \texttt{isModified}(Var(c_1)) \; U \; \texttt{isNode}(endIf, \omega.2))$$

4.5.3 Optimisation de l'instruction letRef

Au lieu d'adapter les règles du `let` pour le `letRef`, il est préférable de le traiter au niveau des `letAssign`. Ainsi, lorsqu'il n'y a plus de `letAssign` associé à un `letRef`, on peut le transformer en `let`. L'élimination de variable morte est alors réalisée par la règle du `let`.

Cette solution a plusieurs avantages. Elle permet d'être plus fin dans l'optimisation en réalisant l'inlining et l'élimination de variable morte sur les `letAssign` et donc de traiter plus de cas. De plus, en transformant un `letRef` en `let`, elle permet de réutiliser les règles du `let` au lieu de les adapter. On minimise ainsi le nombre de règles de transformations de programmes.

En ce qui concerne la fusion de deux blocs `letRef`, la condition nécessite que la variable ne soit pas modifiée dans le premier bloc et ça n'a donc pas d'intérêt. Cette fusion n'est pas intéressante dans le cadre d'équivalence syntaxique mais dans un cadre plus sémantique, par exemple, avec la gestion d'environnement, elle pourrait être définie.

Finalement, il ne reste plus qu'une règle pour le `letRef` : le remplacement d'un `letRef` par un `let`.

Remplacement d'un letRef par un let

Si dans le bloc *instr* d'un `letRef`, il n'y a pas de modification de la variable alors on peut considérer ce `letRef` comme un `let`.

$$\texttt{letRef}(var, term, instr) \ \rightarrow \ \texttt{let}(var, term, instr)$$
$$\texttt{IF getNode}(instr, \omega) \models A(\neg \texttt{isAssigned}(var) \ U \ \texttt{isNode}(free(var), \omega))$$

Par exemple, l'instruction $\texttt{letRef}(v, t, \texttt{hostcode}())$ peut être réécrite en $\texttt{let}(v, t, \texttt{hostcode}())$. On peut alors appliquer la règle d'élimination de variable morte et ainsi obtenir `hostcode`().

4.5.4 Optimisation de l'instruction letAssign

Élimination de variable morte

On élimine les instructions `letAssign` pour lesquelles il n'y a pas d'utilisation de cette nouvelle valeur. La condition doit assurer que la variable n'est pas utilisée avant une nouvelle affectation ou la fin de bloc du `letRef`.

$$\texttt{ref@}(\texttt{letAssign}(var, term, instr)) \ \rightarrow \ instr$$
$$\texttt{IF getNode}(\texttt{ref}, \omega) \models A(\neg \texttt{isUsed}(var) \ U \ (\texttt{isAssigned}(var) \ \vee \ \texttt{isNode}(free(var), \_)))$$

Le prédicat $\texttt{isNode}(free(var), \_)$ teste si le nœud courant est un nœud dont le sous-terme est $free(var)$ et la position quelconque. Cette condition est suffisante car un programme `PIL` correct n'autorise pas de redéfinition de variable.

Propagation de constante et Inlining

La définition de la propagation de constante et l'inlining est semblable à celle pour `let` sauf que la variable doit être remplacée par sa définition sur toute sa portée, c'est-à-dire jusqu'à une prochaine affectation ou jusqu'à la fin du `letRef`. En outre, pour l'inlining, les variables composant le terme ne doivent pas être modifiées entre l'affectation et l'utilisation pour assurer l'équivalence.

Remplacement d'un letAssign par un letRef

Si entre l'instruction `letRef` et `letAssign`, la variable n'est pas utilisée, on peut déplacer l'instruction `letRef` au niveau du `letAssign`, en remplaçant `letAssign` par `letRef`.

$$\texttt{ref@}(\texttt{letRef}(var, term_1, x; refLet\texttt{@}(\texttt{letAssign}(var, term_2, instr)); y)) \ \rightarrow \ x; \texttt{letRef}(var, term_2, instr; y)$$

$$\texttt{IF getNode}(\texttt{ref}, \omega) \models A(\neg \texttt{isUsed}(var) \ U \ \texttt{isNode}(refLet, \omega_{refLet}))$$

Cette optimisation est intéressante car elle permet de calculer la valeur d'une variable le plus tard possible et donc d'éviter de réaliser des calculs coûteux et parfois inutiles.

Finalement, toutes ces règles forment un nouveau système d'optimisation. La même stratégie que dans le chapitre 3 peut être appliquée. On ajoute ces règles au premier opérateur *repeat*. Elles peuvent s'exécuter dans n'importe quel ordre comme les autres (opérateur |) tant que l'entrelacement est toujours réalisé à la fin. On remarquera qu'aucune optimisation sur les boucles n'a été définie car les boucles des programmes générés ne nécessitent aucune amélioration. Par exemple, elles ne contiennent pas d'invariant pouvant être extrait (optimisation très classique des boucles).

Dans cette partie, nous avons surtout porté notre attention sur la définition des règles et sur la formalisation de la relation entre le graphe de flot de contrôle et l'arbre syntaxique abstrait. Nous n'avons pas eu assez de temps pour prouver ces règles. En effet, les preuves sont plus techniques dans le sens où il faut définir correctement le lien entre les formules de logique temporelle et le programme `PIL`. Cependant, ces preuves peuvent être réalisées de la même manière que dans le chapitre 3.

Chapitre 5

Mise en œuvre et résultats expérimentaux

5.1 Implantation des règles d'optimisation et de la stratégie

L'implantation des règles est simplifiée par l'utilisation des primitives Tom. Elles s'implantent naturellement par un système de règles défini par l'instruction `%match`. La partie la plus intéressante de la mise en œuvre est l'implantation de la stratégie d'application.

5.1.1 Place de l'entrelacement

L'entrelacement doit être réalisé en fin d'optimisation afin de ne pas bloquer les fusions. L'implantation de cette contrainte est simplifiée par une propriété de Tom. En effet, comme Tom préserve l'ordre du filtrage par compilation, en plaçant la règle d'entrelacement à la fin du système on est assuré qu'elle sera exécutée après toute fusion.

5.1.2 Condition d'application de la permutation

La règle de permutation n'a d'intérêt que si une future fusion est possible. Il faut donc pouvoir exprimer au niveau de l'implantation cette condition d'application. La première solution consiste à utiliser un contexte. On ne définit pas simplement la séquence de deux blocs conditionnels, on se donne aussi un contexte dans lequel on filtre un bloc conditionnel susceptible de fusionner. Cependant, cette définition rend l'application de la règle quadratique en temps puisqu'elle correspond à du filtrage équationnel avec 3 listes à parcourir.

Une solution alternative plus efficace est de considérer un ordre total sur les conditions `is_fsym` portant sur le même sous-terme. On ordonne alors les blocs conditionnels suivant cet ordre, il en découle que deux conditions identiques deviennent contiguës. De plus, comme deux conditions qui n'ont pas le même ordre sont incompatibles, on est assuré que leur permutation respecte la sémantique du programme `PIL`. L'application de la règle devient alors linéaire en temps et on permute bien les blocs susceptibles de fusionner.

5.1.3 Utilisation de la bibliothèque Mutraveler

La bibliothèque Mutraveler basée sur les travaux de E. Visser [VeABT98] et de J. Visser [Vis01b] permet de définir une stratégie à partir d'opérateurs de base qui sont :

- l'opérateur de récursion : μ,
- la composition séquentielle : $seq(s_1, s_2)$ (s_2 seulement si s_1 n'échoue pas),
- le choix : $choice(s_1, s_2)$ (si s_1 échoue, s_2 sinon s_1),
- l'identité : id,

– l'échec : $fail$,
– la stratégie $All(s)$ (succès de $All(s)$ si succès de s pour tous les fils du nœud courant),
– la stratégie $One(s)$ son dual (succès de $One(s)$ si succès de s pour au moins un fils du nœud courant),
– l'application d'un système de règles R

Grâce à ces opérateurs, on peut par exemple définir la stratégie *repeat* qui consiste à répéter l'application d'une stratégie v jusqu'à ce ne soit plus possible :

$$repeat(s) = \mu x(choice(seq(v, x), id))$$

Une première stratégie de normalisation peut consister à répéter l'application s de nos règles jusqu'à obtenir l'identité et en traversant le terme en profondeur d'abord. On obtient la définition suivante :

$$innermost(s) = \mu x(seq(All(x), choice(seq(s, x), id)))$$

Ces opérateurs sont basés sur l'échec et l'implantation a été réalisée par des exceptions Java. Ce mécanisme est coûteux en temps.

Contrairement à l'exploration d'espace de recherche, dans le cas particulier de la normalisation on ne s'intéresse pas à l'échec d'une stratégie. Notre but étant de calculer des formes normales, on peut modéliser l'échec d'une règle par l'identité (une règle qui ne peut pas s'appliquer ne modifie pas le terme courant). On définit donc de nouveaux opérateurs basés sur l'identité :
– $choiceId(s_1, s_2)$ (s_2 si l'application de s_1 est équivalente à id sinon s_1)
– $oneId(s)$ qui échoue si pour tous les fils du nœud courant, l'application de s est équivalente à l'identité,
– $seqId(s_1, s_2)$ (si $s_1 \neq id$, on applique ensuite s_2)

Un exemple de stratégie utilisant $choiceId$ et $oneId$ est $onceBottomUpId$ définie par

$$onceBottomUpId(s) = \mu x(choiceId(oneId(MuVar()), v)$$

Cette stratégie part de la feuille la plus à gauche du terme et remonte en essayant d'appliquer la stratégie v. Elle s'arrête dès que la stratégie v succède. La différence avec onceBottonUp est qu'ici, une stratégie qui renvoie l'identité échoue. Il n'y a pas besoin de gérer explicitement l'échec avec des exceptions.

L'opérateur $seqId$ nous permet de redéfinir notre stratégie *innermost* en une version plus efficace pour la normalisation :

$$innermostId(s) = \mu x(seq(All(x), seqId(s, x)))$$

Ce travail a contribué à identifier une faiblesse de la bibliothèque Mutraveler. Cela nous a amené à modéliser l'échec d'une autre façon et à proposer une extension de Mutraveler qui améliore considérablement les performances lors d'un calcul de forme normale. Le temps d'optimisation d'une centaine de règles de filtrage syntaxique passe de 90 minutes à 3 minutes.

Toutes les règles définies dans le chapitre 3 ainsi que la stratégie ont été intégrées à la version officielle et diffusée de Tom. Les règles du chapitre 4 sont en cours d'implantation. En plus de la correction formelle des règles dans la section 3.3, on peut vérifier que l'implantation des règles est correcte grâce à la certification du code réalisée au sein de l'outil Tom. On ne vérifie alors pas les règles mais le code généré après optimisation. Pour plus de détails sur cette méthode de certification du filtrage, on peut lire [KMR05].

L'implantation des règles du chapitre 3 nous a permis d'évaluer le gain de performances sur des exemples caractéristiques et ainsi tester l'efficacité de l'optimiseur.

5.2 Résultats expérimentaux

Il n'existe pas de critères pour comparer des méthodes de compilation de filtrage. D'une part, comme les approches sont différentes, il est impossible de comparer un algorithme réalisant du filtrage avec

une instruction `switch/case` (comme pour Caml) et une solution basée sur l'instruction `if/then/else` (comme pour Tom). En effet, la discrimination est réalisée de manière dichotomique dans le premier cas et pas dans le second. De plus, le compromis entre linéarité en espace et linéarité en temps donne lieu à des solutions orthogonales (automates déterministes et automates avec backtracking). Le critère devrait prendre en compte ces deux dimensions pour être le plus pertinent possible. Enfin, on ne peut pas donner de manière générale la complexité d'un algorithme car elle dépend complètement du sujet, qui n'est pas connu.

Pour tester le gain de performances apporté par l'optimiseur, nous avons choisi des fonctions classiques comme Fibonacci ou le crible d'Érathostène et des programmes plus spécifiques comme le jeu du Gomoku pour le filtrage équationnel et l'automate cellulaire de Langton pour le filtrage syntaxique.

La suite de Fibonacci La suite de Fibonacci est un exemple classique d'évaluation de l'efficacité du filtrage. En effet, en Tom, la suite de fibonnacci sur des nombres de Peano s'implante par la construction `%match` suivante :

```
%match(term t) {
    zero()      -> { return suc(zero); }
    suc(zero()) -> { return suc(zero); }
    suc(suc(x)) -> { return plus(fib(‘x),fib(suc(‘x))); }
  }
```

L'optimiseur va pouvoir par exemple fusionner le test sur `suc()` des deux dernières règles et l'entrelacer avec le test sur `zero()`. Toutes ces améliorations diminuent le nombre de tests à l'exécution et ainsi le temps d'exécution est améliorée d'environ 23%.

Le crible d'Érathostène Cet algorithme permet de déterminer tous les nombres premiers inférieurs à un nombre entier n. Intuitivement, le principe est basé sur le fait que tout entier naturel composé possède au moins un diviseur différent de 1 et inférieur ou égal à sa racine carré. Dès lors, soit un entier naturel k donné, il suffit d'essayer de le diviser par les entiers naturels non nuls différents de 1 et inférieurs ou égaux à sa racine carrée. Si l'on ne trouve aucun diviseur, alors k est premier.

La réalisation du crible d'Érathostène s'écrit par une instruction de filtrage sur la liste des nombres entiers de 0 à n. Voici la construction `%match` correspondante :

```
%match (list l) {
  conc(x*,e2,y*,e1,z*) -> {
    if(‘e1 % ‘e2 == 0){return erat(‘conc(x*,e2,y*,z*));}
}

}
```

Dans cette implantation, on compare simplement `e1` avec tous les nombres qui lui sont inférieurs. Pour cet exemple, c'est l'inlining de `x`,`y` et `z` qui va nettement améliorer le temps d'exécution car ils ne seront calculés que si le nombre `e1` possède un diviseur. Pour $n = 1000$, le facteur de gain est de 30.

Le jeu du Gomoku Le jeu du gomoku est une version améliorée du morpion. Le plateau de jeu est plus grand ce qui permet des parties plus équilibrées. Le principe est d'aligner sur un plateau 5 pions de sa couleur. Pour choisir quelle case jouer, l'ordinateur donne des points aux cases correspondant à des situations favorables suivant une échelle et joue la case qui a le plus de points. Le plateau est représenté par une liste de listes. L'opération d'évaluation des cases correspond à du filtrage sur des listes. L'ordinateur doit filtrer dans chacune des listes une configuration avantageuse comme par exemple quatre pions contigus. Ce programme est donc très intéressant pour quantifier le gain de l'optimisation sur le filtrage équationnel. En effet, certains tests communs entre deux configurations vont être factorisés. Pour 100 parties jouées, le temps d'exécution est divisé par 2.

L'automate cellulaire L'exemple de l'automate cellulaire correspond à un automate d'auto-reproduction de Langton. Un ensemble de cellules se reproduisent à partir d'elles-mêmes en se basant sur leur état et sur l'état de leurs voisines. L'automate de Langton est basé sur 8 états et à un voisinage de 5 états. Une étape de réplication va consister à filtrer des configurations de voisinage, c'est-à-dire cinq états de cellules voisines (cela correspond à un quadrillage : centre,est,ouest,nord,sud) et modifier l'état de la cellule centrale. Le nombre important de règles (plus de 100) de filtrage purement syntaxique (pas de filtrage équationnel) rend cet exemple approprié pour l'évaluation des performances du compilateur. En effet, il permet de tester l'efficacité des règles de permutation, de fusion et d'entrelacement. De plus, étant donné le nombre de règles, on peut évaluer les performances de la stratégie d'application des règles. C'est en affinant la stratégie à partir de cet exemple que l'on a pu diminuer réduire le temps nécessaire pour optimiser le programme de 90 minutes à 3 minutes. Sur cet automate, le temps d'exécution pour 1000 changements d'états est 5 fois plus rapide.

	$100 * \texttt{Fibonnacci(18)}$	`Erat(1000)`	`Gomoku(100)`	`Langton(1000)`
`sans optimisation`	$11.454s$	$174.998s$	$56.452s$	$109.6s$
`avec optimisation`	$8.874s$	$5.724s$	$21.455s$	$19.8s$

Conclusion et perspectives

Le but de ce stage était de spécialiser des méthodes générales d'optimisation de programmes au cas de la programmation par règles et tout particulièrement du filtrage. La simplicité et la clarté des règles d'optimisation définies durant ce stage justifie ce choix. En effet, par rapport aux algorithmes complexes de compilation de filtrage, cette méthode est d'une part plus générique (l'application des optimisations dépassent le cas du filtrage) et d'autre part extensible (amélioration de l'optimiseur par l'ajout de nouvelles règles). L'efficacité par rapport aux autres algorithmes de filtrage ne peut pas être mesurée précisément car les contraintes diffèrent beaucoup d'une méthode à l'autre. Cependant, les tests ont montré une nette amélioration des performances à l'exécution. Par exemple, pour un système de plus de 100 règles de filtrage syntaxique, le temps d'exécution est divisé par 5.

Ce stage montre que l'utilisation de règles de transformations de programmes pour optimiser le filtrage est une solution à la fois élégante et évolutive par rapport aux méthodes existantes à base d'automates. La définition d'un ensemble de règles simples couplé à l'utilisation d'une stratégie fine de contrôle est une puissante méthode de compilation et d'optimisation applicable à la fois pour les langages à base de règles mais aussi pour les langages fonctionnels. La définition de la stratégie de l'optimiseur a été à l'origine de la définition de nouveaux opérateurs de stratégie. Ces opérateurs basés sur l'identité améliorent nettement le calcul de forme normale, opération omniprésente dans les systèmes de réécriture.

La combinaison des travaux de D. Lacey pour les conditions temporelles et de E. Visser pour la définition de stratégies nous a permis de formaliser un nouveau cadre de transformations de programmes permettant de raisonner à la fois sur l'arbre syntaxique abstrait et le graphe de flot de contrôle. En choisissant de faire référence au niveau de chaque sommet du graphe de flot de contrôle à un nœud de l'arbre syntaxique abstrait, on obtient une structure de représentation (arbre syntaxique abstrait+graphe de flot de contrôle) à la fois statique et dynamique avec un partage maximal des informations. C'est une démarche nouvelle car en général, la construction des représentations intermédiaires est réalisée indépendamment, dupliquant ainsi l'information sur le programme. Enfin, l'utilisation de la logique temporelle dans les conditions est une manière élégante de calculer de l'information sur le graphe de flot de contrôle et qui simplifie le calcul des prédicats. Ce sont les opérateurs temporels qui contrôlent le parcours du graphe de flot de contrôle, les prédicats ne portent que sur le sommet courant de la formule temporelle dont ils font parti.

L'étude de la bibliothèque Mutraveler [VeABT98] nous a permis de définir une stratégie fine d'optimisation. Cette bibliothèque pourrait être aussi utilisée pour l'implantation des opérateurs temporels utilisés dans les conditions sur le graphe de flot de contrôle. Vérifier une certaine condition temporelle sur le graphe de flot de contrôle peut revenir à appliquer une certaine stratégie définie par les opérateurs de stratégie sur le graphe de flot de contrôle. Ainsi, l'échec d'une stratégie correspond à une formule fausse et la réussite à une formule vraie. De plus, pour améliorer les règles, au lieu d'utiliser une équivalence syntaxique, on peut imaginer une équivalence plus fine qui se rapproche plus de l'équivalence sémantique. Cette équivalence peut par exemple utiliser une notion d'environnement. A chaque sommet du graphe de flot de contrôle, on calcule un environnement qui donne des informations sur la valeur des variables et permet de déterminer plus finement si deux expressions sont équivalentes ou incompatibles.

A plus long terme, cette approche de transformations de programmes par réécriture à base de logique temporelle pourrait s'appliquer à d'autres domaines comme par exemple l'analyse de programmes. Dans ce cas, les transformations ne seraient plus source-à-source mais permettrait statiquement d'extraire de l'information du programme.

Bibliographie

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

[BHKMR98] Peter Borovanský, Claude Kirchner Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. In *WRLA'98 : Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications*, volume 15, Pont-à-Mousson (France), 1998. Electronic Notes in Theoretical Computer Science.

[Car84] Luca Cardelli. Compiling a functional language. In *LFP'84 : Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 208–217, 1984.

[CELM96] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), 1996.

[Chi93] Wei-Ngan Chin. Towards an automated tupling strategy. In *PEPM'93 : Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages iv + 215. ACM Press, 1993.

[DGT96] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Partial Evaluation, International Seminar (Dagstuhl Castle, Germany, February 12–16, 1996)*, volume 1110 of *LNCS*, 1996.

[FM01] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *ICFP'01 : Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming*, pages 26–37. ACM Press, 2001.

[Grä91] Albert Gräf. Left-to-right tree pattern matching. In *RTA'91 : Proceedings of the 4th international conference on Rewriting Techniques and Applications*, volume 488 of *LNCS*, pages 323–334. SV, 1991.

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

[KMR05] Claude Kirchner, Pierre-Etienne Moreau, and Antoine Reilles. Formal validation of pattern matching code. In *PPDP'05 : Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, 2005.

[Lac03] David Lacey. *Program Transformation using Temporal Logic Specifications*. PhD thesis, Oxford University Computing Laboratory, 2003.

[MRV03] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. In *CC'2003 : 12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76, 2003.

[OV02] Karina Olmos and Eelco Visser. Strategies for source-to-source constant propagation. In *WRS'02 : Workshop on Reduction Strategies*, volume 70 of *Electronic Notes in Theoretical Computer Science*, page 20, Copenhagen, Denmark, 2002.

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

[Tur86] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3) :292–325, 1986.

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

[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, and Joost Visser. The ASF+SDF Meta-Environment : a Component-Based Language Development Environment. In *CC'2001 : Proceedings of Compiler Construction*, volume 2027 of *LNCS*. Springer, 2001.

[VeABT98] Eelco Visser, Zine el Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *ICFP'98 : Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 13–26, 1998.

[Vis01a] Eelco Visser. Stratego : A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *RTA'01 : Rewriting Techniques and Applications*, volume 2051 of *LNCS*, pages 357–361, 2001.

[Vis01b] Joost Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11) :270–282, 2001.

[Wad88] Philip Wadler. Deforestation : transforming programs to eliminate trees. In *Proceedings of the Second European Symposium on Programming*, pages 231–248. North-Holland Publishing Co., 1988.