# THÈSE

présentée par

**Christophe ALIAS**

pour obtenir le titre de

DOCTEUR de l'UNIVERSITÉ de VERSAILLES

Spécialité : Informatique

Sujet de la thèse :

## Optimisation de programmes par reconnaissance de templates

## *Program Optimization by Template Recognition and Replacement*

Soutenue le *6 décembre 2005*

Devant la commission d'examen composée de :

| | | | | |
|---|---|---|---|---|
| Pr. | William | JALBY | Président | Université de Versailles |
| Pr. | François | IRIGOIN | Rapporteur | ENSM Paris |
| Pr. | Pierre | LESCANNE | Rapporteur | ENS Lyon |
| Pr. | Paul | FEAUTRIER | Directeur | ENS Lyon |
| Dr. | Denis | BARTHOU | Examinateur | Université de Versailles |

Thèse préparée à l'Université de Versailles au sein du laboratoire PR*i*SM,
Unité Mixte de Recherche CNRS n°8144.

# Remerciements

Cette thèse, bien qu'étant le fruit d'un travail personnel, n'aurait certainement pas pu aboutir sans la contribution d'un certain nombre de personnes que je tiens à remercier, sincérement, ici.

En premier lieu, je tiens à remercier William Jalby pour avoir accepté de présider mon jury de thèse. Son dynamisme de tous les instants et sa capacité de travail impressionnante en font un chercheur hors pair à qui je tiens ici à rendre hommage.

Je tiens également à remercier François Irigoin ainsi que Pierre Lescanne pour l'honneur qu'ils m'ont fait en acceptant de rapporter ma thèse. Leurs précieuses remarques m'ont permis d'améliorer grandement la qualité du manuscrit de thèse, et ont apporté un éclairage nouveau sur mes travaux.

Je tiens à exprimer toute ma gratitude à Paul Feautrier pour la confiance qu'il m'a témoigné en acceptant de diriger ma thèse. Ses conseils éclairés ainsi que ses nombreuses remarques et suggestions ont permis à cette thèse d'atteindre sa maturité actuelle.

A tout seigneur tout honneur, que Denis Barthou soit vivement remercié pour avoir encadré ma thèse avec autant de professionnalisme. Sa disponibilité permanente, ses nombreuses remarques et suggestions et son incroyable aptitude à supporter mon caractère ont fait de lui un encadrant exemplaire que je tiens, encore une fois, à remercier ici. *Let the mana be with you*, Denis.

Sans oublier (comment le pourrais-je ?) mes collègues et amis, Karine, Christophe, Patrick, Cédric et les autres, qui ont largement contribué à rendre ces années de thèse le plus agréable possible, malgré les inévitables moments difficiles. J'ai une pensée particulière pour mes amis Algériens Amdjed, Issam, Lotfi et Lamia qui m'ont beaucoup apporté, et à qui je souhaite la même réussite[1].

Une dernier paragraphe (mais pas le moindre) pour remercier ma famille, dont le soutien inconditionnel et sans faille à largement contribué à l'aboutissement de ce travail, et sans qui mon pot de thèse n'aurait pas été si savouveux ;-) Encore merci à tous !

[1] Sauf Amdjed qui à déjà soutenu!

# Table des matières

# Présentation

Cette partie offre un résumé en français de la thèse écrite en anglais. Son organisation respecte celle du document original et chacune de ses sections correspond à un chapitre ou une partie dont les développements, algorithmes, preuves et exemples ont été simplifiés ou retirés. Le lecteur s'intéressant à un sujet particulier est donc naturellement invité à se reporter aux chapitres correspondants pour y trouver un niveau de détail satisfaisant.

# I Introduction

La *reconnaissance de templates* consiste à trouver dans un programme toutes les parties qui peuvent être réécrites comme une instance d'un template. Le reconnaissance de templates trouve de nombreuses applications en conception de compilateurs et en génie logiciel. Citons par exemple le reverse-engineering, la vérification de transformations de programmes, la factorisation de code ou encore l'optimisation de programmes. Dans cette thèse, nous nous intéressons au problème de la reconnaissance de templates et à son application à l'optimisation de programmes. Nous proposons une nouvelle optimisation *source-à-source* qui réécrit automatiquement un programme pour utiliser une bibliothèque optimisée. Etant donné un programme, notre méthode détecte les implémentations directes des fonctions d'une bibliothèque, et les remplace par l'appel de fonction approprié, lorsque c'est possible et intéressant.

## I.1 Motivations

Cette thèse part d'un constat : quelque soit la qualité des optimisations, elles ne remplaceront jamais un bon algorithme. La plupart des optimisations appliquent en effet des transformations locales bas-niveau, sans se soucier du calcul exprimé par le programme [1] (propagation de constantes, simplifications algébriques, *inlining*) ; tandis que d'autres optimisations tendent à exploiter le *hardware* de la meilleure manière (parallélisation automatique [110], sélection optimale d'instructions [38], *pipeline* logiciel [7], allocation de registres[20], *data prefetching* [78], hiérarchie mémoire [75]). Bien que ces optimisations produisent des résultats satisfaisants, elles ne remplacent pas un bon algorithme, et amènent bien souvent le programmeur à choisir entre:

- **Optimisation à la main.** Malheureusement, les langages de programmation actuels cachent souvent les comportements critiques de l'architecture. Les effets de cache, par exemple, rendent les performances imprévisibles. Par conséquent, il est presque impossible de trouver à la main une optimisation proche de l'optimal.

- **Routines optimisées.** Une solution immédiate est de chercher une routine optimisée existante. Elles se déclinent sous différentes formes, dont les *langages métier* (Domain Specific Language, DSL) [107, 25], la programmation générative [28], ou encore les bibliothèques adaptatives [99, 47, 22].

  Les DSL fournissent un niveau d'abstraction permettant d'effectuer des optimisations haut-niveau très aggressives. Par ailleurs, le programme résultant est souvent plus facile à lire, et donc à maintenir. Bien que les DSLs soient une alternative séduisante, ils posent des problèmes de portabilité, et de support à long terme.

  La *programmation générative* permet d'écrire un code en utilisant des constructions de haut niveau, et de générer le code correspondant dans le langage usuel (typiquement C ou Fortran). Lex et Yacc sont des exemples typiques d'outils de programmation générative, qui génèrent des analyseurs syntaxiques efficaces, et faciles à maintenir. Comme les DSL, les langages génératifs travaillent à un niveau d'abstraction qui permet des optimisations aggressives.

  Les *bibliothèques adaptatives* fournissent un panel de routines domain-specific tunées pour l'architecture courante. Lors de l'installation, les routines sont exécutées en faisant varier les paramètres qui affectent les performances (par exemple, la structure des boucles, la taille des tuiles, encore le facteur de déroulage). L'espace des paramètres est exploré point par point, mesurant les performances de chaque variante, jusqu'à ce que la meilleure implémentation soit atteinte. Cette technique est utilisée par de nombreuses bibliothèques optimisées comme ATLAS [99], FFTW [47] ou PhiPAC [22]. Malgré le coût du tuning, les bibliothèques adaptatives semblent être une voie prometteuse pour l'optimisation.

  La *programmation générique* [95, 93, 9, 97, 61, 28] est un paradigme récent qui utilise des fonctions paramétrables par des types de données et des opérations génériques. Outre le gain en temps de développement, la taille des bibliothèques s'en trouve réduite, un seul template représentant une famille d'algorithmes. On évite ainsi de faire exploser la taille de la bibliothèque en ajoutant une spécialisation de chaque opération par structure de données. Par exemple, la Matrix Template Library [93] abstrait l'anneau sous-jacent et les opérations correspondantes ($+$ et $\times$). En effet, 13 types de stockage, et 4 anneaux correspondant à différents niveaux de précision ont étés ajoutés, permettant de représenter plus d'une centaine de versions de la même routine! Comme la programmation générique est relativement nouvelle, il n'y a encore qu'un nombre restreint de bibliothèques génériques, mais nous croyons que ce paradigme sera très largement utilisé dans le futur.

Apprendre et utiliser une nouvelle bibliothèque est une tâche fastidieuse, et il est surprenant de voir le peu d'aide apporté par le compilateur. Une solution naturelle serait de chercher les occurrences immédiates des fonctions de bibliothèque dans le programme, et de les remplacer par l'appel correspondant. Dans cette thèse, nous proposons une approche totalement automatique pour reconnaitre les occurrences des fonctions de bibliothèque, et les remplacer, lorsque c'est possible et intéressant par l'appel approprié. En plus de fonctions, nous sommes également capables de trouver des *instances de templates* dans le programme, fournissant les paramètres correspondants. Une telle caractéristique

rend possible la réécriture automatique d'un programme pour utiliser une bibliothèque générique.

## I.2 Exemple d'optimisation par reconnaissance de templates

La figure 1 donne un exemple de template (a) à trouver dans un programme (b). Le template représente une réduction générique sur un tableau $I$ selon l'opérateur $X$. On cherche a remplacer les réductions trouvées dans le programme par un appel vers une version parallèle `par_reduc(`$X$`,I)`. La première étape de notre approche est de reconnaître *toutes* les instances du template dans le programme. Ici, nous obtenons les instances suivantes:

- Lignes 1 à 8, avec $X(x, y) = x+y^5$, $n = 10$, $I(0) = 0$ et $I(k) = a(k)$ pour $1 \leq k \leq 10$.
- Lignes 3 à 6, avec $X(x, y) = x \times y$, $n = 5$, $I(0) = 1$ et $I(k) = a(i)$ pour $1 \leq k \leq 5$.

Puisque les deux parties reconnues (*slices*) se recouvrent, on doit choisir la partie susceptible de fournir la meilleure amélioration de performance. Par exemple, on peut choisir le second slice. Il reste alors à générer le code avec la substitution. (voir (c)).

```
s = I(0)
do i = 1,n
| s = X(s,I(i))
enddo
return s
```

(a) Template

```
1 s = 0
2 do i = 1,10
3 | p = 1
4 | do j = 1,5
5 | | p = p*a(i)
6 | enddo
7 | s = s + p
8 enddo
```

(b) Programme

```
s = 0
do i = 1,10
| I(0) = 1
| I(1:5) = a(i)
| p = call par_reduc(λxy.x×y,I)
| s = s + p
enddo
```

(c) Programme optimisé

FIG. 1 – Un problème de reconnaissance de templates

## I.3 Difficultés de la reconnaissance de templates

### Variations algorithmiques

La principale difficulté en reconnaissance de template vient des manières très variées d'implémenter un algorithme donné. Wills [102] donne une classification de tous les types de variations qui peuvent être trouvées dans un programme, que nous présentons sur la figure 7.

En théorie, la reconnaissance d'algorithmes est plus difficile que l'équivalence sémantique entre programmes, qui est elle-même indécidable. La plupart des approches existantes pour la reconnaissance d'algorithmes sont basées sur un matching syntaxique, et peuvent uniquement gérer les variations d'organisation. Malheureusement, la plupart des variations trouvées dans les programmes sont des variations de structures de données, et des variations de contrôle. Cette thèse propose une heuristique capable de reconnaitre dans la plupart des cas les variations d'*organisation*, de *structure de données* et de *contrôle*. Notre approche ne peut toutefois pas gérer les variations sémantiques.

**Variations d'organisation**

N'importe quelle permutation de statements indépendents, et introduction de variables temporaires. L'exemple suivant donne une variation d'organisation avec des permutations légales (LP), du *garbage code* (GC) et des temporaires (T):

```
s = a(0)                    s = a(0)
c = 0                       c = 0
do i = 1, n             GC  garbage = 0
| s = s + a(i)              do i = 1, n
| c = c + 1           LP    | c = c + 1
enddo                   T   | temp = a(i)
return s + c                | do j = 1, p
                        GC  | | garbage = garbage + 1
                            | enddo
                            | s = s + temp
                        GC  | garbage = garbage + a(i)
                            enddo
                            return s + c
```

**Variations de contrôle**

N'importe quelle transformation de contrôle comme la if-conversion, l'élimination de code mort, ou n'importe quelle transformation de boucles comme le peeling, le splitting ou le skewing. L'exemple suivant donne une variation de contrôle avec un peeling:

```
s = a(0)                    s = a(0)
do i = 1,n                  s = s + a(1)
| s = s + a(i)              do i = 2,n-1
enddo                       | s = s + a(i)
return s                    enddo
                            s = s + a(n)
                            return s
```

**Variations de structure de données**

Le même calcul en utilisant une structure de données différente. L'exemple suivant donne une variation de structure de données avec des tableaux et des enregistrements:

```
s(0) = a(0)                 s.sum1 = a(0)
do i = 1, 2*n               do i = 1, n
| s(i) = s(i-1) + a(i)      | s.sum1 = s.sum1 + a(i)
enddo                       enddo
return s(2*n)               s.sum2 = a(n+1)
                            do i = n+2, 2*n
                            | s.sum2 = s.sum2 + a(i)
                            enddo
                            return s.sum1 + s.sum2
```

**Variations sémantiques**

N'importe quelle transformation qui fait des hypothèses sur les propriétés des opérateurs. La transformation suivante suppose la commutativité de +:

```
s = a(0)                    s = a(0)
do i = 1,n                  do i = 1,n
| s = s + a(i)              | s = s + a(n-i+1)
enddo                       enddo
return s                    return s
```

FIG. 2 – Classification des variations de programmes

### Prédiction de performances

La prédiction de performances permet ici de quantifier l'impact d'une substitution sur les performances du programme. De la même manière que l'équivalence sémantique, on ne peut pas décider si une transformation améliore effectivement les performances d'un programme [79]. Néanmoins, plusieurs approches proposent des modèles précis, comme ceux de Fahringer [40], Ghosh [51] ou Padua [21]. Ces approches fournissent des paramètres quantifiant les performances du programme, comme le temps d'exécution, ou le nombre de défauts de cache ; elles utilisent souvent du profiling. Cette thèse propose une approche à base de benchmarking adaptée à la problématique de la reconnaissance de templates.

## I.4 Plan

La suite du résumé est structurée de la façon suivante. La section II présente les notions nécessaires à la compréhension de notre approche. En particulier, une formalisation du problème de reconnaissance de templates est présentée, ainsi qu'une décomposition importante, sur laquelle notre méthode se calque. La section III donne une vue d'ensemble

de la méthode et précise le rôle de chaque étape. Les sections IV, V et VII.2 présentent briévement chaque étape de notre méthode, dont on trouvera le détail et les preuves dans la version anglaise. La section VII présente les résultats expérimentaux obtenus. Enfin, la section VIII conclut et présente les pistes de recherches ouvertes par nos travaux.

# II Définitions et notations

Cette section définit les notions qui seront utilisées dans la suite de cette thèse. Une description formelle du problème de la reconnaissance de templates est également donnée. Après avoir présenté le modèle de programme étudié, on défini la notion de *template*. On présente ensuite l'*équivalence de Herbrand*, le sous-ensemble de l'équivalence sémantique étudié dans cette thèse. Enfin, on défini le problème de la reconnaissance de template, et on présente la décomposition de ce problème qui sera suivie dans cette thèse.

## II.1 Modèle de programme

La méthode présentée dans cette thèse est capable de traiter des programmes généraux. Ceci étant, les parties de programmes à reconnaitre doivent être des *programmes à contrôle statique*.

**Définition 1 (Programme à contrôle statique).** *Un programme est à contrôle statique s'il vérifie les conditions suivantes:*

- *Les structures de données sont des scalaires et des tableaux.*
- *Les structures de contrôle autorisées sont la séquence, les conditionnelles* `if` *et les boucles* `do`.
- *Les conditions, les bornes des boucles ainsi que les fonctions d'accès aux tableaux sont des expressions* affines *des compteurs des boucles englobantes, et des paramètres de structure.*
- *Les blocs de base (*statements*) sont des affectations*

La plupart des routines utilisées pour la résolutions de problèmes d'algèbre linéaire sont des programmes à contrôle statique : décomposition $LU$, factorisation de Cholesky, etc. La figure 3.(a) donne l'exemple bien connu du produit de deux polynômes. On notera la fonction d'index affine $(i, j) \mapsto i + j$ de `c`, ainsi que les paramètres de structure `n` et `m`, qui correspondent ici aux degrés des polynômes `a` et `b`.

Un *vecteur d'itération* d'un statement $S$ est constitué des valeurs des compteurs des boucles englobant $S$. L'ensemble des vecteurs d'itération de $S$ pendant l'exécution est appélé *domaine d'itération* de $S$. Les domaines d'itération des programmes à contrôle statique ont la bonne propriété d'être des $\mathbb{Z}$-polyèdres calculables lors de la compilation. Sous ces restrictions, de nombreux problèmes peuvent être décidés, comme l'analyse de dépendances *exacte* (*instance-wise dataflow analysis*) [43].

Une *opération* est une instance d'un statement $S$ pendant l'exécution *i.e.* un élément de la trace d'exécution. Elle est généralement notée $\langle S, \vec{i} \rangle$, où $\vec{i}$ est un vecteur d'itération de $S$. Par exemple, $\langle S_2, n, m \rangle$ est la dernière opération du produit de polynômes.

```
   do i = 0,n+m
S1 | c(i) = 0
   enddo
   do i = 0,n
   | do j = 0,m
S2 | | c(i+j) = c(i+j) + a(i)*b(j)
   | enddo
   enddo
```

(a) Produit de polynômes

| | |
|---|---|
| $\langle S_1, 0\rangle$ | `c(0) = 0` |
| $\langle S_1, 1\rangle$ | `c(1) = 0` |
| $\langle S_1, 2\rangle$ | `c(2) = 0` |
| $\langle S_2, 0, 0\rangle$ | `c(0) = c(0) + a(0)*b(0)` |
| $\langle S_2, 0, 1\rangle$ | `c(1) = c(1) + a(0)*b(1)` |
| $\langle S_2, 1, 0\rangle$ | `c(1) = c(1) + a(1)*b(0)` |
| $\langle S_2, 1, 1\rangle$ | `c(2) = c(2) + a(1)*b(1)` |

(b) Trace d'exécution pour $n = m = 1$

FIG. 3 – Produit de deux polynômes

## II.2 Templates

Cette thèse étudie la reconnaissance de *templates* dans un programme. D'une certaine façon, les programmes à contrôle statique sont des templates, puisque qu'ils dépendent de paramètres de structure (cf. `n` et `m` dans l'exemple 3). On améliore le niveau de généricité en permettant de paramètrer les programmes à contrôle statique avec des *fonctions pures*, comme précisé dans la définition suivante:

**Définition 2 (Template).** *Un* template *est un programme à contrôle statique paramétré par un tuple de fonctions* $(X_1 \dots X_n)$*, qu'on appelle des* fonctions libres *ou des* variables de template. *Chaque fonction libre* $X_i$ *est supposée* pure *(pas d'effets de bord), et doit être utilisée dans le membre droit d'une affectation:*

$$\dots \texttt{ = } \dots X_i(\dots) \dots$$

Le dernier point interdit l'utilisation des fonctions libres dans les conditions, et les bornes des boucles `do`. Cette définition est générale, et ne restreint pas l'ordre des fonctions libres. Dans cette thèse, on supposera que les fonctions libres sont du *second-ordre au plus*. Par exemple, voici le template d'une réduction paramétrée par la fonction libre $X$, et le paramètre de structure `n`:

```
s = a(0)
do i = 1,n
| s = X(s,a(i))
enddo
return s
```

La fonction libre $x$ peut être substituée par n'importe quel morceau de code qui calcule une valeur à partir de `s` and `a(i)`, sans effet de bord.

Une *instance* d'un template $T$ est un programme obtenu en substituant les fonctions libres $\vec{X} = (X_1 \dots X_n)$, les paramètres $\vec{n} = (n_1 \dots n_p)$, et les inputs $\vec{I} = (I_1 \dots I_n)$ avec des valeurs spécifiques. Elle est notée $T[\vec{X}, \vec{n}, \vec{I}]$. Par exemple, on a:

$$T\left[\begin{array}{l}\lambda x.\lambda y.\\ \\ \\ \\ \end{array}\begin{array}{l}\texttt{p = 1}\\ \texttt{do j = 1,5}\\ \texttt{| p = p * } y\\ \texttt{enddo}\\ \texttt{return } x \texttt{ + p}\end{array}\quad,\ \texttt{n} = 10,\ a = \left[\begin{array}{c}1\\ B(1)\\ \vdots\\ B(n)\end{array}\right]\right] = \begin{array}{l}\texttt{s = 1}\\ \texttt{do i = 1,10}\\ \texttt{| p = 1}\\ \texttt{| do j = 1,5}\\ \texttt{| | p = p * B(i)}\\ \texttt{| enddo}\\ \texttt{| s = s + p}\\ \texttt{enddo}\\ \texttt{return s}\end{array}$$

De manière générale, $X$ n'est pas restreint aux programmes à contrôle statique, il peut être défini par n'importe quelle portion de code avec des boucles `while` et des expressions non affines dans les conditions, les bornes de boucles, les fonctions d'index des tableaux, etc. Cependant, on supposera dans cette thèse que $X$ ne peut être défini que par un programme à contrôle statique.

## II.3 Définitions visibles

Comme précisé en section 2.1, l'exécution d'un programme sur l'input $I$ peut être vue comme une séquence d'opérations $\omega_1^I;\ \omega_2^I;\ \ldots;\ \omega_n^I;\ \ldots$ possiblement infinie. On définit l'*ordre d'exécution* $\prec_I$ entre opérations comme: $\omega_i^I \prec_I \omega_j^I \Leftrightarrow i < j$. $\prec_I$ définit un ordre total sur les opérations $\omega_i^I$. On précise maintenant la notion de *définition visible*, qui sera utilisée tout au long de cette thèse.

**Définition 3 (Définition visible).** *Considérons un programme exécuté sur un input $I$, et une opération $\omega$ lisant une variable $v$. La* définition visible *de la variable $v$ est la dernière opération $\tau$ exécutée avant $\omega$ qui écrit $v$. Elle généralement notée* $\mathsf{RD}_\omega^I(v)$ *(pour* reaching definition*):*

$$\mathsf{RD}_\omega^I(v) = \max_{\prec_I}\{\tau \mid \tau \prec_I \omega \text{ et } \tau \text{ écrit } v\}$$

La séquence d'opérations exécutée par les programmes à contrôle statique ne dépend pas de l'input $I$. Par conséquent, on peut simplifier les notations et écrire: $\mathsf{RD}_\omega(v)$ pour $\mathsf{RD}_\omega^I(v)$. Dans l'exemple suivant:

```
S1  sum = a(0)
    do i = 1,n
S2  | sum = sum + a(i)
    enddo
```

La définition visible de `sum` lue par $\langle S_2, \vec{i}\rangle$ est :

$$\mathsf{RD}_{\langle S_2,\vec{i}\rangle}(\texttt{sum}) = \begin{cases} i = 1: & \langle S_1, \rangle \\ 2 \le i \le n: & \langle S_2, i-1\rangle \end{cases}$$

Bien que le calcul des définitions visibles soit *indécidable* sur des programmes généraux, il est possible dans les programmes à contrôle statique. Feautrier [43] en donne une solution exacte que nous présentons page 88, chapitre 6.

## II.4 Notion de slice

Un *slice* est une tranche de programme qui peut être abstraite dans une fonction sans modifier la sémantique du programme. La notion de slice a été introduite pour la première fois par Weiser [98] pour aider ses étudiants à débugguer leurs programmes.

La définition de Weiser est trop imprécise pour notre propos puisqu'elle travaille au niveau du statement. Nous proposons ci après une autre définition d'un slice, au niveau de l'*opération*:

**Définition 4 (Full slice).** *Considérons un programme $P$ exécuté sur un input $I$, et notons $\tau^I$ une des opérations exécutées. Le* full slice *du programme $P$ selon l'opération $\tau^I$ est la limite de la suite $(\Omega_i)_i$ définie par:*

$$\begin{array}{ll} \Omega_0 & = \{\tau^I\} \\ \Omega_{i+1} & = \Omega_i \cup \bigcup_{\omega\in\Omega_i} \bigcup_{v\in\mathsf{rhs}(\omega)} \mathsf{RD}_\omega(v) \end{array}$$

*où $\mathsf{rhs}(\omega)$ dénote l'ensemble des variables lues par $\omega$. On écrit: $\widehat{\mathcal{S}}^I_P(\tau^I) = \lim_{i\to\infty} \Omega_i$.*

Basiquement, un *full slice* contient exactement l'ensemble des operations nécéssaires pour calculer la variable écrite par $\tau^I$. Il est calculé en remontant les définitions visible des variables lues par $\tau^I$.

Avec cette définition, un *full slice* capture le morceau de programme requis pour calculer une variable. Un *slice* est quelque peu différent, puisqu'il peut commencer son calcul en partant de « paramètres » écrits par plusieurs opérations. Basiquement, un slice est une tranche de programme qui peut être abstraite dans une fonction pouvant dépendre de paramètres.

**Définition 5 (Slice).** *Reprenant les notations de la définition 2.5, et notant $\iota = (\iota^I_1 \dots \iota^I_n)$ un tuple d'opérations où chaque $\iota^I_i$ est un ancêtre de $\tau^I$, le* slice *de $P$ selon l'output $\tau^I$ et l'input $\iota$ est la limite de la suite $(\widetilde{\Omega}_i)_i$ définie par :*

$$\begin{array}{ll} \widetilde{\Omega}_0 & = \{\tau^I\} \\ \widetilde{\Omega}_{i+1} & = \widetilde{\Omega}_i \cup \bigcup_{\omega\in\widetilde{\Omega}_i} \bigcup_{v\in\widetilde{\mathsf{rhs}}(\omega)} \mathsf{RD}_\omega(v) \end{array}$$

*Où $\widetilde{\mathsf{rhs}}(\omega) = \emptyset$ si $\omega$ est un $\iota^I_i$, et $\widetilde{\mathsf{rhs}}(\omega) = \mathsf{rhs}(\omega)$ sinon. On écrit $\mathcal{S}^I_P(\iota, \tau^I) = \lim_{i\to\infty} \widetilde{\Omega}_i$.*

Encore une fois, on écrira $\mathcal{S}_P(\iota, \tau^I)$ pour $\mathcal{S}^I_P(\iota, \tau^I)$ lorsque $I$ n'est pas requis. C'est en particulier le cas pour les programmes à contrôle statique.

## II.5 Equivalence de Herbrand

Dans cette thèse, on considère une équivalence plus faible que l'équivalence sémantique appelée *équivalence de Herbrand*. Au lieu d'indiquer que deux algorithmes calculent la même fonction (au sens mathématique), l'équivalence de Herbrand indique qu'ils utilisent même formule, syntaxiquement. Considérons un algorithme $A$ qui prend en entrée un tableau $I$, et qui retourne un tableau $O$. $A$ réalise son calcul en utilisant un ensemble $\mathsf{Fun}$ de *fonctions atomiques i.e.* constantes, opérations arithmétiques, et opérations internes

au langage utilisé. Par exemple, le produit de deux polynômes $\times_{\mathbb{R}[X]}$ donné en figure 3.(a) utilise les fonctions atomiques +, * et 0.

Si on exécute $A$ sur l'input $I$ en maintenant les fonctions atomiques non-interprétées, on obtient un tableau $\mathcal{T}_A(I)$ de *termes* sur Fun. Par exemple, en prenant $a = [\ 1\ ,\ 0\ ]$ et $b = [\ 0\ ,\ 1 + 2\ ]$, on obtient :

$$\mathcal{T}_{\times_{\mathbb{R}[X]}}(a, b) = [\ (0 + (1 * 0)) + (0 * (1 + 2))\ ,\ (0 + (1 * (1 + 2))) + (0 * 0)\ ]$$

Suivant les travaux de Knoop *et al.* [91], on étend l'équivalence de Herbrand aux algorithmes, comme précisé dans la définition suivante.

**Définition 6 (Equivalence de Herbrand).** *Deux algorithmes $A_1$ et $A_2$ sont* Herbrand-equivalents *ssi pour chaque input $I$:*

- *soit $A_1$ et $A_2$ ne s'arrêtent pas*
- *soit $A_1$ et $A_2$ s'arrêtent et retournent le même tableau de termes:*

$$\mathcal{T}_{A_1}(I)[\vec{i}] = \mathcal{T}_{A_2}(I)[\vec{i}]$$

  *pour chaque $\vec{i}$.*

*Dans ce cas, on note $A_1 \equiv_{\mathcal{H}} A_2$.*

En d'autres termes, $A_1$ et $A_2$ sont Herbrand-équivalents s'ils calculent le même tableau de termes. La proposition suivante montre que $\equiv_{\mathcal{H}}$ définit bien une équivalence sur les programmes, qui est plus faible que l'équivalence sémantique.

**Proposition 1 (Correction).** *Etant donnés deux algorithmes $A_1$ et $A_2$, on a :*

$$A_1 \equiv_{\mathcal{H}} A_2 \Longrightarrow A_1 \equiv A_2$$

Bien que l'équivalence de Herbrand soit plus faible que l'équivalence sémantique, on ne peut pas la décider sur des programmes généraux [13]. En dépit de classes d'équivalence plus petites que l'équivalence sémantique, l'équivalence de Herbrand couvre toutes les variations de programme qui ne changent pas le terme calculé par le programme. Ces variations incluent notament toutes les transformations de boucle standard, telles que le splitting, la fusion, le déroulage, la torsion de domaines d'itérations, ou encore le tiling. On pourra trouver plus de transformations de boucles dans le livre de Wolfe [104]. Pour les mêmes raisons, les variations d'organisation et de structure de données sont couvertes. Ces caractéristiques importantes de l'équivalence de Herbrand sont résumées en figure 2.7. Malheureusement, l'équivalence de Herbrand *ne couvre pas* les variations qui utilisent les propriétés sémantiques des opérateurs. Par exemple l'associativité et la commutativité de +.

## II.6 Reconnaissance de templates

Cette section présente une formalisation du problème de reconnaissance de templates. Après avoir défini le problème du *matching* entre un programme et un template, on présente une formalisation du problème de reconnaissance de templates. En particulier, on en donne une décomposition intéressante, qui sera suivie tout au long de cette thèse.

Le problème du *matching* est de décider si un programme est une instance d'un template, comme précisé dans la définition suivante:

**Problème 1 (Matching de template).** *Le problème du* matching *entre un template $T$ et un programme $P$ selon une équivalence $\sim$ sur les programmes, consiste à décider s'il existe une instance de $T$ qui est équivalent à $P$ au sens de $\sim$. Plus précisement, on cherche à trouver les fonctions libres $\vec{X}$, les paramètres $\vec{n}$ et les inputs $\vec{I}$ tels que:*

$$T[\vec{X}, \vec{n}, \vec{I}] \sim P$$

*Par analogie avec la théorie de l'unification, une telle solution, si elle existe, est appelée un* semi-unificateur *du problème de matching $T \stackrel{?}{=} P$.*

Puisque ce problème est plus difficile que l'équivalence selon $\sim$, il est indécidable selon l'équivalence sémantique, et l'équivalence de Herbrand.

La définition suivante introduit le problème plus général de la reconnaissance de templates, qui a pour but de trouver dans un programme les instances d'un template donné.

**Problème 2 (Reconnaissance de template).** *Etant donné un template $T$ et un programme $P$, le* problème de reconnaissance de template $T \leq^? P$ *selon une équivalence $\sim$ consiste à trouver tous les slices de $P$ qui matchent $T$ selon $\sim$.*

Ce problème est clairement plus compliqué que le problème du matching de template, et est par conséquent indécidable selon l'équivalence sémantique, et l'équivalence de Herbrand.

Dans cette thèse, on étudie le problème de reconnaissance de templates selon l'équivalence de Herbrand $\equiv_{\mathcal{H}}$. Une sur-approximation de l'équivalence de Herband est utilisée pour trouver les slices qui matchent le template *a priori*. Le résultat est un ensemble qui contient les bons slices (conservativité). Une sous-approximation exacte de l'équivalence de Herbrand est ensuite utilisée pour filtrer les bons candidats. La proposition suivante établit formellement cette décomposition pour n'importe quelle équivalence $\sim$ sur les programmes.

**Proposition 2 (Décomposition).** *On considère un template $T$, un programme $P$, une équivalence $\sim$ sur les programmes et $\approx$, une sur-approximation de $\sim$ ($\sim \subseteq \approx$). La procédure suivante résout le problème de reconnaissance de template entre $T$ et $P$ selon $\sim$:*

1. Résoudre le problème de reconnaissance de template $T \leq^? P$ selon $\approx$.
2. Pour chaque slice $S$ obtenu:

   Résoudre le problème de matching $T \stackrel{?}{=} S$ selon $\sim$.

On doit bien sûr trouver un *trade-off* dans le choix de $\approx$. En effet, une approximation trop grossière fournirait trop de slices à la coûteuse étape 2, alors qu'une approximation trop précise serait trop coûteuse.

La méthode de reconnaissance décrite dans cette thèse suit cette décomposition. L'étape 1 est appelée *slicing*, et exposée brièvement dans la section IV. On consultera le chapitre 5 pour une description complète de la méthode, avec les preuves. Nous proposons également deux heuristiques différentes pour réaliser l'étape 2 (*test d'instanciation*). La section V présente le test d'instanciation utilisant les automates d'arbre, dont on trouvera une description approfondie chapitre 8. Un deuxième test, moins coûteux, mais également moins puissant, est proposé chapitre 7. Ces deux procédures résolvent le problème du matching selon des équivalences $\sim_1$ *resp.* $\sim_2$, avec $\sim_1 \subset \sim_2 \subset \equiv_{\mathcal{H}} \subset \equiv$.

# III Vue d'ensemble de la méthode

La figure 4 donne une vue d'ensemble de la méthode. On suppose que le concepteur de la

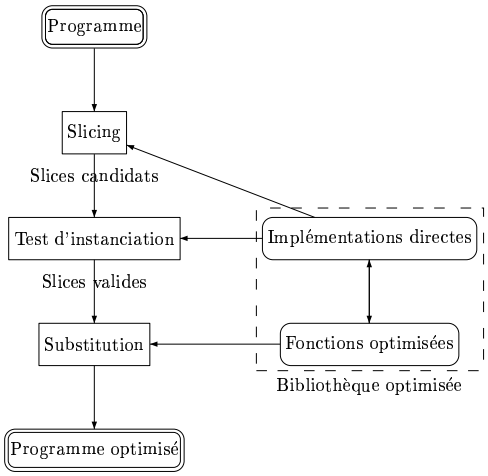



FIG. 4 – Vue d'ensemble de la méthode

bibliothèque fournit une interface d'implémentations directes pour chacunes des fonctions. Notre méthode applique deux grandes phases:

**Phase de détection** Cette phase se fait en deux étapes, et suit la décomposition presentée précédement:

- La méthode de **slicing** trouve les tranches de programme qui calculent le même terme que la fonction à détecter. Il s'agit d'une *sur-approximation*, qui détecte plus de tranches qu'il n'y en a réellement. L'approximation abstrait les bornes des boucles, et les branches choisies dans les conditionnelles. Cette analyse traite l'ensemble du programme, et doit être efficace.
- Le **test d'instanciation** prend les slices détectés par l'étape précédente, et vérifie qu'ils sont bien équivalents à la fonction cherchée. Ce test repose sur

une *sous-approximation* de l'équivalence de Herbrand. Cette approximation est sûre dans le sens où tous les slices retenus sont bien équivalents. Ceci étant, des slices équivalents peuvent être rejetés. Le résultat de cette phase est un ensemble de slices équivalents, et les fonctions de bibliothèque correspondantes.

**Phase de substitution** Une fois les slices équivalents trouvés, il reste à selectionner ceux dont la substitution est *possible* et *intéressante*. Malheureusement, les dépendances de flot entre un slice et les opérations entrelacées peuvent interdire la substitution. Voici un exemple de programme où un daxpy a été découvert. Le slice est constitué des opérations $\langle S, 1\rangle$ à $\langle S, n\rangle$:

```
      do i = 1,n
S     |  y(i) = y(i) + a*x(i)
      |  x(i+1) = 2*y(i)
      enddo
```

Puisque le slice et les opérations entrelacées dépendent l'un de l'autre, on ne peut pas *séparer* le slice du reste du programme, pour le remplacer par un appel à daxpy. Une fois les slices séparables trouvés, la sélection du bon ensemble de substitution se fait par un système de notes. Deux autres solutions expérimentales basées sur du *benchmarking* sont également proposées, l'une décrivant exhaustivement l'espace des substitutions, et l'autre testant les substitutions en suivant une approche gloutonne.

# IV Slicing

Pour simplifier la présentation, cette section présente une version allégée du slicing dans laquelle les fonctions libres ne sont pas gérées. On trouvera une description compléte de la méthode de slicing dans le chapitre 5.

La figure 5 présente un pattern (a) à trouver dans un programme (b). D'après la définition de l'équivalence de Herbrand, un tel problème revient à trouver les occurrences du terme calculé par le pattern (c) dans le terme du programme (d). On a ici un match en prenant les noeuds colorés commes valeurs pour les inputs du pattern.

Une première étape est d'associer au pattern et au programme un *automate d'arbre* permettant de parcourir facilement les termes calculés. Déroulant complétement le pattern, on obtient la séquence d'*opérations* donnée figure 6.(a). Chacune de ces opérations permet de calculer une partie du terme du pattern, comme montré en (b).

Par conséquent, un automate d'arbre reconnaissant *exactement* le terme calculé par le pattern est:

$$\begin{array}{lcl} I(0) & \rightarrow & \langle S_1, \rangle \\ +(\langle S_1, \rangle, I(1)) & \rightarrow & \langle S_2, 1\rangle \\ +(\langle S_2, 1\rangle, I(2)) & \rightarrow & \langle S_2, 2\rangle \\ \ldots & & \\ +(\langle S_2, n-1\rangle, I(n)) & \rightarrow & \langle S_2, n\rangle \end{array}$$

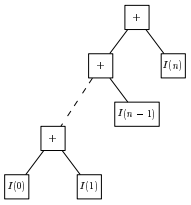
Pour permettre aux inputs du pattern $I(i)$ de prendre les valeurs représentées par les noeuds colorés sur la figure 5.(d), on ajoute les transitions suivantes (dites *de grignotage*)

```
S1  s = I(0)
    do i = 1,n
S2  | s = s + I(i)
    enddo
```

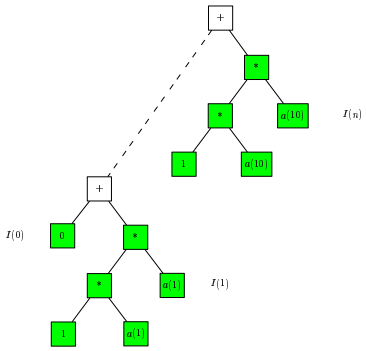
(a) Pattern

```
T1  s = 0
    do i = 1,10
T2  | p = 1
    | do j = 1,2
T3  | | p = p*a(i)
    | enddo
T4  | s = s + p
    enddo
```

(b) Programme



(c) Terme du pattern



(d) Terme du programme

FIG. 5 – Un problème (simplifié) de reconnaissance de template

pour chaque cellule $I(i)$:

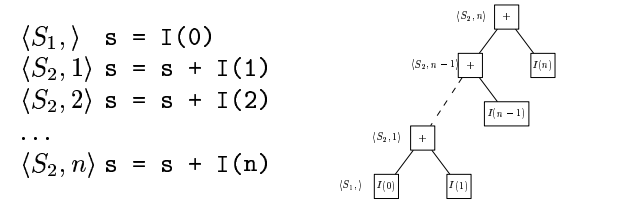
$$\begin{array}{lcll} a(k) & \rightarrow & I(i) & \text{pour chaque } k \\ 0 & \rightarrow & I(i) & \\ 1 & \rightarrow & I(i) & \\ +(I(i), I(i)) & \rightarrow & I(i) & \\ *(I(i), I(i)) & \rightarrow & I(i) & \end{array}$$

De la même manière, on associe au programme un automate d'arbre $\mathcal{A}_P$:

$$\begin{array}{lcl} 0 & \rightarrow & \langle T_1, \rangle \\ 1 & \rightarrow & \langle T_2, 1 \rangle \\ *(\langle T_2, 1 \rangle, a(1)) & \rightarrow & \langle T_3, 1, 1 \rangle \\ *(\langle T_3, 1, 1 \rangle, a(2)) & \rightarrow & \langle T_3, 1, 2 \rangle \\ +(\langle T_1, \rangle, (\langle T_3, 1, 2 \rangle) & \rightarrow & \langle T_4, 1 \rangle \\ \ldots 1 & \rightarrow & \langle T_2, 1 \rangle \\ *(\langle T_2, 10 \rangle, a(1)) & \rightarrow & \langle T_3, 10, 1 \rangle \\ *(\langle T_3, 10, 1 \rangle, a(2)) & \rightarrow & \langle T_3, 10, 2 \rangle \\ +(\langle T_4, 9 \rangle, \langle T_3, 1, 2 \rangle) & \rightarrow & \langle T_4, 10 \rangle \end{array}$$

Plus généralement, pour chaque opération $\omega$:

$$\omega : \quad s = f(s_1 \ldots s_n)$$




(a) Pattern déroulé (b) Calcul correspondant

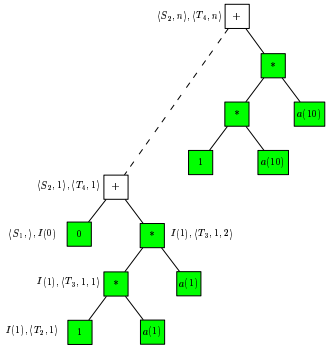
FIG. 6 – Sous-termes calculés par les différentes opérations

On génére la transition:

$$f(\mathsf{RD}_\omega(s_1)\dots\mathsf{RD}_\omega(s_n)) \to \omega$$

où $\mathsf{RD}_\omega(s_i)$ dénote la *définition visible exacte* de la variable $s_i$ dans l'opération $\omega$. Les fonctions libres des templates nécessitent l'ajout de transitions supplémentaires, dont on trouvera le détail dans le chapitre 5.

Il reste maintenant à parcourir *simultanément* les deux automates $\mathcal{A}_T$ and $\mathcal{A}_P$ en calculant leur *produit cartésien* $\mathcal{A}_T \times \mathcal{A}_P$. Voici un exemple de chemin de $\mathcal{A}_T \times \mathcal{A}_P$ menant à une solution:



Noter le *grignotage* réalisé par les transitions sur les inputs $I(i)$. On construit enfin un slice en collectant les opérations du programme sur un chemin décrivant toutes les opérations du pattern. (Des feuilles jusqu'à l'état final de $\mathcal{A}_T$, ici $\langle S_2, n\rangle$).

En fait, les automates d'arbre $\mathcal{A}_T$ et $\mathcal{A}_P$ ont souvent un nombre *paramétrique* de transitions. Par conséquent, on ne peut pas les traiter (ni les représenter) directement. La méthode de slicing effectue alors une *sur-approximation* en travaillant au niveau des *statements* avec des définitions visibles approchées, dont on trouvera une définition dans [1]. Une description de l'approximation est également donnée en section 5.1.1, page 70. Les définitions visibles approchées assurent la finitude de $\mathcal{A}_T$, $\mathcal{A}_P$ et donc de $\mathcal{A}_T \times \mathcal{A}_P$. A partir d'un *statement* $S$ (et non plus d'une opération) :

$$S: \quad s = f(s_1 \dots s_n)$$

On calcule la définition visible approchée $Q_i = \mathsf{RD}_S(s_i)$, qui donne un *ensemble* de statements dont une instance au moins est une définition visible d'une instance de $S$ (approximation conservative). Il reste alors à émettre les transitions :

$$f(S_1 \dots S_n) \to S$$

pour chaque combinaison de statements $S_i \in Q_i$. On peut montrer que l'algorithme obtenu est *conservatif i.e.* tous les slices Herbrand-équivalents sont détectés. Cependant, comme on l'a remarqué précédement, cette approximation amène la méthode à émettre des slices non-équivalents. On a donc besoin d'une méthode *exacte* pour vérifier que les slices trouvés sont bien équivalents au pattern.

# V Test(s) d'instanciation

La méthode de slicing retourne un ensemble de slices candidats qui matchent *potentiellement* le template. Cette section présente brièvement une méthode *exacte*, *le test d'instanciation*, pour filtrer les slices qui matchent le template. Le détail de la méthode pourra être trouvé dans le chapitre 8. Par ailleurs, nous proposons un autre test d'instanciation, dont nous ne parlerons pas ici, et dont on trouvera la description dans le chapitre 7.

Le test d'instanciation suit les étapes de la méthode de slicing décrite dans la section précédente. Une analyse de définition visibles *exacte* est appliquée pour obtenir les automates d'arbre exacts $\mathcal{A}_T$ et $\mathcal{A}_P$. (voir la figure 8.2, page 130). Comme les définition visibles peuvent dépendre des valeurs des compteurs de boucle, ces conditions sont mises sur les transitions des automates d'arbre. Finalement, décider si le fragment de code étudié est une instance du template se ramène à calculer les valeurs des comptes tours obtenus en atteignant l'état final du produit cartésien. Une heuristique efficace [63] permet d'effectuer ce calcul.

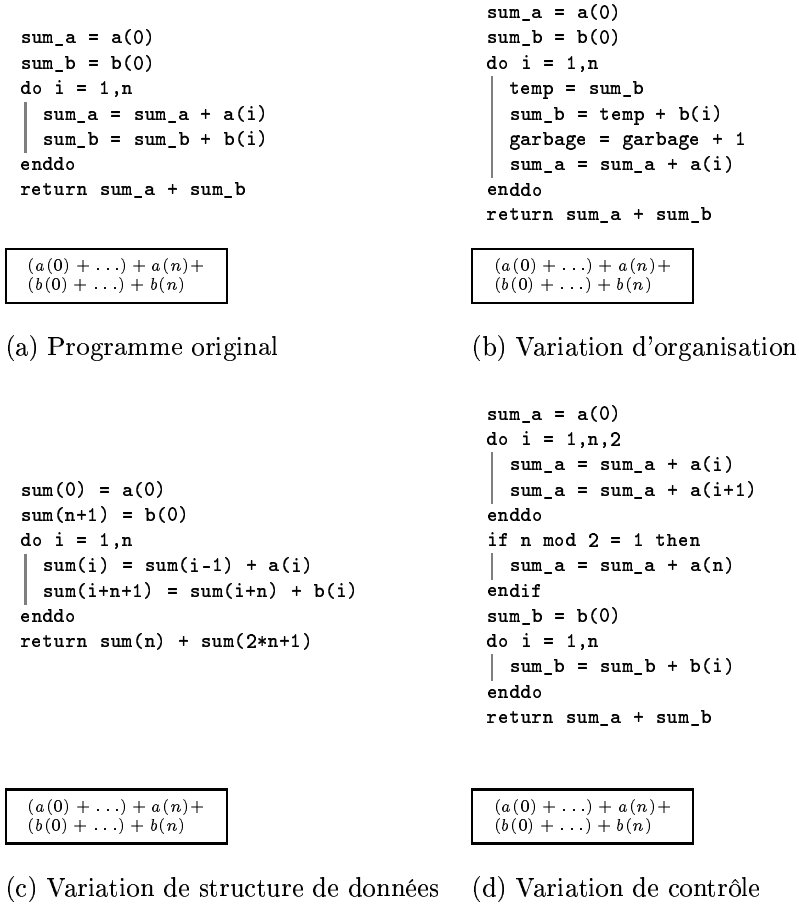
La puissance de cette approche est évaluée selon sa capacité à montrer l'équivalence entre le slice et l'instance appropriée du template, l'un étant une variation de l'autre. Le test supporte les variations venant des transformations de boucles (splitting, fusion, skewing, tiling, unroll,...), des structures de données (scalar expansion, scalar promotion, utilisation de variables temporaires), mais aussi de l'élimination de sous-expressions communes et autres factorisations de calcul. Ceci étant, le test ne gère pas les propriétés sémantiques des opérateurs telles que la commutativité, ou encore l'associativité. La figure 7 montre les différents types de variations supportés par notre test d'instanciation.

# VI Substitution

Une fois les fonctions détectées dans le programme, il reste à les remplacer par l'appel approprié vers la bibliothèque. On doit tout d'abord vérifier que la suppression du slice est valide, puis générer le code, et finalement sélectionner parmis toutes les substitutions celle qui produit la meilleure amélioration de performances.

## VI.1 Validité de la substitution

La plupart du temps, les slices sont entrelacés avec le reste du programme. A cause des dépendences de données, la suppression d'un slice peut changer la sémantique du




(a) Programme original

(b) Variation d'organisation

(c) Variation de structure de données

(d) Variation de contrôle

FIG. 7 – Variations supportées par le test d'instanciation

programme. Plus précisément, deux conditions sont requises pour qu'une substitution soit valide:

- Aucune des variables temporaires du slice ne doit être lue ou écrite dans le reste du programme. Pour préserver la sémantique, la réplication des calculs nécessaires devrait être effectué, ce que nous ne gérons pas dans notre méthode.
- Les dépendences doivent être dirigées *exclusivement* du slice vers le reste du programme, ou bien du reste du programme vers le slice. Ceci interdit notament le cas où l'input du slice est construit dynamiquement à partir de l'output.

La seconde est exprimée par des conditions sur le slice complémentaire: considérons un algorithme reconnu (slice) $A = \{\langle A_1, I_1\rangle \ldots \langle A_n, I_n\rangle\}$, avec pour première opération $\langle A_1, \vec{i}_1\rangle$, et pour dernière opération $\langle A_n, \vec{i}_n\rangle$. Son *slice complémentaire* est l'ensemble des opérations du reste du programme, exécutées entre la première et la dernière opération de $A$:

$$\overline{A} = \{\langle S, \vec{i}\rangle \mid \langle A_1, \vec{i}_1\rangle \prec \langle S, \vec{i}\rangle \prec \langle A_n, \vec{i}_n\rangle \wedge \nexists i \text{ avec } S = A_i\}$$

Une fois $\overline{A}$ calculé, il reste à décider s'il est séparable de $A$, afin d'effectuer la substitution. On dit que $A$ est *séparable* si les dépendences de flot partent *exclusivement* de $A$ vers $\overline{A}$,

ou bien *exclusivement* de $\overline{A}$ vers $A$. Dans le premier cas, on dit que $A$ est *séparable par le haut*, et on peut placer l'appel *avant* $\overline{A}$. L'exemple suivant montre un slice séparable par le haut (à gauche), et sa substitution (à droite).

```
   do i = 1,n
   | if i >= 2 then
A  | | s(i) = y(i-1) + 1
   | endif
A  | y(i) = y(i) + a*x(i)
A  | s(i) = 2*y(i)
   enddo
```

```
   call daxpy(a,x,y)
   do i = 1,n
   | if i >= 2 then
A  | | s(i) = s(i) + 1
   | endif
A  | s(i) = 2*y(i)
   enddo
```

Dans le deuxième cas, symétrique, on dit que $A$ est *séparable par le bas*, et on peut remplacer $A$ par un appel *après* $\overline{A}$. Voici un exemple de slice séparable par le bas (à gauche), et sa substitution (à droite).

```
   do i = 1,n
   | if x(i) >= 0 then
A  | | x(i) = s(i) + 1
   | endif
A  | y(i) = y(i) + a*x(i)
A  | x(i+1) = 2*x(i)
   enddo
```

```
   do i = 1,n
   | if x(i) >= 0 then
A  | | x(i) = s(i) + 1
   | endif
A  | x(i+1) = 2*x(i)
   enddo
   call daxpy(a,x,y)
```

Dans les autres cas, on considère que $A$ et $\overline{A}$ *ne sont pas séparables*, et on ne fait pas de substitution. Voici un exemple de slice non-séparable:

```
   do i = 1,n
A  | y(i) = y(i) + a*x(i)
A  | x(i+1) = 2*y(i)
   enddo
```

On peut remarquer que n'importe quelle analyse de dépendence conservative peut être utilisée. Au pire, un slice séparable sera rejeté.

Une fois les slices séparables isolés, la substitution peut être effectuée en supprimant les opérations de $A$, et en plaçant l'appel avant, ou après $\overline{A}$ selon que le slice soit séparable par haut, ou par le bas. Les opérations de $A$ sont supprimées en ajoutant une garde :

$$S: \quad \texttt{if } i \notin I \texttt{ then } \ldots$$

devant chaque opération $S$, où $I$ designe l'ensemble des vecteurs d'itération de $S$ dans $A$. Deux portions de code doivent également être ajoutées avant, et après le call, pour exprimer les paramètres de la fonction à partir des variables du programme, et pour récupérer le résultat du call dans le point de sortie du slice. Ces portions de code, qu'on appelera *code d'entrée*, et *code de sortie* peuvent malheureusement dégrader les performances du programme, et ainsi minimiser l'impact des substitions. Cet aspect est discuté dans la section suivante.

## VI.2 Sélectionner l'ensemble de substitutions optimal

Intuitivement, remplacer un code par un appel à une fonction optimisée devrait toujours améliorer les performances du programme. Ce n'est malheureusement pas toujours le cas, et ce pour plusieurs raisons:

**Codes d'entrée et de sortie** Comme les inputs de la fonction reconnue n'ont pas toujours leur propre disposition en mémoire, on doit ajouter des codes d'entrée et de sortie, qui sont généralement des memcopy. De tels codes réduisent inévitablement le gain de la substitution.

**Hiérarchie mémoire** Remplacer une portion de code par un appel vers une fonction peut avoir un impact sur la localité, et ainsi réduire fortement les performances.

Pour ces raisons, la sélection parmis toutes les substitutions possibles du sous-ensemble qui réalise les meilleures performances est obtenu de manière empirique. Pour chaque substitution, le code est compilé, exécuté sur un input donné, et le temps d'exécution est mesuré. Comme le nombre de possibilités croit de manière exponentielle avec le nombre de détections, on choisi d'appliquer une méthode gloutonne qui s'exécute en temps linéaire. On examine les substitutions incrémentalement, en gardant une substitution si le temps d'exécution est inférieur au temps d'exécution précédent.

On pourrait diminuer le nombre de mesures à l'aide d'une technique sophistiquée de prédiction de performances [21, 51]. De telles approches ne sont pas étudiées dans cette thèse.

# VII Résultats expérimentaux

La méthode présentée dans cette thèse a été implementée dans un outil appelé `TeMa` (Template Matcher). `TeMa` a étée implémentée en Objective Caml, et comporte plus de 17000 lignes de code. Nous avons implementé notre propre front-end Fortran, qui s'est révélé capable de traiter avec succés plusieurs noyaux des SpecFP 2000 [54] et des Perfect Club [39] benchmarks.

Nous avons appliqué notre méthode à la détection des appels à la bibliothèque BLAS (Basic Linear Algebra Subroutines) [67] dans les noyaux des SpecFP 2000 et des Perfect Club benchmarks. La reconnaissance a été effectuée sur un Pentium 4, 1.6 Ghz avec 256 MB RAM, et les facteurs d'accélération (speed-ups) ont été mesurés sur un Itanium 2, 897 Mhz bi-processeur avec 2 GB RAM. La figure 8 donne les speed-ups de trois fonctions

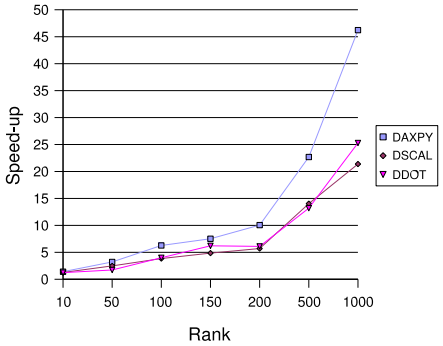



FIG. 8 – Speed-ups de trois fonctions BLAS souvent utilisées

| Noyau | Fonctions analysées | % retenu | Lignes de code | |
|---|---|---|---|---|
| | | | Original | Normalisé |
| 171.swim | 5: (All) | 100 % | 351 | 414 |
| 172.mgrid | 3: `interp psinv resid` | 25 % | 120 | 243 |
| 173.applu | 3: `blts buts rhs` | 14 % | 446 | 1415 |
| 301.apsi | 14: `advt wcont smth smthf csmth horsmt horbc dctdx dctdxd dpdx dftdx ccrank dudz dvdz` | 6 % | 464 | 807 |
| qcd | 2: `project syslop` | 5 % | 115 | 355 |
| mdg | 3: `interf poteng intraf` | 52 % | 646 | 1496 |
| tis | 2: `trfa olda` | 26 % | 127 | 171 |

TABLE 1 – Parties analysées dans les SpecFP et les Perfect Club benchmarks

BLAS usuelles. Basiquement daxpy calcule la combinaison linéaire $\vec{y} \leftarrow a\vec{x}+\vec{y}$ étant donné un scalaire $a$ et deux vecteurs $\vec{x}$ et $\vec{y}$, dscal calcule la dilatation $\vec{x} \leftarrow a\vec{x}$ et ddot calcule le produit scalaire $(\vec{x}|\vec{y})$. Ces courbes ont été obtenues en comparant les temps d'exécution des fonctions BLAS, à ceux d'une implémentation directe, en faisant varier la dimension des vecteurs (rank).

Nous avons restreint notre analyse aux fonctions qui représentent une part importante du temps d'exécution. La table 1 détaille les fonctions analysées dans chaque noyau. La colonne *Normalisé* donne le nombre de lignes après avoir normalisé chaque fonction pour avoir une opération par affectation. Les formes normales de applu et mdg ont une taille importante, puisqu'elles utilisent des affectations avec des formules de taille importante. Notre base de patterns est constituée d'implémentations directes des fonctions BLAS. Plusieurs variations sémantiques usuelles ont été ajoutées pour augmenter le nombre de détections. Les sous-sections suivantes détaillent les résultats expérimentaux obtenus pour chaque étape du processus de reconnaissance.

## VII.1 Slicing

La méthode de slicing à pour but de trouver toutes les instances d'un template dans le programme. Puisqu'elle repose sur une approximation conservative du flot de données, plusieurs candidats non-équivalents peuvent être retournés. Cette section présente une étude expérimentale de la précision du slicing.

La figure 9 donne le nombre de slices candidats detectés pour chaque noyau, et met en évidence les slices non-équivalents (noir), et les slices équivalents (gris et blanc). La méthode de slicing retourne également les instances triviales des patterns. Par exemple, $y = \alpha x + y$ peut être considéré comme un daxpy sur des vecteurs à une dimension. Ceci nous amène à distinguer les slices triviaux (gris), des slices dont la substitution pourrait mener à une amélioration de performance (blanc). Par ailleurs, la figure 10 fait le point sur la proportion de candidats non-équivalents, triviaux, et intéressants trouvés dans chaque noyau.

Il apparaît que 35.9% des candidats ne matchent pas, 56.4% sont des instances cor-

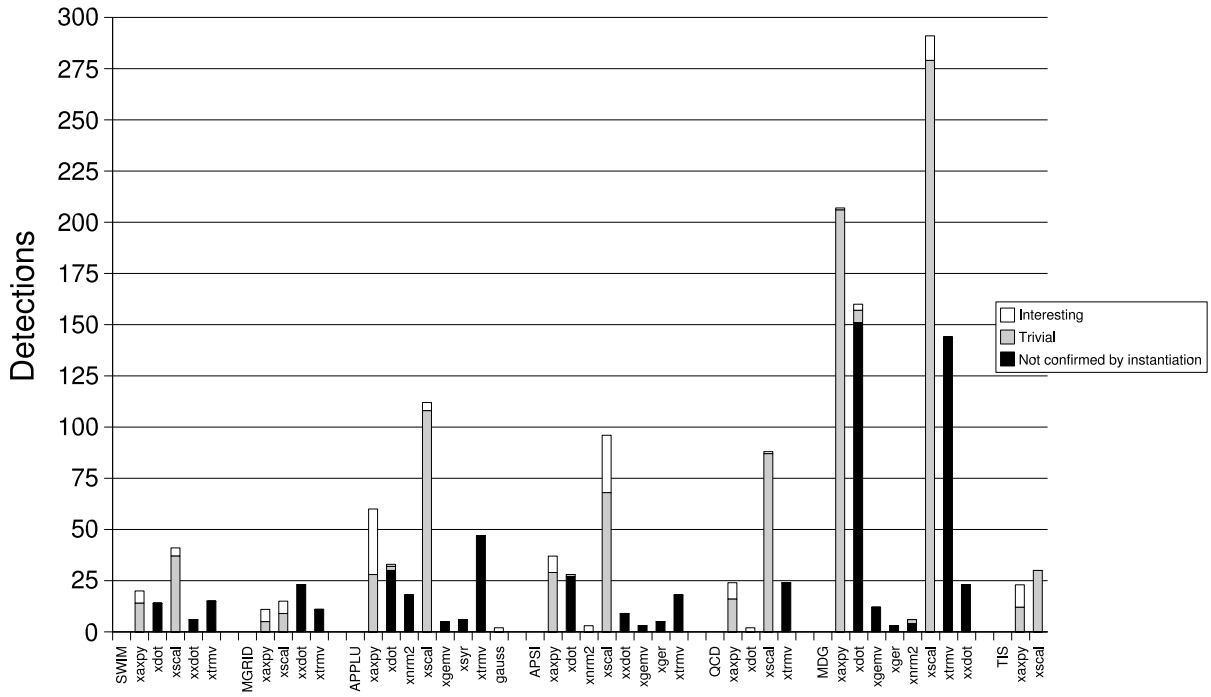



FIG. 9 – Fonctions BLAS détectées dans les noyaux des SpecFP 2000 et des Perfect Club benchmarks

rectes, mais sur des vecteurs de rang trop petit pour avoir des speed-ups, et 7.7% des candidats sont des instances correctes qui peuvent être remplacées par l'appel approprié à BLAS.

Les résultats expérimentaux ont mis en évidence deux causes de mauvaises détections.

**Absence de types** Les définitions visibles approchées traitent les tableaux comme des variables scalaires, et introduisent par conséquent des fausses dépendences entre les statements. Ceci mène par exemple à confondre un daxpy `y(i) = y(i) + a*x(i)` avec un produit scalaire `dot = dot + a(i)*b(i)`.

**Contrôle approché** A cause de l'approximation faite par les définitions visibles approchées, les structures de contrôle ne sont traitées correctement. En effet, considérons le programme suivant (à gauche), et les définitions visibles approchées de chacunes des variables lues (entre crochets [.], à droite).

```
S1  s = a(0)
    do i = 1,n
    |  if i mod 2 = 0 then
S2  |  |  s = s + 1
    |  endif
S3  |  s = s + a(i)
    enddo
S4  r = s
```

```
s = a(0)
do i = 1,n
|  if i mod 2 = 0 then
|  |  s = [S1, S3]1 + 1
|  endif
|  s = [S1, S2, S3]2 + a(i)
enddo
r = [S1, S3]3
```

| Kernel | % Bad | % Trivial | % OK |
|---|---|---|---|
| 171.swim | 35.6 | 53.1 | 10.4 |
| 172.mgrid | 56.7 | 23.3 | 20.0 |
| 173.applu | 37.7 | 49.1 | 13.2 |
| 301.apsi | 31.2 | 49.2 | 19.6 |
| qcd | 17.4 | 74.6 | 8.0 |
| mdg | 39.9 | 58.3 | 1.8 |
| tis | 21.4 | 75.0 | 3.6 |
| SpecFP 2000 | 35.3 | 48.5 | 16.1 |
| Perfect Club | 35.9 | 61.4 | 2.7 |
| **Total** | **35.9** | **56.4** | **7.7** |

FIG. 10 – Répartition des détections

Considérons $[]_2$. Puisqu'on peut choisir indifféremment $S_2$ ou $S_3$, les branches choisies dans les conditionnelles sont ignorées. De la même façon, on peut choisir indifféremment $S_1$ ou bien un des statements $S_2$ ou $S_3$, ce qui amène à ignorer le nombre d'itérations exécuté par les boucles.

Par ailleurs, 56.4% des slices sont des instances correctes, mais dont la substitution n'est pas intéressante, puisqu'ils travaillent sur des vecteurs de rang trop faible pour obtenir un speed-up. La plupart du temps, il s'agit d'expressions arithmétiques simples, comme $y = \alpha x + y$ pour un daxpy. Finalement, 7.7% des slices sont des candidats corrects, et intéressants, dont la substitution peut potentiellement améliorer les performances du programme. Notre algorithme les a toutes découvertes, en particulier les instances «cachées». En effet, la plupart des slices sont entrelacés avec le reste du programme, et fortement destructurés.

## VII.2 Substitution

Une fois les fonctions BLAS découvertes, il reste à sélectionner les substitutions qui amélioreront effectivement les performances du programme. Cette section présente les speed-ups résultant de (a) la recherche exhaustive dans l'espace des substitutions, ainsi de (b) l'heuristique gloutonne présentée dans la section VII.2.

### Recherche exhaustive

Nous avons appliqué une recherche exhaustive pour trouver la meilleure combinaison de daxpy parmis les 12 detectées dans le noyau swim. L'espace de recherche comporte donc $2^{12} = 4096$ combinaisons de substitutions, dont l'exploration à pris 3 heures sur la machine Itanium 2 utilisée pour les expérimentations.

La figure 11 donne les speed-ups obtenus en essayant tous les ensembles de substitutions avec 1, 2, ..., 12 daxpys. Il apparaît que le speed-up optimal est atteint avec deux substitutions. Chaque sous-ensemble de test comporte un palier entre 65 % et 70 %, et une partie entre 10 % et 25 %. Le palier s'explique par la présence des deux substitutions dans chaque combinaison, et la partie basse par leur absence.

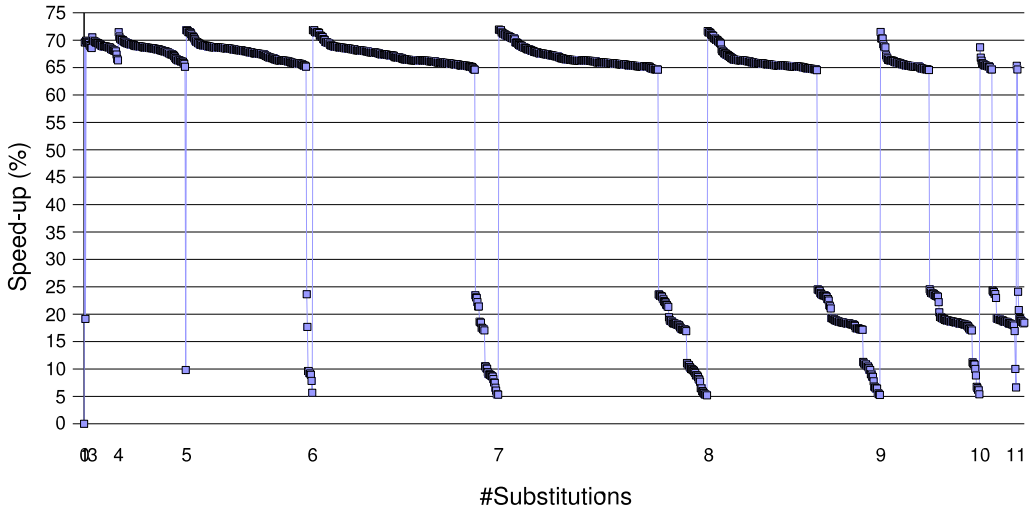



FIG. 11 – Recherche exhaustive

### Recherche gloutonne

| Noyau | daxpy | | scal | | dot | | gauss | | Gain (%) | Speed-up |
|---|---|---|---|---|---|---|---|---|---|---|
| | Testé | Subst. | Testé | Subst. | Testé | Subst. | Testé | Subst. | | |
| 171.swim | 12 | 6 | 4 | 0 | 0 | 0 | 0 | 0 | **66.8** | **3.0** |
| 172.mgrid | 7 | 7 | 6 | 0 | 0 | 0 | 0 | 0 | 3.6 | 1.04 |
| 173.applu | 32 | 0 | 4 | 0 | 1 | 0 | 2 | 0 | 0 | 1 |
| 301.apsi | 8 | 2 | 28 | 0 | 0 | 0 | 0 | 0 | 0.7 | 1.0 |
| qcd | 8 | 0 | 1 | 0 | 2 | 0 | 0 | 0 | 0 | 1 |
| mdg | 0 | 0 | 10 | 3 | 2 | 0 | 0 | 0 | **16.5** | **1.19** |
| tis | 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| **Moyenne** | | | | | | | | | **12.5** | **1.14** |

TABLE 2 – Recherche gloutonne

Nous avons appliqué notre heuristique gloutonne pour trouver les substitutions intéressantes dans les benchmarks analysés. La table 2 montre les speed-ups obtenus après avoir appliqué les substitutions sur les différents noyaux. Il apparaît que notre approche permet d'obtenir un gain moyen de 12.5 %, correspondant à un speed-up de 1.14. Le speed-up important de swim est dû à 6 daxpys travaillant sur des gros vecteurs (1334 éléments), et prenant une part importante du temps d'exécution. Les fonctions détectées dans mgrid travaillent avec des vecteurs de petite taille, ce qui explique le speed-up obtenu. La plupart des slices trouvés dans apsi n'ont pas pu être substitués, faute d'être séparables. De même que pour swim, les speed-ups obtenus pour mdg viennent de la taille importante des tableaux manipulés par les fonctions détectées, et de la part importante prise dans le temps d'exécution. Aucune bonne substitution n'a été trouvée dans les noyaux applu, qcd et tis. En effet, elles nécessitent pour la plupart deux memcopy dans les codes d'entrée et de sortie.

| Noyau | Slicing | Test d'instanciation | Total |
|---|---|---|---|
| 171.swim | 50 s | 17 mn | 17 mn 50 s |
| 172.mgrid | 8 s | 48 s | 56 s |
| 173.applu | 2 mn 55 s | 20 mn | 22 mn 55 s |
| 301.apsi | 2 mn 7 s | 7 mn | 9 mn 7 s |
| qcd | 1 mn 16 s | 8 mn 33 s | 9 mn 49 s |
| mdg | 14 mn 52 s | 49 mn | 1 h 3 mn |
| tis | 3 s | 1 mn 21 s | 1 mn 24 |
| Total | | | **1 h 44 mn** |

TABLE 3 – Temps d'exécution de la méthode de reconnaissance

### VII.3 Temps d'exécution

La table 3 montre le temps d'exécution de l'ensemble du processus de reconnaissance, détaillé étape par étape. Au final, notre méthode aura pris 1 h 44 mn pour optimiser l'ensemble des noyaux présentés. Le lecteur trouvera des tests plus détaillés dans le chapitre 10, avec en particulier une vérification expérimentale de la complexité théorique du slicing et du test d'instanciation.

## VIII Conclusion

Cette section rappelle les contributions principales de la thèse, et présente brièvement les perspectives, dont on pourra trouver une description plus complète et plus détaillée dans le chapitre 11.

### VIII.1 Contributions

Dans cette thèse, nous avons proposé une approche totalement automatique pour reconnaître dans un programme les occurrences des fonctions d'une bibliothèque optimisée, et les substituer par l'appel de fonction approprié lorsque c'est possible et intéressant. Notre méthode est capable de détecter toutes les tranches de programme équivalentes aux fonctions recherchées au sens de l'équivalence de Herbrand ; un sous-ensemble de l'équivalence sémantique qui ne tient pas compte de la sémantique des opérations atomiques. En plus des fonctions, nous sommes également capables de trouver des instances de templates dans un programme. Une telle caractéristique rend possible la reconnaissance de bibliothèques de templates, et la réécriture d'un programme pour utiliser des templates. Une fois les instances trouvées dans le programme, il reste à sélectionner les candidats dont le remplacement par un appel de fonction est possible et améliore effectivement les performances du programme. Nous proposons également un algorithme pour sélectionner les substitutions valides, et pour générer le code avec la substitution. La sélection du bon ensemble de substitution se fait par un système de notes. Deux autres solutions expérimentales basées sur du benchmarking sont proposées, l'une décrivant exhaustivement l'espace des substitutions, et l'autre testant les substitutions en suivant une approche gloutonne.

Notre approche a étée implémentée dans l'outil `TeMa` (Template Matcher). `TeMa` représente plus de 17000 lignes de code C++ et OCaml, et a été appliqué à la détec-

tion des fonctions de la bibliothèque BLAS (Basic Linear Algebra Subroutines) dans les noyaux des benchmarks SpecFP 2000 et Perfect Club. Les résultats expérimentaux montrent un facteur d'accélération moyen de 1.14, avec un pique à 3 pour swim, un noyau de SpecFP 2000.

## VIII.2 Perspectives

### Améliorations

**Variations sémantiques** Notre méthode de reconnaissance est capable de détecter les portions de programme Herbrand-équivalentes à une instance d'un template donné. Malheureusement, nous ne sommes pas capables pour le moment de gérer les variations algorithmiques utilisant des propriétés sémantiques sur les opérateurs atomiques telles que l'associativité, ou encore la commutativité. Une normalisation «quick-and-dirty» du programme peut résoudre la plupart des cas simples. Nous pensons que les règles de l'unification équationnelle peuvent mener à une solution traitant des cas plus généraux.

**Clôtures transitives de relations de Presburger** Notre test d'instanciation repose sur l'exécution symbolique d'un automate à compteur, qui requiert le calcul de l'ensemble des valeurs prises par un compteur sur un état. Comme toujours, le problème vient des cycles, qui mènent à calculer des clotûres transitives de relations de Presburger. Nous utilisons actuellement l'heuristique implémentée dans la bibliothèque Omega. Malheureusement, cette procédure est coûteuse et inadaptée à l'usage intensif qu'en fait `TeMa`. Une partie des clôtures transitives est constituée de cas simples *e.g.* $[i \leftarrow i-1, i>0]*$ pouvant être résolus avec une technique de normalisation *ad-hoc*. Nous pensons que les automates de Presburger [105] sont un point de départ prometteur pour traiter l'autre partie.

### Applications

**Vérification de transformations de programmes** Les optimisations devenant de plus en plus compliquées, il devient de plus en plus dur de certifier un compilateur. Une solution serait de vérifier lors de la compilation que les optimisations n'altèrent pas la sémantique du programme. Notre test d'instanciation est déjà capable de vérifier l'équivalence sémantique avant et après optimisation sur des programmes à contrôle statique, et nous pensons qu'il pourrait être étendu à des cas simples de programmes avec des boucles while et des conditionnelles non-affines. Le travail de Denis Barthou sur l'analyse de flot de données en présence de contraintes non-affines [12] fournit des pistes intéressantes à explorer.

**Model Checking** Le model-checking consiste à vérifier qu'un programme satisfait les contraintes temporelles spécifiées par une formule de logique temporelle LTL (Linear Temporal Logic). Ce problème peut être résolu dans la classe des programmes dont le calcul ne dépend pas des entrées, en associant au programme un automate dit de Büchi $A_P$ qui reconnaît un sous-ensemble représentatif de ses traces d'exécution, et

un automate $A_F$ qui reconnaît toutes les traces qui valident la formule. Il suffit alors de vérifier que le langage $\mathcal{L}(A_P) - \mathcal{L}(A_F)$ est vide. Nous pensons que les templates peuvent encoder un sous ensemble des formules LTL faisant de notre approche une alternative intéressante aux automates de Büchi.

**Réécriture automatique d'un programme dans un langage métier** Les langages métier, plus connus sous la terminologie anglo-saxonne de *domain-specific languages* (DSL) permettent d'exprimer les programmes d'une manière plus abstraite et compacte que les languages impératifs généraux. Les compilateurs de DSL peuvent effectuer des optimisations plus aggressives que les compilateurs traditionnels. Dans une publication récente [6], nous avons étudié la rétro-ingénierie de programmes Fortran vers le langage SPL, un DSL dédié aux applications de traitement du signal. Nous proposons une méthode pour retrouver les parties de programmes exprimables en SPL, et nous pensons que notre méthode peut être appliquée à d'autres DSL. En plus d'améliorer la lisibilité du programme, cela permettrait de tirer profit des optimisations effectuées par le compilateur du DSL.

# Chapter 1

# Introduction

Template recognition consists in finding in a program all the parts that can be rewritten as an instance of a given template. Such a facility has many applications in compiler design and software engineering. One can cite reverse engineering, automatic program comprehension, verification of program transformations, code refactoring, or program optimization. In this thesis, we address the problem of template recognition, and its application to program optimization. We propose a new *source-to-source* optimization that automatically rewrites a program to use a performance library. Given a program, we recognize naive implementations of library functions, and we replace them by a call to the library whenever it is interesting.

## 1.1 Motivations

Most of compiler optimization techniques apply local transformations on the code, replacing sub-optimal fragments with better ones. They are often low-level, and applied without knowing what the code is supposed to do [1] (constant propagation, algebraic simplifications, inlining). Further optimizations try to exploit the hardware in the best manner (automatic parallelization [110], optimal code selection [38], software pipelining [7], register allocation [20], data prefetching [78], data locality [75]). Unfortunately, these optimizations are not enough to produce a definitively optimized code, and leads the programmer to make a choice between:

- **Hand optimizing.** No matter how powerful the compiler optimizations are, they are no substitute for good algorithms. Unfortunately, modern programming language interface often hides critical hardware behaviors such as pipeline and cache effects, which can leads to unpredictable performances. As a consequence, it is difficult for a programmer to find a near-optimal implementation.

- **Optimized routines.** An immediate solution is to look for an existing optimized routine. It may appears in the form of domain-specific languages [107, 25], generative approaches [28], or adaptative libraries [99, 47, 22].

  *Domain-specific languages* (DSL) are programming languages dedicated to specific problems (*e.g.* signal processing [88], device drivers interfaces [92], large data-base

handling [46]). They provide appropriate built-in abstractions and notations allowing to perform high-level aggressive optimizations.

*Generative programming* allows to write a code by using high-level operations, and to generate the corresponding code in the mainstream language. Lex and Yacc are typical examples of generative programming tools, which provides powerful and easy-to-maintain parsers. In the same manner as domain-specific languages, generative approaches provide an abstraction level allowing to perform aggressive optimizations. Even if generative optimizations can produce a good code for individual operations, it may miss optimizations which depend on context.

*Adaptative libraries* provide a bunch of domain-specific routines tuned for the current architecture. At installation time the routines are benchmarked while varying parameters which affects performances (for instance loop structures, tile size and unroll factor). The parameter space is then explored point by point, and the performance of each variant is measured until the best implementation is found. Such an optimization technique is used in many optimized libraries such as ATLAS [99], FFTW [47] and PhiPAC [22]. Despite the cost of the tuning pass, adaptative libraries seems to be a promising approach for optimization. For example, the authors of FFTW report superior performances over all commonly used FFT packages. In addition, the systematic use of libraries ensure portability, reuse and readability of code [59].

*Generic programming* [95, 93, 9, 97, 61] has emerged recently as a new powerful paradigm for simplifying the development of libraries in which a set of algorithms have to be implemented for many data-structures. In order to avoid code explosion, the algorithms do not manipulate concrete data structures directly, but instead operate on abstract interfaces defined for entire equivalence class of data structures. For example, the Matrix Template Library [93] abstracts the sparse storage type of the matrices, and the operations on the underlying ring. Additionally, 13 storage types, and 4 rings corresponding to different precision types have been added, allowing to represent more than one hundred of versions of the same routine! Since generic programming is relatively new, there is a few number of generic libraries for the moment, but we believe that this paradigm will be used widely in the future.

For the moment, the library functions must be called by hand. Research effort has been made to provide a convivial and efficient environment to browse library functions. Several approaches are able to retrieve a library function given pre- and post-conditions (see for example [58], [85] or [44]). However, learning and using a new library remains fastidious, and it is surprising how little the compiler helps the programmer in this task.

A natural solution would be to search naive occurrences of library functions through the program, and replace them by the corresponding call. Paul and Prakash [84] proposes a pattern language with wildcards on syntactics entities which allows to find patterns with specific sequences and imbrications of control structures. Despite a low complexity, their approach is purely syntactic and cannot handle many program variations. Wills [102] represents programs by their dependence graph, and encodes the knowledge about functions to recover by using a graph-grammar. The recognition is then achieved by parsing the program's dependence graph according to graph-grammar rules. This approach

is a pure bottom-up code-driven analysis based on exact graph matching. Unfortunately, this approach is expensive, and the variations recognized directly depends on the grammar rules. Additionally, the base of grammar rules seems to be difficult to maintain, and oblige the user to add manually his own rules whenever he want to look for a new pattern. Metzger [76] provides a complete framework to recognize algorithms in a program, and to substitute them by an optimized call. His equivalence test is based on a normalization of abstract syntax trees, and cannot handle many program variations.

In this thesis, we propose a fully automatic approach to recognize naive occurrences of library functions, and to substitute them, whenever it is possible and interesting, by a call to the optimized library. In addition to functions, we are also able to find *template instances* in the source code, providing the relevant template variables for each match. Such a characteristic enable the rewriting of a program to use generic libraries.

## 1.2 Example of Optimization by Template Recognition

Figure 1.1 provides a template (a) to find in a program (b). The template is a naive implementation of a generic reduction over an array $I$ with respect to the operator $X$. We want to replace the reductions found in the program by a call to a parallel version `par_reduc(`$X$`,I)` provided by an efficient library. This is only possible if $X$ has a special form $X = x \oplus f(y)$ where $\oplus$ is at least associative. How to decide this property has been explained in [89] and will not be discussed further in this thesis. The first step of our approach will recognize *all* the instances of the template in the program. Here we should obtain the two following instances:

- Lines 1 to 8, with $X(x, y) = x+y^5$, $n = 10$, $I(0) = 0$ and $I(k) = a(k)$ for $1 \leq k \leq 10$.
- Lines 3 to 6, with $X(x, y) = x \times y$, $n = 5$, $I(0) = 1$ and $I(k) = a(i)$ for $1 \leq k \leq 5$.

Since the two recognized parts (slices) overlap, one must choose the slice whose substitution will lead to the best performance improvement. Our selection algorithm depends on parameters specified by the library designer. For instance, we could choose the second slice. It remains then to generate the code with the right substitution (see (c)).

```
s = I(0)
do i = 1,n
| s = X(s,I(i))
enddo
return s
```

(a) Template

```
1 s = 0
2 do i = 1,10
3 |  p = 1
4 |  do j = 1,5
5 |  | p = p*a(i)
6 |  enddo
7 |  s = s + p
8 enddo
```

(b) Program

```
s = 0
do i = 1,10
| I(0) = 1
| I(1:5) = a(i)
| p = call par_reduc(λxy.x×y,I)
| s = s + p
enddo
```

(c) Optimized program

FIG. 1.1 – A template recognition problem

## 1.3 Difficulties in Template Recognition

### 1.3.1 Common Program Variations

The main difficulty in algorithm recognition comes from the various way to implement a given algorithm. Wills [102] provides a classification of all possible types of variations that we summarize in figure 1.2.

**Organization variations**

Any permutation of independent statements and introduction of temporary variables. The following example give an organization variation with legal permutations (LP), garbage code (GC) and temporaries (T):

```
s = a(0)                    s = a(0)
c = 0                       c = 0
do i = 1, n             GC  garbage = 0
| s = s + a(i)              do i = 1, n
| c = c + 1             LP  | c = c + 1
enddo                   T   | temp = a(i)
return s + c                | do j = 1, p
                        GC  | | garbage = garbage + 1
                            | enddo
                            | s = s + temp
                        GC  | garbage = garbage + a(i)
                            enddo
                            return s + c
```

**Control variations**

Any control transformation as if-conversion, dead-code suppression and loop transformations as peeling, splitting, skewing, etc. The following example give a control variation with a simple peeling:

```
s = a(0)                    s = a(0)
do i = 1,n                  s = s + a(1)
| s = s + a(i)              do i = 2,n-1
enddo                       | s = s + a(i)
return s                    enddo
                            s = s + a(n)
                            return s
```

**Data structure variations**

The same computation with a different data structure. The following example give a data structure variation with arrays and non-recursive structures:

```
s(0) = a(0)                 s.sum1 = a(0)
do i = 1, 2*n               do i = 1, n
| s(i) = s(i-1) + a(i)      | s.sum1 = s.sum1 + a(i)
enddo                       enddo
return s(2*n)               s.sum2 = a(n+1)
                            do i = n+2, 2*n
                            | s.sum2 = s.sum2 + a(i)
                            enddo
                            return s.sum1 + s.sum2
```

**Semantics variations**

Any transformation which makes hypothesis on operator properties. For example, the following transformation assumes the commutativity of +:

```
s = a(0)                    s = a(0)
do i = 1,n                  do i = 1,n
| s = s + a(i)              | s = s + a(n-i+1)
enddo                       enddo
return s                    return s
```

FIG. 1.2 – Typology of program variations

In theory, algorithm recognition is more difficult than semantic equivalence between programs, which is undecidable. Most existing approaches for algorithm recognition are based on syntactic matching, and can only cope with some organization variations. Unfortunately, the occurrences of an algorithm are often data-structure or control variations. This thesis proposes an heuristic able to recognize in most cases all possible kind of program variations, except variations that take semantic properties of operators into account.

### 1.3.2 Performance Prediction

Performance prediction aims here to quantify the impact of a substitution on program performances. In the same manner as semantic equivalence, the problem to decide whether an transformation improve the performance is undecidable [79]. Nevertheless, several

approaches provides accurate performance prediction models, such as Fahringer's [40], Ghosh's [51] or Padua's approches [21]. Such approaches provides parameters quantifying the performances of the program, such as the execution time, or the number of cache misses ; they are often based on profiling. Accurate performance prediction models are *not* in the scope of this thesis.

## 1.4 Contributions

### 1.4.1 Template Recognition

In this thesis, we propose a fully automatic approach to recognize occurrences of library functions in a program, and to substitute them, whenever it is possible and interesting, by a call to the library. Our approach is able to recognize all the program slices computing the same mathematical formula than the searched function. This allow to cope with organization, data-structure and control variations, and more generally with any program transformation which does not take operators properties (associativity, commutativity, etc.) into account. In addition to functions descriptions, we are also able to find *template instances* in the source code. Such a characteristic enable the recognition of template libraries, and the rewriting of a program to use templates.

Our method is divided into three steps. One apply a first pass to get the program parts which *possibly* matches the template. This step, called *slicing* is shown to be *conservative*, we do not miss any legitimate instance (conservativity). Moreover, the amount of false positive remains reasonnably low. A second pass check whether the slices are effectively instances of the template by using an *instantiation test*. We provide two instantiations tests based on different theoretical frameworks. The first one works in the context of *unification theory*, and adapts Huet and Lang's matching procedure [56]. The other one represents the template and the program to match by *tree-automata*, and achieves the matching by using a cartesian product. We show that the two algorithms differs by their complexity and their detection capabilities. The first instantiation test has a relatively low complexity ; it can only handle template variables $X$ defined by a finite arithmetic expression, The second instantiation test is able to detect template variables involving a `do` loop *e.g.* the first unifier in the example above, at the cost of a greater complexity. Finally, we select the interesting substitution, and we generate the corresponding program, as stated in the next section.

### 1.4.2 Substitution

Once the proper instances are found within the program, it remains to select the slices whose replacement by a library call is possible, and interesting. We propose a complete algorithmic framework to select all valid substitutions, and to generate the corresponding code. A *separability test* has been designed to decide whether a slice can be separate from the remaining of the program. To select a good substitution set, we propose a preliminary solution based on a system of marks, which provides correct results in practice.

## 1.5 Outline

This thesis is structured as follows.

**Chapter 2** states formally the problem of template recognition, and justifies the decomposition slicing/exact instantiation test which is addressed in this thesis.

**Chapter 3** presents and discusses several existing approaches for algorithm recognition, motivated by program understanding and program optimization.

**Chapter 4** provides an overview of our recognition framework, and describes accurately the recognition process on a simple example.

**Chapter 5** presents the first step of our recognition framework, a slicing method which aims to find the program slices which *possibly* matches the template. A formalization of the template recognition problem using tree automata is also presented, and is used in Chapter 8.

**Chapter 6** provides a detailed explanation of the exact equivalence test due to Barthou et al [13], which inspires our two instantiation tests described in Chapters 7 and 8.

**Chapter 7** expresses the template matching problem as a second-order matching problem, and proposes an instantiation test extending the equivalence test described in Chapter 6 by using rules of Huet and Lang's algorithm [56]. The obtained algorithm is able to handle many template matching problems, with a linear complexity.

**Chapter 8** presents another instantiation test based on tree-automata. The resulting algorithm is an exact version of the slicing algorithm, which is able to handle more template matching problems, but at a larger computation cost.

**Chapter 9** describes a method to substitute the proper slices by a call to a library function, whenever it is possible and interesting. The selection of the substitution set, and the generation of the code with the substitutions are addressed.

**Chapter 10** presents the implementation of these algorithms in a tool called `TeMa`, and provides the experimental results obtained while matching the BLAS level 1 and 2 functions [67] in the kernels of the SpecFP 2000 [54] and the Perfect Club [39] benchmarks. Particularly, $\times 3$ speed-up was obtained on the swim kernel.

**Chapter 11** concludes this thesis by discussing the value of our work and possible future developments.

# Chapter 2

# Definitions and Notations

In this chapter, we provide a formal description of the template recognition problem. We also present several notions which are used in this dissertation. After defining what a template is, we present two equivalence relations over programs, and we define formally the problem of template recognition. We finally present and justify a decomposition of this problem into two sub-problems, decomposition which is used in this thesis.

## 2.1 Program Model

The optimization framework presented in this thesis addresses general programs. However, the program parts to recognize must be *static control programs*.

**Definition 2.1 (Static control program).** *A* static control program *satisfies the following conditions:*

- *Data structures are scalars and arrays.*
- *Control structures are sequences,* `if` *conditions and* `do` *loops.*
- `if` *conditions,* `do` *loop bounds and index functions of arrays are* affine *expressions of surrounding loop counters, and of integer parameters.*
- *Basic statements are assignments.*

Most linear algebra routines are static control programs: Gaussian elimination, *LU* decomposition, Cholesky factorization, etc. Figure 2.1.(a) gives the example of the product of polynomials. Notice the affine array index function $(i, j) \mapsto i + j$ of `c`, and the structure parameters `n` and `m`, which corresponds here to the degree of polynomials `a` and `b`.

An *iteration vector* of a statement $S$ is constituted of the values of the counters of the loops surrounding $S$. The set of iteration vectors of $S$ during the execution is called the *iteration domain* of $S$. Iteration domains of static control programs are $\mathbb{Z}$-polytopes, and can be computed at compilation time. Under these restrictions, several problems can be decided, such as exact instancewise dataflow analysis [43].

```
   do i = 0,n+m
S1 | c(i) = 0
   enddo
   do i = 0,n
   | do j = 0,m
S2 | | c(i+j) = c(i+j) + a(i)*b(j)
   | enddo
   enddo
```

(a) Product of polynomials

| | |
|---|---|
| $\langle S_1, 0\rangle$ | `c(0) = 0` |
| $\langle S_1, 1\rangle$ | `c(1) = 0` |
| $\langle S_1, 2\rangle$ | `c(2) = 0` |
| $\langle S_2, 0, 0\rangle$ | `c(0) = c(0) + a(0)*b(0)` |
| $\langle S_2, 0, 1\rangle$ | `c(1) = c(1) + a(0)*b(1)` |
| $\langle S_2, 1, 0\rangle$ | `c(1) = c(1) + a(1)*b(0)` |
| $\langle S_2, 1, 1\rangle$ | `c(2) = c(2) + a(1)*b(1)` |

(b) Execution trace for $n = m = 1$

FIG. 2.1 – Product of polynomials

An *operation* is an instance of a statement $S$ during the execution *i.e.* an element of the execution trace. It is usually denoted by $\langle S, \vec{i}\rangle$, where $\vec{i}$ denotes an iteration vector of $S$. For example, $\langle S_2, n, m\rangle$ is the last operation of the polynomial product (b).

## 2.2 Templates

This thesis investigates the recognition of *template* instances in programs. In a way, static control programs are already templates, since they depend on integer parameters (cf. `n` and `m` in the example 2.1). We enhance the level of genericity by allowing to parametrize static control programs with pure functions, as stated in the following definition:

**Definition 2.2 (Template).** *We define a* template *as a static control program parametrized by a tuple of functions* $(X_1 \dots X_n)$*, that we call* free functions *or* template variables. *Each free function* $X_i$ *is assumed to be* pure *(no side effect), and must be involved in the right hand side of an assignement:*

$$\dots \texttt{ = } \dots X_i(\dots) \dots$$

The last point forbids template variables in conditions, and bounds of `do` loops. This definition is general, and does not restrict the order of template variables. In this thesis, we will assume that template variables are *second-order at most.* For instance, here is the template of a reduction parametrized by the template variable $X$, and the structure parameter `n`:

```
s = a(0)
do i = 1,n
| s = X(s,a(i))
enddo
return s
```

The template variable $X$ can be substituted by any piece of code which computes a value from `s` and `a(i)`, with no side effect.

An *instance* of a template $T$ is the program obtained by substituting the templates variables $\vec{X} = (X_1 \dots X_n)$, the parameters $\vec{n} = (n_1 \dots n_p)$ and the template inputs $\vec{I} = (I_1 \dots I_n)$ by specific values. It is denoted by $T[\vec{X}, \vec{n}, \vec{I}]$. For instance, we can have:

$$T\left[\begin{array}{l}\lambda x.\lambda y.\\ \\ \\ \\ \end{array}\begin{array}{l}\texttt{p = 1}\\ \texttt{do j = 1,5}\\ \texttt{| p = p * } y\\ \texttt{enddo}\\ \texttt{return } x \texttt{ + p}\end{array}\quad,\ \texttt{n} = 10,\ a = \left[\begin{array}{c}1\\ B(1)\\ \vdots\\ B(n)\end{array}\right]\right] = \begin{array}{l}\texttt{s = 1}\\ \texttt{do i = 1,10}\\ \texttt{| p = 1}\\ \texttt{| do j = 1,5}\\ \texttt{| | p = p * B(i)}\\ \texttt{| enddo}\\ \texttt{| s = s + p}\\ \texttt{enddo}\\ \texttt{return s}\end{array}$$

Generally, $X$ is not restricted to static control programs, it can be defined as any piece of code with `do while` loops, non-affine expressions in conditionals, loops bounds and array index functions, etc. However, we will assume in this thesis that $X$ is defined by a static control program.

In a more theoretical manner, a template can be defined as a term $t$ of $\lambda$-calculus with free variables $X_1 \dots X_n$. Writing $\hat{t} = \lambda X_1 \dots X_n.t$ the closure of $t$, the instance of $t$ *w.r.t.* closed terms $t_1 \dots t_n$ can be defined simply by the application $\hat{t}\ t_1 \dots t_n$.

## 2.3 Data Dependences

*Data dependences* were introduced by Bernstein [16] to represent ordering constraints among operations in a program. More often, they are less restrictive than the sequential execution order, and thus provide the compiler with more flexibility to rearrange the code. Three kinds of dependences are usually defined:

- *Flow dependence* occurs when an operation writing a variable is followed in the execution order by an operation reading the same variable.
- *Anti-dependence* occurs when an operation reading a variable is followed in the execution order by an operation writing the same variable.
- *Output dependence* occurs between two operations writing the same variable.

Consider the following program:

$$\begin{array}{ll} S_1 & \texttt{sum = a(0)}\\ & \texttt{do i = 1,n}\\ S_2 & \texttt{| sum = sum + a(i)}\\ & \texttt{enddo}\end{array}$$

Flow dependences occurs from operations $\langle S_1, \rangle$ to $\langle S_2, 1\rangle$, and from operations $\langle S_2, i\rangle$ to $\langle S_2, i+1\rangle$ when $1 \leq i \leq n-1$ (write `sum` / read `sum`). Anti-dependences occurs from operations $\langle S_2, i\rangle$ to $\langle S_2, i+1\rangle$, when $1 \leq i \leq n-1$ (read `sum` / write `sum`). Finally, output dependences occurs from operations $\langle S_1, \rangle$ to $\langle S_2, 1\rangle$ and from operations $\langle S_2, i\rangle$ to $\langle S_2, i+1\rangle$, when $1 \leq i \leq n-1$ (write `sum` / write `sum`).

Although checking whether a dependence exists between two operations is generally undecidable, litterature provides many *may*-approximations [110, 1] which work on general programs ; and several exact solutions on particular kinds of programs [43, 72, 74, 87]. The algorithms described in this thesis use approximate and exact flow dependences, that we describe thereafter in a more formal manner.

As stated in Section 2.1, the execution of a program on the input $I$ can be seen as a (possibly infinite) sequence of operations $\omega_1^I;\ \omega_2^I;\ \ldots;\ \omega_n^I;\ \ldots$. We define the *execution order* $\prec_I$ between operations as $\omega_i^I \prec_I \omega_j^I \Leftrightarrow i < j$.

$\prec_I$ obviously defines a total order over the operations $\omega_i^I$. We now define the notion of reaching definition, which will be used along this thesis.

**Definition 2.3 (Reaching definition).** *Consider a program executed on an input $I$, and an operation $\omega$ reading a variable $v$. The* reaching definition *the variable $v$ is the last operation $\tau$ executed before $\omega$ that writes $v$. It is usually denoted* $\mathsf{RD}_\omega^I(v)$*:*

$$\mathsf{RD}_\omega^I(v) = \max_{\prec_I}\{\tau \mid \tau \prec_I \omega \text{ and } \tau \text{ writes } v\}$$

Hopefully, the operations executed by static control programs do not depend on the input $I$. Hence, we can simplify the notation, and write $\mathsf{RD}_\omega(v)$ for $\mathsf{RD}_\omega^I(v)$. In the above example, the reaching definition of `sum` read by $\langle S_2, \vec{i}\rangle$ is:

$$\mathsf{RD}_{\langle S_2,\vec{i}\rangle}(\texttt{sum}) = \begin{cases} i = 1: & \langle S_1, \rangle \\ 2 \leq i \leq n: & \langle S_2, i-1\rangle \end{cases}$$

Unfortunately, reaching definitions computation is undecidable on *general* programs, since the iteration domains of statements, and the conditional expressions values are generally unknown at compile time. Consider any program built from $P$ and $Q \in \mathbb{Z}[X]$ in the following manner:

```
do x1 = ...
|  ...
|  do xn = ...
|  |  a(P(x)) = ...
|  |  ... = a(Q(x))
```

To find the reaching definition of $a(Q(\vec{x}))$, we have to check whether the diophantine equation $P(x) = Q(x)$ has a solution over $[\![1, n]\!]$, with a parametric size $n$. This problem is known as the tenth Hilbert's problem, which has been proven undecidable [73]. Several reasearches try nevertheless to provide an exact solution for particular kinds of programs. Particularly, Feautrier [43] gives an exact solution restricted to static control programs. His algorithm, which is used in our framework, is detailed in Chapter 6.

We finally introduce an object widely used in program analysis, the well-known *dependence graph*. Whereas the common definition deals with statements, and allows to represent *may*-approximations of data-dependences, the definition proposed here is slightly different, and propose to deal with *exact flow dependences* at operation level.

**Definition 2.4 (Dependence graph).** *Consider a program $P$ executed on an input $I$, and denote by $\omega_1^I \ldots \omega_n^I \ldots$ the operations executed. The dependence graph of $P$* w.r.t. *$I$ is a directed graph $\mathcal{G}(P, I)$ whose:*

- Nodes *are the operations $\omega_i^I$*
- Edges *express the exact flow dependences:*

$$\omega^I \to \tau^I \iff \exists v \text{ s.t. } \mathsf{RD}_{\tau^I}^I(v) = \omega^I$$

As for reaching definitions, one can write $\mathcal{G}(P)$ for $\mathcal{G}(P, I)$ when the operations executed by $P$ do not depend on $I$. Once again, this is the case for static control programs. Since an operation $\omega$ cannot depend on the next operations $\tau : \omega \prec_I \tau$, the dependence graph is a DAG. This important property is expressed in the following proposition.

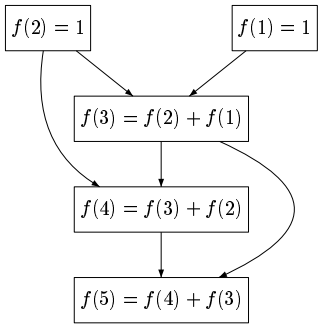
**Proposition 2.1.** *Given a program $P$ and an input $I$, $\mathcal{G}(P, I)$ is a DAG.*

*Proof.* Assume $\mathcal{G}(P, I)$ is *not* a DAG and consider a cycle $\omega_1 \rightarrow \ldots \rightarrow \omega_n \rightarrow \omega_1$. Since $\omega_n \rightarrow \omega_1$, $\exists v$ s.t. $\mathsf{RD}^I_{\omega_1}(v) = \omega_n$. As a consequence, $\omega_n \prec_I \omega_1$. By transitivity, we have $\omega_1 \prec_I \omega_1$, which contradicts the definition of $\prec_I$. □

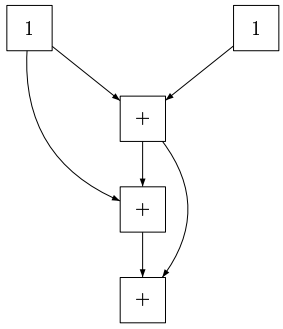
Basically, $\mathcal{G}(P, I)$ can be seen as a compact representation of the the expression computed by $P$ while executing on $I$. Consider indeed the program given in figure 2.2 (a). The program computes the well-known Fibonnacci sequence by using dynamic programming. Since the program has no inputs, the operations executed are always the same, leading to the dependence graph $\mathcal{G}(P)$ given in (b). Labelling each node of $\mathcal{G}(P)$ by the operator used in the corresponding operation and handling the constant 1 as a 0-ary operation, we obtain the DAG given in (c), which represents exactly the expression computed by the program.

```
do i = 1,5
| if i = 1 .or. i = 2 then
| | f(i) = 1
| else
| | f(i) = f(i-1) + f(i-2)
| endif
enddo
```

(a) Original program




(b) Dependence graph




(c) Expression DAG

FIG. 2.2 – Similarities between dependence graph and expression tree

## 2.4 Program Slicing

Program slicing is a program analysis technique which aims to extract from the program self-contained parts which can be factored out as a function. Program slicing was first introduced by Weiser [98] to help students to debug their code. He defined a *slicing criterion* as a pair $(p, V)$, where $p$ is a program point and $V$ a subset of program variables. A *slice w.r.t.* $(p, V)$ is a subset of program statements that allow to compute the right values of the variables of $V$. Figure 2.3 presents the slice of a program (a) *w.r.t.* the slicing criterion $(S, r)$. The slice found is (b).

Weiser has shown that computing a slice with a minimum number of statements is *undecidable*. However an approximation can be found by computing consecutive sets of

```
      sum = 0
      fact = 1
      e = 0
      do i = 1,n
      |  sum = sum + i
      |  fact = fact*i
      |  e = e + 1/fact
      enddo
S     r = 2*e
```

(a) Original program

```
      fact = 1
      e = 0
      do i = 1,n
      |
      |  fact = fact*i
      |  e = e + 1/fact
      enddo
S     r = 2*e
```

(b) Slice *w.r.t.* $(S, \{r\})$

FIG. 2.3 – An example of slice

indirectly relevant statements, according to data-flow and control-flow dependences. Most often, slices are built by following *approximate* reaching definitions, from the selected statement to the input of the program [98, 70, 48, 90].

This definition of a slice is too imprecise for our purpose since it deals with *statements*, and lead to include systematically all corresponding operations in the slice. We propose thereafter another definition of a slice which deals with *operations*.

**Definition 2.5 (Full slice).** *Consider a program $P$ executed on an input $I$, and denote by $\tau^I$ an executed operation. The* full slice *of the program $P$* w.r.t. *the operation $\tau^I$ is the limit of the sequence $(\Omega_i)_i$ defined as follows:*

$$\begin{array}{ll} \Omega_0 & = \{\tau^I\} \\ \Omega_{i+1} & = \Omega_i \cup \bigcup_{\omega\in\Omega_i} \bigcup_{v\in\mathsf{rhs}(\omega)} \mathsf{RD}_\omega(v) \end{array}$$

*where $\mathsf{rhs}(\omega)$ denotes the set of variables read by $\omega$. We write $\widehat{\mathcal{S}}^I_P(\tau^I) = \lim_{i\to\infty} \Omega_i$.*

As for the dependence graphs and the reaching definitions, when the operations executed by $P$ do not depend on $I$, we write $\widehat{\mathcal{S}}_P(\tau^I)$ for $\widehat{\mathcal{S}}^I_P(\tau^I)$. Basically, a *full* slice is exactly the set of operations needed to compute the variable assigned by $\tau^I$. It is built by following the *exact* reaching definitions of the variables readen by $\tau^I$, up to the leaves of $\mathcal{G}(P, I)$.

With this definition, a *full slice* captures the complete program part required to compute a variable. A *slice* is slightly different since it can start from « parameters » values written by several operations. Basically, a slice is a program part that can be factored (or replaced) by a pure function.

**Definition 2.6 (Slice).** *Taking the notations of definition 2.5, and denoting by $\iota = (\iota^I_1 \dots \iota^I_n)$ a tuple of operations where each $\iota^I_i$ is an ancestor of $\tau^I$ in $\mathcal{G}(P, I)$, the* slice *of $P$* w.r.t. *output $\tau^I$ and input $\iota$ is the limit of the sequence $(\widetilde{\Omega}_i)_i$ defined by:*

$$\begin{array}{ll} \widetilde{\Omega}_0 & = \{\tau^I\} \\ \widetilde{\Omega}_{i+1} & = \widetilde{\Omega}_i \cup \bigcup_{\omega\in\widetilde{\Omega}_i} \bigcup_{v\in\widetilde{\mathsf{rhs}}(\omega)} \mathsf{RD}_\omega(v) \end{array}$$

*Where $\widetilde{\mathsf{rhs}}(\omega) = \emptyset$ if $\omega$ is a $\iota^I_i$, and $\widetilde{\mathsf{rhs}}(\omega) = \mathsf{rhs}(\omega)$ otherwise. We write $\mathcal{S}^I_P(\iota, \tau^I) = \lim_{i\to\infty} \widetilde{\Omega}_i$.*

Once again, we write $\mathcal{S}_P(\iota, \tau^I)$ for $\mathcal{S}_P^I(\iota, \tau^I)$ when $I$ is not needed. Particularly, this is the case for the static control programs.

One can remark that slices of static control programs have a parametric size at most. Indeed, consider a static control program with statements $S_1 \dots S_n$, and corresponding iteration domains $D_1 \dots D_n$. The total number of operations executed is $|D_1| + \dots + |D_n|$ which is finite since the $D_i$ are $\mathbb{Z}$-polytopes. However, their size can depend on an integer parameter.

## 2.5 Program Equivalence

Algorithm recognition can be defined as the process of finding all the program slices *equivalent* to a reference algorithm. This section makes this notion of equivalence between programs precise. We first define the semantic equivalence, then we present the Herbrand-equivalence, a weaker equivalence that is addressed in this thesis.

*Semantics* [103] is a theoretical framework to specify the behaviour of a program. One traditionally distinguish three kind of semantics:

**Operational semantics** is concerned with *how* to execute programs. It describes how the variables are modified during the execution of the program. Operational semantics is usually described by a transition system specifying of to derive a state $\langle S, \sigma \rangle$ to a state $\sigma'$ where $S$ is a statement, and $\sigma$ and $\sigma'$ are mappings specifying variables values (*memory states*).

**Denotational semantics** associates to a program the *function* that it computes, also called the *function denoted*. Denotational semantics provides a set of *semantics functions* that maps each important syntactic category to a function of memory states. The semantics functions are defined by compositionality. More accurately, when a syntactic category $S$ involves syntactic categories $S_1 \dots S_n$, the function denoted by an object $S$ is defined by using the functions denoted by the objects $S_i$.

**Axiomatic semantics** specifies the behaviour of a program with assertions involving inputs and outputs. Assertions are expressed in Hoare logic [55], and consists in triplets $[P]\ S\ [Q]$ where $S$ is a statement, and $P$ and $Q$ are logical formula expressing relations between program variables. $P$ describes the memory state *before* the execution of $S$, whereas $Q$ describes the memory state obtained *after* the execution of $S$. Axiomatic semantics defines the relation between $P$ and $Q$ by induction on the construction rules of the programming language. It is often presented as a deduction system.

With *abstract interpretation* [26], Cousot and Cousot provides a general framework formalizing the construction of approximate or *abstract* semantics. Unlike usual or *concrete* semantics, abstract semantics describe the behaviour of the program over a set of approximate or *abstract values*. For instance, integers can be approximated by intervals. The abstract semantics are ordered in a *lattice* with respect to the precision relation. When the approximation is rough enough, abstract semantics can be computed, and allow to (semi-)decide some non-trivial assertions on programs. In this manner, abstract semantics allow to compute *must*-approximations.

We provide now the *concrete* denotational semantics for the significant fragment of Fortran 77 handled in this thesis. For presentation reasons, we will describe here only a small subset of the language, including assignments, sequence, conditionals and loops (`do` and `do while`). However, our optimization framework is able to handle most of Fortran 77 constructions.

The definition of denotational semantics requires the syntactic categories described in the following table. The second column provides the variables that will be used to range each category.

| | | |
|---|---|---|
| $\mathsf{Num}$ | $c_n$ | numeric values (integers, reals) |
| $\mathsf{Bool}$ | $c_b$ | boolean values (true, false) |
| $\mathsf{Ref}$ | $v$ | references to variables (scalars and arrays) |
| $\mathsf{Fun}_k$ | $f_k$ | $k$-ary functions |
| $\mathsf{Expr}_{\mathsf{Num}}$ | $e$ | numeric expressions |
| $\mathsf{Expr}_{\mathsf{Bool}}$ | $e$ | boolean expressions |
| $\mathsf{Stmt}$ | $S$ | program statement |

Interpreting $\mathsf{Bool}$ by $\mathbb{B} = \{\mathbf{True}, \mathbf{False}\}$ where $\mathbf{True} = \lambda xy.x$ and $\mathbf{False} = \lambda xy.y$ ; and $\mathsf{Num}$ by the set of real numbers $\mathbb{R}$ ; we define a *memory state* as a mapping:

$$\sigma : \mathsf{Ref} \longrightarrow \mathbb{B} \cup \mathbb{R}$$

and we denote by $\Sigma$ the set of memory states. Additionally, we assume the existence of the following semantics functions:

$$\begin{array}{l} \mathcal{N} : \mathsf{Num} \longrightarrow \mathbb{R} \\ \mathcal{B} : \mathsf{Bool} \longrightarrow \mathbb{B} \\ \mathcal{F}_k : \mathsf{Fun}_k \longrightarrow ((\mathbb{B} \cup \mathbb{R})^k \longrightarrow \mathbb{B} \cup \mathbb{R}) \quad \text{where } k \in \mathbb{N} - \{0\} \end{array}$$

$\mathcal{N}$ maps a syntactic numeric value to its interpretation in $\mathbb{R}$, $\mathcal{B}$ maps each syntactic boolean value to its interpretation in $\mathbb{B}$, and $\mathcal{F}_k$ provides an interpretation for each $k$-ary syntactic function of $\mathsf{Fun}_k$. With these notations, we can define the denotational semantics of expressions as a mapping:

$$\mathcal{E} : \mathsf{Expr}_{\mathsf{Num}} \cup \mathsf{Expr}_{\mathsf{Bool}} \longrightarrow (\Sigma \longrightarrow \mathbb{B} \cup \mathbb{R})$$

The definition is provided in figure 2.4.

$$\begin{array}{rcl} \mathcal{E}[\![c_n]\!] & = & \lambda\sigma.\mathcal{N}[\![c_n]\!] \\ \mathcal{E}[\![c_b]\!] & = & \lambda\sigma.\mathcal{B}[\![c_b]\!] \\ \mathcal{E}[\![v]\!] & = & \lambda\sigma.\sigma(v) \\ \mathcal{E}[\![f_k(e_1 \ldots e_k)]\!]\sigma & = & \mathcal{F}_k[\![f_k]\!](\mathcal{E}[\![e_1]\!]\sigma \ldots \mathcal{E}[\![e_k]\!]\sigma) \end{array}$$

FIG. 2.4 – Denotational semantics of expressions

In the same manner, we define denotational semantic of programs as a mapping:

$$\mathcal{P} : \mathsf{Stmt} \longrightarrow (\Sigma \longrightarrow \Sigma)$$

Figure 2.5 provides the definition of $\mathcal{P}$. Since a `do` loop can be construct by using a `while` loop, these rules are enough. In the definition of the assignement, the resulting memory state is denoted by $\sigma[x \mapsto t]$ to show that it is identical to $\sigma$ except for the value of $x$. The definitions of `if` and `while` exploit the definitions of **True** and **False**. The semantics of `while` is recursive. One can remark that it is a *fixpoint* of $\lambda f.\lambda\sigma.(\mathcal{E}[\![C]\!]\sigma \;\; f \circ \mathcal{P}[\![S]\!]\sigma) \;\; \sigma$. Consequently, a best definition is:

$$\mathcal{P}[\![\texttt{while } C \texttt{ do } S]\!] = \mathrm{lfp}(\lambda f.\lambda\sigma.(\mathcal{E}[\![C]\!]\sigma \;\; f \circ \mathcal{P}[\![S]\!]\sigma) \;\; \sigma)$$

where $\mathrm{lfp}(f)$ denotes the *least* fixpoint of $f$. In a general manner, $\mathcal{P}$ is *not* computable because of `while` loop.

$$\begin{array}{rcl}
\mathcal{P}[\![v = e]\!] & = & \lambda\sigma.\sigma[v \mapsto \mathcal{E}[\![e]\!]\sigma] \\
\mathcal{P}[\![S_1;\ S_2]\!] & = & \mathcal{P}[\![S_2]\!] \circ \mathcal{P}[\![S_1]\!] \\
\mathcal{P}[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] & = & \lambda\sigma.(\mathcal{E}[\![C]\!]\sigma \;\; \mathcal{P}[\![S_1]\!]\sigma \;\; \mathcal{P}[\![S_2]\!]\sigma) \\
\mathcal{P}[\![\texttt{while } C \texttt{ do } S]\!] & = & \lambda\sigma.(\mathcal{E}[\![C]\!]\sigma \;\; (\mathcal{P}[\![\texttt{while } C \texttt{ do } S]\!] \circ \mathcal{P}[\![S]\!]\sigma) \;\; \sigma)
\end{array}$$

FIG. 2.5 – Denotational semantics of programs statements

Before defining the semantic equivalence, we first precise the notion of *input* and *output* for a program. Basically, a program $P$ achieves a sequence of readings and writings over the references of Ref during its computation.

- The references whose first operation is a reading are called *inputs* of $P$ and denoted by $I_P$.
- The *outputs* of $P$ is a given subset $O_P$ of references.

Given a context $\sigma$, we will write $\sigma_{|O_P}$ the restriction of $\sigma$ to the references of $O_P$. Additionally, we will indifferently talk about *programs* or *algorithms*. With these notations, we define the semantic equivalence between two algorithms as follows.

**Definition 2.7 (Semantic equivalence).** *Two algorithms $A_1$ and $A_2$ are* semantically equivalent *iff*

- *There exists a bijection $\delta : \mathsf{Ref} \longrightarrow \mathsf{Ref}$ linking their inputs within a memory state.*
- *For each context $\sigma$:*
  - *either $A_1$ and $A_2$ don't stop.*
  - *or $A_1$ and $A_2$ stop and output the same value:*

$$(\mathcal{P}[\![A_1]\!]\sigma)_{|O_{A_1}} = (\mathcal{P}[\![A_2]\!](\sigma \circ \delta))_{|O_{A_2}}$$

*In this case, we note $A_1 \equiv A_2$.*

In other words, $A_1$ and $A_2$ are semantically equivalent if they compute the same (mathematical) function. Semantic equivalence is a well known problem, whose undecidability is an immediate consequence of the halting problem.

In this thesis, we will consider a weaker equivalence called *Herbrand-equivalence*. Instead of indicating whether two algorithms compute the same (mathematical) function, Herbrand-equivalence just indicates if they use the same mathematical formula, syntactically. Consider an algorithm $A$ which takes its inputs over an array $I$, and outputs an array $O$. $A$ achieves its computation by using a set $\mathsf{Fun} = \mathsf{Num} \cup \mathsf{Bool} \cup \cup_{k \geq 1} \mathsf{Fun}_k$ of *atomic functions i.e.* constants, arithmetic operators and intrinsic functions. For example, the polynomial product $\times_{\mathbb{R}[X]}$ given in figure 2.1 uses atomic functions `+`, `*` and `0`. If we execute $A$ on a given input $I$ by keeping the atomic functions uninterpreted, we obtain an array $\mathcal{T}_A(I)$ of *terms* over $\mathsf{Fun}$. For instance, taking $a = [\,\boxed{1}\,,\,\boxed{0}\,]$ and $b = [\,\boxed{0}\,,\,\boxed{1+2}\,]$, we obtain:

$$\mathcal{T}_{\times_{\mathbb{R}[X]}}(a,b) = \Big[\ (0 + (\boxed{1} * \boxed{0})) + (\boxed{0} * \boxed{1+2})\ ,\ (0 + (\boxed{1} * \boxed{1+2})) + (\boxed{0} * \boxed{0})\ \Big]$$

Denoting by $\mathcal{T}(\mathsf{Fun})$ the set of *terms* over $\mathsf{Fun}$ ; we define *Herbrand memory states* as memory states with terms of $\mathcal{T}(\mathsf{Fun})$:

$$\sigma_{\mathcal{H}} : \mathsf{Ref} \longrightarrow \mathcal{T}(\mathsf{Fun})$$

and we denote by $\Sigma_{\mathcal{H}}$ the set of Herbrand memory states. We extend the semantics functions $\mathcal{E}$ and $\mathcal{P}$ with Herbrand memory states:

$$\begin{array}{lll} \mathcal{E}_{\mathcal{H}} : \mathsf{Expr}_{\mathsf{Num}} \cup \mathsf{Expr}_{\mathsf{Bool}} & \longrightarrow (\Sigma \times \Sigma_{\mathcal{H}} \longrightarrow (\mathbb{B} \cup \mathbb{R}) \times \mathcal{T}(\mathsf{Fun})) \\ \mathcal{P}_{\mathcal{H}} : \qquad \mathsf{Stmt} & \longrightarrow (\Sigma \times \Sigma_{\mathcal{H}} \longrightarrow \Sigma \times \Sigma_{\mathcal{H}}) \end{array}$$

Figure 2.6 provides the definition of $\mathcal{E}_{\mathcal{H}}$ and $\mathcal{P}_{\mathcal{H}}$. $\mathcal{E}^1_{\mathcal{H}}$ denotes the first component computed by $\mathcal{E}_{\mathcal{H}}$ (in $\mathbb{B} \cup \mathbb{R}$), and $\mathcal{E}^2_{\mathcal{H}}$ denotes the second component. Herbrand memory states basically accumulate the terms computed for each reference used by the program ; while memory states of $\Sigma$ drive the execution of `if` and `while`.

With $\mathcal{P}_{\mathcal{H}}$, one can define formally the notion of term computed.

**Definition 2.8 (Term computed).** *Consider an algorithm $A$, and $\sigma$ and $\sigma_{\mathcal{H}}$ two equivalent memory states containing an input $I$. We write:*

$$\mathcal{T}_A(I) = ((\lambda xy.y)\mathcal{P}_{\mathcal{H}}[\![A]\!]\sigma\sigma_{\mathcal{H}})_{|O_A}$$

*And we say that $\mathcal{T}_A(I)$ is the* term computed *by $A$ on the input $I$.*

When $A$ outputs an array, $\mathcal{T}_A(I)$ is an *array* of terms and we denote by $\mathcal{T}_A(I)[\vec{i}]$ the term indexed by $\vec{i}$.

$$
\begin{array}{rcl}
\mathcal{E}_{\mathcal{H}}[\![c_n]\!] & = & \lambda\sigma\sigma_{\mathcal{H}}.(\ \mathcal{N}[\![c_n]\!]\ ,\ c_n()\ ) \\
\mathcal{E}_{\mathcal{H}}[\![c_b]\!] & = & \lambda\sigma\sigma_{\mathcal{H}}.(\ \mathcal{B}[\![c_b]\!]\ ,\ c_b()\ ) \\
\mathcal{E}_{\mathcal{H}}[\![v]\!] & = & \lambda\sigma\sigma_{\mathcal{H}}.(\ \sigma(v)\ ,\ \sigma_{\mathcal{H}}(v)\ ) \\
\mathcal{E}_{\mathcal{H}}[\![f_k(e_1 \dots e_k)]\!]\sigma\sigma_{\mathcal{H}} & = & (\ \mathcal{F}_k[\![f_k]\!](\mathcal{E}^1_{\mathcal{H}}[\![e_1]\!]\sigma\sigma_{\mathcal{H}} \dots \mathcal{E}^1_{\mathcal{H}}[\![e_k]\!]\sigma\sigma_{\mathcal{H}})\ , \\
 & & \qquad f_k(\mathcal{E}^2_{\mathcal{H}}[\![e_1]\!]\sigma\sigma_{\mathcal{H}} \dots \mathcal{E}^2_{\mathcal{H}}[\![e_k]\!]\sigma\sigma_{\mathcal{H}})\ ) \\
 & & \\
\mathcal{P}_{\mathcal{H}}[\![v = e]\!] & = & \lambda\sigma\sigma_{\mathcal{H}}.(\ \sigma[v \mapsto \mathcal{E}^1_{\mathcal{H}}[\![e]\!]\sigma\sigma_{\mathcal{H}}]\ ,\sigma_{\mathcal{H}}[v \mapsto \mathcal{E}^2_{\mathcal{H}}[\![e]\!]\sigma\sigma_{\mathcal{H}}]\ ) \\
\mathcal{P}_{\mathcal{H}}[\![S_1;\ S_2]\!] & = & \mathcal{P}_{\mathcal{H}}[\![S_2]\!] \circ \mathcal{P}_{\mathcal{H}}[\![S_1]\!] \\
\mathcal{P}_{\mathcal{H}}[\![\texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] & = & \lambda\sigma\sigma_{\mathcal{H}}.(\mathcal{E}^1_{\mathcal{H}}[\![C]\!]\sigma\sigma_{\mathcal{H}}\ \ \mathcal{P}_{\mathcal{H}}[\![S_1]\!]\sigma\sigma_{\mathcal{H}}\ \ \mathcal{P}_{\mathcal{H}}[\![S_2]\!]\sigma\sigma_{\mathcal{H}}) \\
\mathcal{P}_{\mathcal{H}}[\![\texttt{while}\ C\ \texttt{do}\ S]\!] & = & \lambda\sigma\sigma_{\mathcal{H}}.(\mathcal{E}^1_{\mathcal{H}}[\![C]\!]\sigma\sigma_{\mathcal{H}} \\
 & & \qquad (\mathcal{P}_{\mathcal{H}}[\![\texttt{while}\ C\ \texttt{do}\ S]\!] \circ \mathcal{P}_{\mathcal{H}}[\![S]\!])\sigma\sigma_{\mathcal{H}} \\
 & & \qquad (\sigma, \sigma_{\mathcal{H}}))
\end{array}
$$

FIG. 2.6 – Denotational semantics for Herbrand-equivalence

Following the work of Knoop *et al.* [91], we extend the Herbrand-equivalence to algorithms as stated in the following definition.

**Definition 2.9 (Herbrand-equivalence).** *Two algorithms $A_1$ and $A_2$ are* Herbrand-equivalent *iff:*

- *either $A_1$ and $A_2$ do not stop.*
- *either $A_1$ and $A_2$ stop and output the same array of terms:*

$$\mathcal{T}_{A_1}(I)[\vec{i}] = \mathcal{T}_{A_2}(I)[\vec{i}]$$

  *For each relevant $\vec{i}$.*

*In this case, we note $A_1 \equiv_{\mathcal{H}} A_2$.*

In other words, $A_1$ and $A_2$ are Herbrand-equivalent if they compute the same array of terms. The following proposition establishes that Herbrand-equivalence is an equivalence relation between programs, and is weaker than semantic equivalence.

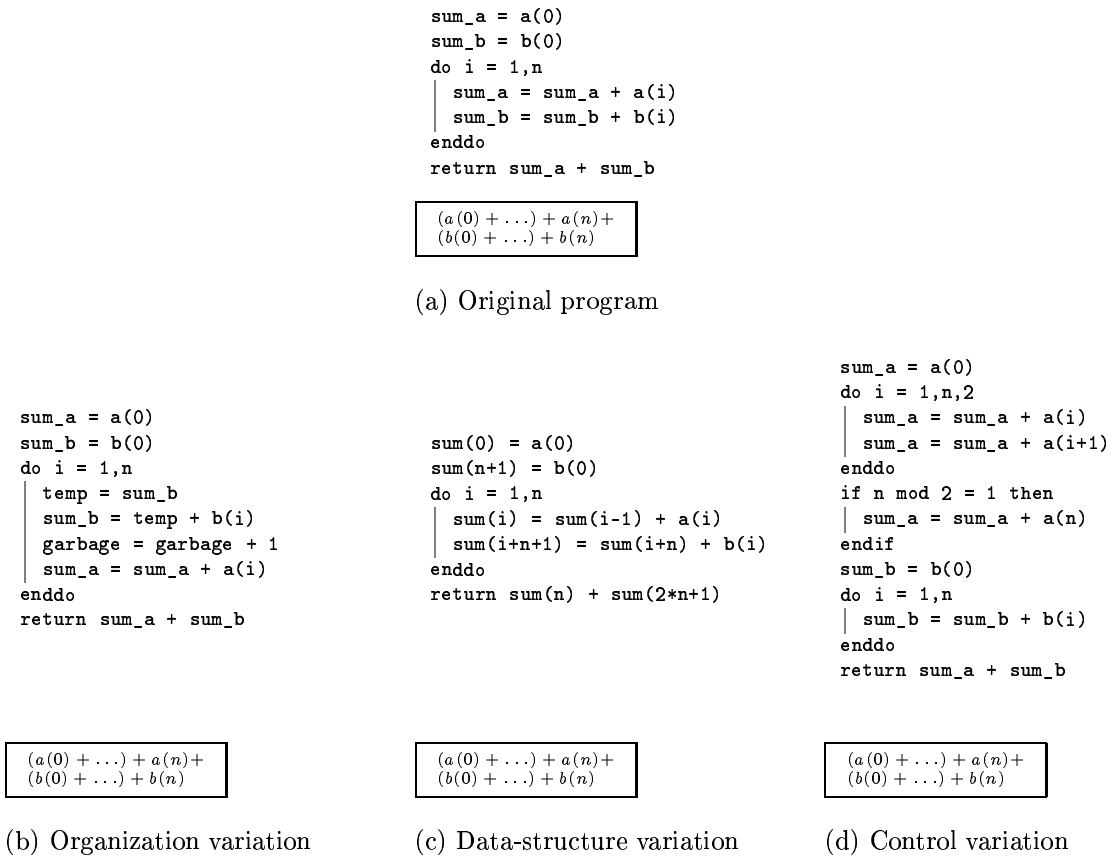
**Proposition 2.2 (Correction).** *Given two algorithms $A_1$ and $A_2$, we have:*

$$A_1 \equiv_{\mathcal{H}} A_2 \Longrightarrow A_1 \equiv A_2$$

*Proof.* Assume $A_1 \equiv_{\mathcal{H}} A_2$ and consider an input $I$ such that $A_1(I)$ and $A_2(I)$ terminate. Since $\mathcal{T}_{A_1}(I) = \mathcal{T}_{A_2}(I)$ are in the Herbrand universe, $[\![\mathcal{T}_{A_1}(I)]\!]_{\mathcal{I}} = [\![\mathcal{T}_{A_2}(I)]\!]_{\mathcal{I}}$ for every interpretation $\mathcal{I}$ of atomic operations. Thus $A_1(I) = [\![\mathcal{T}_{A_1}(I)]\!]_{\mathcal{I}} = [\![\mathcal{T}_{A_2}(I)]\!]_{\mathcal{I}} = A_2(I)$. □

Although Herbrand-equivalence is weaker than semantic equivalence, it has been proven undecidable in [13]. In spite of equivalence classes smaller than full semantic equivalence, Herbrand-equivalence covers all program transformations which do not change the

term computed by the program. Such transformations include in particular all standard loop transformations such as splitting, fusion, unroll, skewing or tiling. More loop transformations can be found in [104]. For the same reasons, organization and data-structure variations are covered. These important characteristics of Herbrand-equivalence are summarized in figure 2.7.




(a) Original program

(b) Organization variation

(c) Data-structure variation

(d) Control variation

FIG. 2.7 – Program variations supported by Herbrand-equivalence

Unfortunately, Herbrand-equivalence does not cover program variations taking semantics properties of operators into account. For instance, associativity and commutativity of $+$. This property is illustrated in figure 2.8 with three algorithms $A_1$, $A_2$ and $A_3$ semantically equivalent such that:

$$A_1 \equiv_{\mathcal{H}} A_2 \not\equiv_{\mathcal{H}} A_3$$

Indeed, $A_1$ and $A_2$ compute the same term:

$$\mathcal{T}_{A_1}(a) = \mathcal{T}_{A_2}(a) = ((a(0) + a(1)) + \ldots + a(9)) + a(10)$$

whereas $A_3$ computes the term:

$$\mathcal{T}_{A_3}(a) = a(10) + (a(9) + \ldots + (a(1) + a(0)))$$

which is not *syntactically* equal to $\mathcal{T}_{A_1}(a)$ and $\mathcal{T}_{A_2}(a)$.

```
s = a(0)
do i = 1,10
  s = s + a(i)
enddo
return s
```

$(A_1)$

```
s = a(0)
do i = 1,5
  s = s + a(i)
enddo
do i = 6,10
  s = s + a(i)
enddo
return s
```

$(A_2)$

```
s = a(0)
do i = 1,10
  s = a(i) + s
enddo
return s
```

$(A_3)$

FIG. 2.8 – Three algorithms semantically equivalent with $A_1 \equiv_{\mathcal{H}} A_2 \not\equiv_{\mathcal{H}} A_3$.

## 2.6 Template Matching and Recognition

This section states formally the problem addressed throughout this thesis. After defining formally the matching problem at program level, we present the template recognition problem. Particularly, we propose an interesting decomposition which will be followed in this thesis. The *template matching problem* consists in deciding whether a program is an instance of a template, as stated in the following definition.

**Problem 1 (Template Matching).** *The* template matching problem *between a template $T$ and a program $P$* w.r.t. *a given program equivalence $\sim$ consists in finding the values for the free variables $\vec{X}$, the parameters $\vec{n}$ and the template inputs $\vec{I}$ such that:*

$$T[\vec{X}, \vec{n}, \vec{I}] \sim P$$

*By analogy with unification theory, such a solution, if it exists, is called an* semi-unifier *of the template matching problem $T \stackrel{?}{=} P$.*

Since this problem is more difficult than the equivalence problem *w.r.t.* $\sim$, it is also undecidable over $\equiv$ and $\equiv_{\mathcal{H}}$. The following definition introduces the more general problem of *template recognition*, that aims at finding all program slices equivalent to a specific template instance.

**Problem 2 (Template Recognition).** *Given a template $T$, a program $P$ and an equivalence $\sim$ on programs, the* template recognition problem $T \leq^{?} P$ *consists in finding* all *the slices of $P$ that match with $T$* w.r.t. $\sim$.

This problem is obvioulsy more difficult than the template matching problem, and is therefore undecidable *w.r.t.* $\equiv$ and $\equiv_{\mathcal{H}}$. This thesis addresses the template recognition problem *w.r.t.* the Herbrand-equivalence $\equiv_{\mathcal{H}}$. Our method consists in two passes, based on two approximations of Herbrand equivalence. An over-approximation of Herbrand equivalence is first used to discover an over set of the solution slices. Then, an exact sub-approximation of Herbrand equivalence is used to filter the right slices. The following proposition states formally this decomposition for any equivalence relation $\sim$.

**Proposition 2.3 (Decomposition).** *Consider a template $T$, a program $P$, an equivalence relation $\sim$ over programs and $\approx$ an over-approximation of $\sim$ ($\sim \subseteq \approx$). The following procedure solves the template recognition problem $T \leq^? P$ w.r.t. $\sim$:*

1. *Solve the template recognition problem $T \leq^? P$ w.r.t. $\approx$.*
2. *For each slice $S$ obtained:*
   *Solve the matching problem $T \stackrel{?}{=} P$ w.r.t. $\sim$*

*Proof.* Consider a slice $S$ satisfying $T \leq^? P$ *w.r.t.* $\sim$. Since $\sim \subset \approx$, $S$ is obtained in step 1. Since $S$ matches $T$, $S$ is obtained in step 2. Conversely, consider a slice $S$ obtain from the above procedure. $S$ is a correct slice of $P$ (step 1), which matches $T$ *w.r.t.* $\sim$ (step 2). This leads to the result. □

A trade-off must be found in the choice of $\approx$. Indeed, a too rough approximation would provide too many slices to the expensive step 2, whereas a too precise approximation will be too expensive at step 1. The recognition framework described in this thesis respects this decomposition. Especially, step 1 will be called *slicing* ; and detailed in Chapter 5. The choice of $\approx$, and the amount of false positives will be discussed. We also propose two differents heuristics to achieve step 2 in Chapters 7 and 8. These two different procedures solve the matching problem *w.r.t.* $\sim_1$ *resp.* $\sim_2$, with $\sim_1 \subset \sim_2 \subset \equiv_{\mathcal{H}} \subset \equiv$. Before describing our approach, we present several related works in the next chapter.

# Chapter 3

# Related Work

Algorithm recognition is an old and difficult problem of computer science, that has been intensively addressed during the AI period (from the 70s to the 90s), and more sporadically after. Most of the existing approaches are motivated by the maintenance of large softwares and tends to provide frameworks helping the programmer to understand, debug and maintain large softwares. These approaches are essentially based on psychological experiments [100, 94] showing that in trying to *understand* a program, an experienced programmer may *recognize* parts of the program design by identifying frequently used computational structures in the code. Most of the existing approaches uses *graph-parsing* to perform the recognition [102, 60], while further approaches use *symbolic execution* and general reasoning techniques [23]. Unfortunately, as far as we know, only a few approaches are concerned with program optimization. As for program understanding approaches, most of them rely on graph-parsing to perform the recognition [86, 71], while other approaches uses heuristics to achieve the matching on the abstract syntax tree of the program [64, 17, 76].

This chapter presents and discusses several important works related to algorithm recognition. Section 3.1 presents the methods concerned with reverse engineering and program understanding, whereas Section 3.2 describes the few existing approaches motivated by program optimization. The limitations of each method is pointed out, particularly, we discuss – when it is possible – the scalability and the detection capabilities in terms of program variations, as defined in introduction.

## 3.1 Program Understanding Approaches

Cognitive studies show how humans understand program, and provides the factors that affect their understanding. This section summarizes several psychological aspects that become fundamental bases of program understanding techniques.

### 3.1.1 Cognitive Studies

The programs to handle are often written by humans. Some psychological experiments show the existence of recurring patterns in most programs, and a hierarchical structure between them [100, 94], exploited by numerous algorithm recognition methods.

Ehrlich and Soloway [94] have established experimentally the use of recurring basic patterns in programming. According to Ehrlich, programming knowledge is a set of frame-like structures, called *plans*, for handling stereotypical situations which arise frequently in programming. Basically, a programmer would possess plans in memory for operations such as iteration, counting, accumulating values, etc. These basic plans would serve as building blocks for writing programs and also as the knowledge base for understanding programs. More precisely, they establish that experimented programmer's comprehension is disturbed by programs that were written without a plan. They found that subjects who have seen a subtle unplan-like program tended naturally to see a plan-like program instead. It suggests that plan structure form a part of their mental representation.

Letovski [68] presents the mental representation of a program as a layered network whose nodes represent goals and subgoals to be achieved ; each level representing the subgoals needed to achieve the next level. This hierarchical organization of the knowledge must be enough explicit in the program code to improve the program comprehension, and thus to increase the productivity while maintaining software.

### 3.1.2 Will's GRASPR

Wills [102] investigates the application of *graph-parsing* to algorithm recognition. She represents the program by a particular kind of dependence graph called *flow-graph*, whose nodes and edges capture the data-flow information of the program. Figure 3.1 provides the flow-graph of a selection-sort. One may remark the *control environment* attributes, representing the control information. In the figure, the execution starts with nodes labeled with `ce1`. If the predicate $<$ succeed, the control is given to nodes labeled with `ce2`, otherwise, nodes with `ce3` are executed.

The algorithm to recognize is represented by *flow-graph grammar rules*, that encode the algorithm as a hierarchy of plans. The recognition is then achieved by parsing the program graph according to the grammar rules. Figure 3.2 provides the parsing tree obtained while recognizing the selection sort. The parsing tree provides a hierarchical description of the program in terms of plans, that Wills call *clichés*. In addition to recover algorithms, such a hierarchy provides an informal description of the program organization to the user, allowing to improve the program comprehension. An important drawback of this approach is the difficulty to maintain pattern base. Indeed, it requires the user to add manually the graph-grammar rules whenever he adds a new plan to the library.

We now evaluate Will's work by using the criteria described in introduction.

**Scalability** Wills has shown that the flow-graph parsing is NP-complete, and argues that even if the cost of her algorithm is exponential in the worst case, it is feasible to apply it to practical partial program recognition. However, no evaluation of the method is provided on real applications.

**Variations detected** *Organization variations* are partially supported thanks to the graph representation. All others variations (data-structure, control and semantics) can be handled only if they are explicitly described in the pattern base.

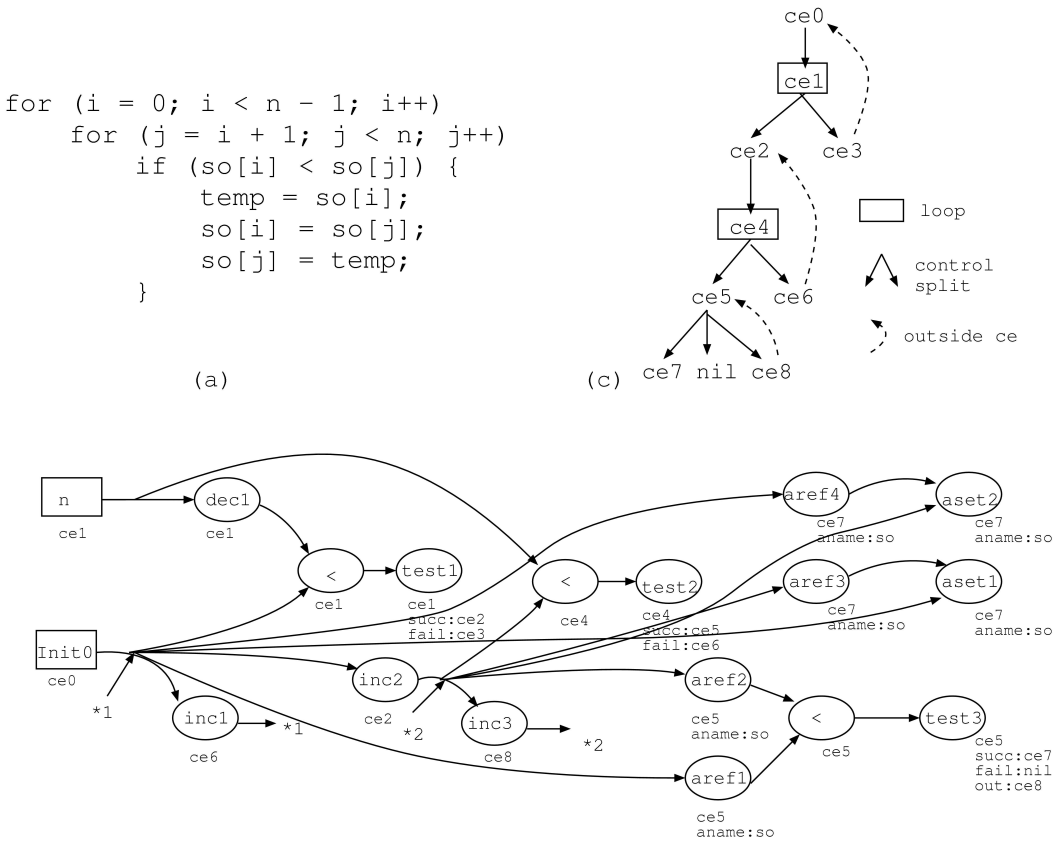



FIG. 3.1 – Flow-graph of a selection sort [102]

### 3.1.3 Johnson's PROUST

PROUST [60] is another graph-parsing approach to analyze and debug Pascal programs written by novice programmers. It takes as input the *expected* goals of the program, and a rule-based hierarchical description between goals, and program patterns (plans) that achieve them. Based on this information, PROUST performs a *top-down* analysis of the program by searching the space of goal decomposition.

PROUST relies on *plan difference rules* to detect bugs and to propose a correction. Plan difference rules basically provides the common bugs associated to a given plan. As a consequence, PROUST capabilities strongly depends on the knowledge base. Additionally, PROUST employs heuristics able to detect the bugs which are not specified by a plan difference rule.

**Scalability** The scalability of the system will depends on the performance of the search engine. The authors do not provide any complexity study of their method. Since PROUST uses graph-parsing to achieve the recognition, we can assume that the method is expensive, and consequently not scalable.

**Variations detected** In the same manner as Will's approach, the organization variations seems to be correctly supported, and the handling of the other variations directly depends on the knowledge base.

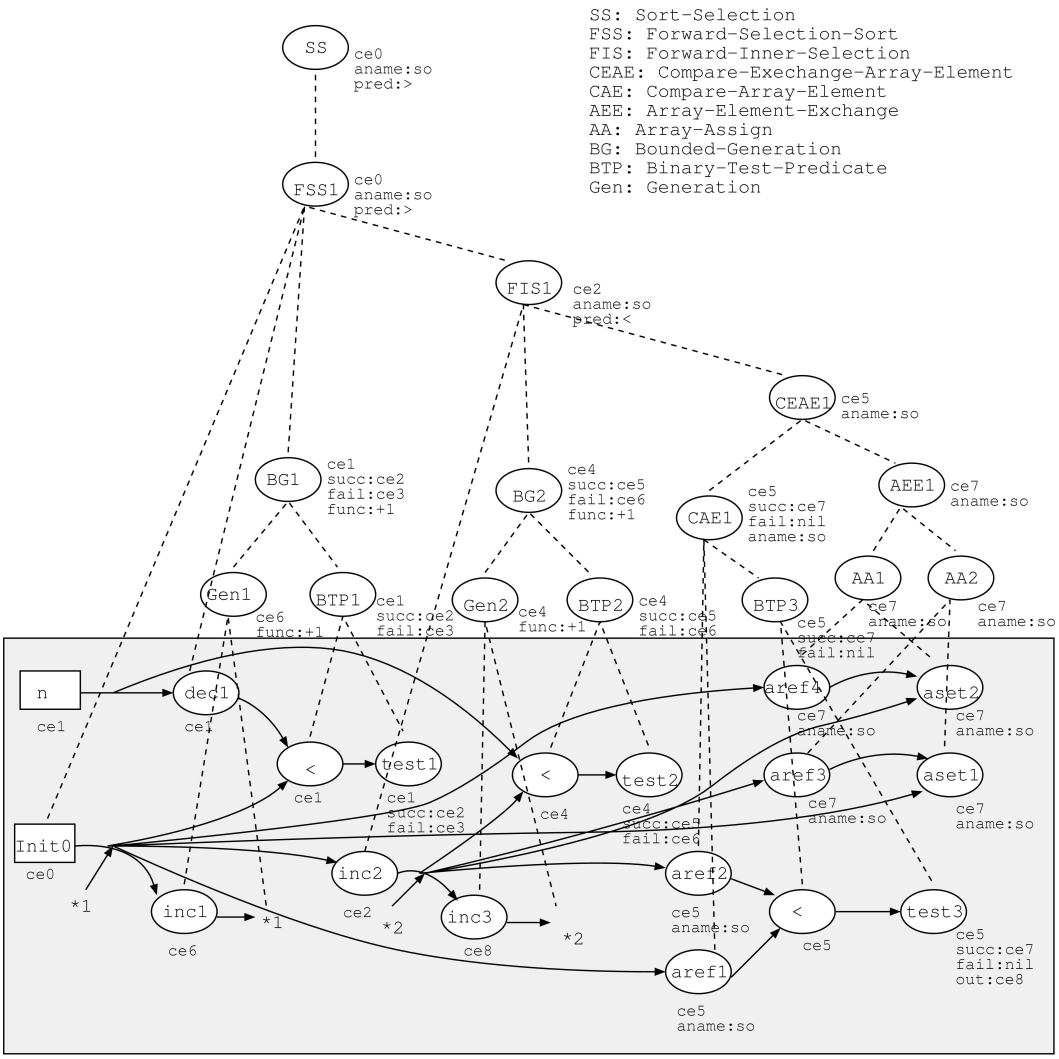



FIG. 3.2 – Parsing tree of selection sort [102]

### 3.1.4 Cimitile's System

Cimitile, De Lucia and Munro [23] address the application of symbolic execution and general reasonning techniques on algorithm recognition. They propose a method to identify the slices verifying given pre-conditions and post-conditions, which consists in two steps. They first compute a symbolic execution of the program, which assigns to each statement its pre-condition.

The recovered conditions are then compared with the precondition and the postcondition of the abstraction. The comparison is achieved by a theorem prover which may need user interaction to associate post-condition variables to program variables. Moreover, as the problem of finding invariant assertions is in general *undecidable*, symbolic execution may require user interaction to prove some assertions and assert some invariants.

The statements where conditions are respectively equivalent to the precondition and the postcondition of the specification are candidate to be the entry and exit points of the slice implementing the abstraction.

**Scalability** No practical evaluation of their method, or theoretical study of complexity is given, but their method seems to be costly. Moreover, the need for user interaction makes the method inappropriate for large programs.

**Variations detected** Organizations variations seems to be handled. The detection of the other variations directly depends on the capabilities of the theorem prover.

## 3.2 Optimization Approaches

We present now several approaches designed in the context of program optimization. These approaches aims to be used in a fully automatic framework, such as a compiler.

### 3.2.1 Pinter's System

Pinter and Pinter [86] propose a fully automatic approach to recognize algorithms, and to replace them by a call to an optimized function. They first preprocess the program by applying standard transformations such as dead-code elimination, common sub-expression detection, etc. The loops are also normalized in order to put each loop carried dependance between two consecutive iterations at most. Such a normalization cannot always be applied, and is achieved by applying an unrolling. The preprocessed program is then translated into a *computation graph*, a dependence graph whose nodes are labelled by assignments operators, and indicates the dependence distances.

The matching and the replacement is then achieved by using graph grammars. While rewriting, the recognized parts are replaced by the relevant idiom. Their list of idioms particularly includes reductions, scan, recurrence equations, transposition, reflection and FFT butterflies.

**Scalability** As for graph parsing approaches, their approach relies on the sub-graph isomorphism problem which is NP-complete. Since no complexity study is given, we can assume that their method is expensive.

**Variations detected** Due to the graph representation, organization variations are detected. The others variations handled depends on the various preprocessing applied on the program.

### 3.2.2 Di Martino's PAP

Di Martino and Iannello [71] propose a bottom-up graph-parsing approach similar to Wills', which is essentially motivated by the recognition of parallelizable patterns. It uses a hierarchical description of patterns, encoded by grammar rules. The basic level of the representation is substantially a Program Dependence Graph (PDG), whose nodes represents statements and edges represents control and data dependences.

The graph-grammar rules and the program dependence graph are encoded by `prolog` clauses, then the recognition is achieved by resolution. No information is provided on the substitution. The evaluation is the same as Will's approach.

**Scalability** Since the graph-parsing problem is NP-complete, we can claim that their approach is not scalable.

**Variations detected** As Wills' approach, the variations detected are limited to organization variations, and algorithmic variations described in the pattern base.

### 3.2.3 Kessler's PARAMAT

PARAMAT [64] is an algorithm recognition system motivated by automatic parallelization. As well as Pinter's approach, the program is first normalized by applying several transformations including constant propagation, induction variables detection and replacement, and dead code elimination. The recognition procedure is based on an exact matching on the abstract syntax tree of the program. Whenever a conditional is reached, the algorithm tries to matches directly. In case of failure, it applies an if-distribution. In the same manner, the loops will be distributed in case of failure.

For each implementation pattern, PARAMAT provides a *runtime prediction driver* that inspects a table of precomputed measured runtimes with varying problem sizes. The main drawback of this approach is obviously the computation of the bench table, whose size grows exponentially with the number of parameters. However, once the tables are computed, the run-time prediction is faster and more accurate than theoretical estimation functions [40, 53]. The back-end code generation is then achieved by an *implementation driver*, that generates the code for any instance of a given pattern.

**Scalability** The pattern recognition uses a graph matching on an AST, which seems to be linear in the program size. As a consequence, PARAMAT seems to be scalable, even if no experimental results are provided.

**Variations detected** The normalization applied on the program allow PARAMAT to detect some control variations. However, simple organization variations such as variable renaming, or data-structure variation are not handled by the matching procedure.

### 3.2.4 Bhansali's System

Bhansali and Hagemeister [17] propose an approach to parallelization based on recognition of domain-specific concepts within a sequential program, followed by a substitution by the corresponding parallel routine. They rely on a simple pattern language, close to the syntax of the target programming language, which is more readable and thus easier to maintain than those of the previous approaches. The pattern language uses *typed wildcards* to match differents syntactic entities in the source such as variables, functions or type declaration. The wildcards allow to handle simple variations such as variable renaming.

The pattern library is organized in a hierarchical fashion, dividing domains in subdomains, downto a list of patterns. An interesting fact is that different domains can share the same basic idioms.

While searching patterns in source code, the user is asked to narrow the set of patterns to look for by going down the hierarchical tree, and choosing a node corresponding to the patterns *expected* in the source code. The pattern recognition technique is based on an exact matching on the abstract syntax tree (AST) of the program, which first looks for basic idioms, then tries to combine them in order to recover patterns. Since the search strategy combines top-down (user expectation) and bottom-up (recognition) approaches, it is said to be *hybrid*. The authors do not provide any information about the substitution by a library call.

**Scalability** The matching technique is based on Paul and Prakash approach [84], which quadratic in the program size. Even if no experimental results are provided, we can assume that their approach is scalable.

**Variations detected** Since the pattern recognition techique is based on an exact matching on the program's AST, the variations detected seems to be limited to variable renaming. However, in the same manner as previous approaches, the detection capabilities strongly depend on the pattern base.

### 3.2.5 Metzger's System

Metzger and Wen [76] have built a complete environment to recognize and replace algorithms. They first normalize the program and pattern AST by applying classical program transformations (if-conversion, loop-splitting, scalar expansion...). Then they search the program for good candidate slices. The candidate slices are strongly connected components of the dependence graph, containing at least one `for` statement.

In the same manner as Bhansali's approach, their equivalence test is based on an isomorphism between the slice and the pattern AST. Given a set of correct slices, several slices may overlap, leading to choose a particular subset of slices for substitution. They also propose an approach to select the set of substitutions which should lead to the best performance improvement. To perform this task, they associate to each algorithm a number called *saving*, which quantifies the corresponding performance improvement. Then they finally choose the set of substitutions which maximize the total savings.

**Scalability** Obviously, this approach is low cost, and scalable. One may point out the large amount of candidate slices given by the first pass, but it is not a real problem thanks to the low complexity of their equivalence test.

**Variations detected** Organization variations, resulting from the permutation of independent statements or the introduction of temporaries are not handled by the algorithm itself, but by preprocessing applied to the program. Reuse of temporaries accross loop iterations for instance is not handled. In the same way, the control variations supported are bounded by preprocessing.

## 3.3 Conclusion

The evaluation of the different approaches presented in this chapter is summarized in the table 3.1.

| Approach | Category | Technique | Replacement | Scalability | Variations detected |
|---|---|---|---|---|---|
| Will's GRASPR [102] | Prog. understanding | Bottom-up graph parsing | No | No | Organization |
| Johnson's PROUST [60] | P.U. & aut. debug. | Top-down graph-parsing | No | No | Organization |
| Cimitile's system [23] | Prog. understanding | Symbolic execution | No | No | Potentially all |
| Pinter's system [86] | Optimization-based | Bottom-up graph-parsing | Yes | No | Organization |
| Di Martino's PAP [71] | Optimization-based | Bottom-up graph-parsing | No | No | Organization |
| Kessler's PARAMAT [64] | Optimization-based | AST matching | Yes | Yes | Organization & Control |
| Bhansali's system [17] | Optimization-based | AST matching | No | Yes | Organization |
| Metzger's system [76] | Optimization-based | AST-matching | Yes | Yes | Organization |

TABLE 3.1 – Some approaches and their evaluation

AST-based approaches are able to cope with the same amount of variations than dependence graph-based methods, whenever the control structures are normalized with transformations reducing the control variations. As a consequence, a key point to detect as much variations as possible is to work on an intermediate representation as much independent as possible of control structures.

Most approaches enumerate exhaustively all the slices without using a slicing method. It is typically the case of graph-parsing approaches. Metzger's system is the only approach which uses a decomposition slicing/equivalence test as described in chapter 2. The approach addressed in this thesis uses such a decomposition, and performs the recognition on an intermediate representation as independent as possible of control structures. The next chapter presents our approach, and applies it to match BLAS functions [67] on a simple linear algebra kernel.

# Chapter 4

# Overview of the Optimization Framework

The approaches described in Chapter 3 suffers of several drawbacks. First of all the equivalence tests are often based on exact graph matching over ASTs or graph representation of the program, and cannot handled complex variations such as data-structure or control variations. Moreover, none of these methods can recognize templates instances, and rewrite a program according to common template libraries.

In this thesis, we propose a complete algorithmic framework to recognize template instances within a program, and to substitute them by a call to an optimized library when it leads to a performance improvement. In addition to organization variations, our template recognition method is able to handle most of data-structure and control variations. Our template recognition method follows the decomposition presented in Chapter 2, Section 2.3 and consists in a *slicing method* which finds an over-approximation of the matching program slices, that we filter by applying an *exact matching procedure*. A pass of selection of the slices to replace follows, then the code with the right substitutions is generated. Section 4.1 describes the underlying hypothesis and the main steps of our method, then Sections 4.3, 4.4 and 4.5 apply the whole method on the motivating example given in Section 4.2.

## 4.1 Overview of the Framework

Figure 4.1 summarizes the different steps of our optimization framework. We take as input a C or Fortran 90 program, and an optimized library. The library is assumed to provide a public interface of naive implementations for each function. The template matching problem is handled by following the decomposition described in Chapter 2, Proposition 2.3. An over-approximation of Herbrand equivalence is used to find the program slices which *possibly* matches the template (*slicing method*). The result is a set of slices which contains the relevant slices (conservativity). An exact sub-approximation of Herbrand equivalence is then used to check whether the slices matches the template (*Instantiation test*). Once the correct slices are found, they are substituted by a library call, whenever it is possible and interesting.

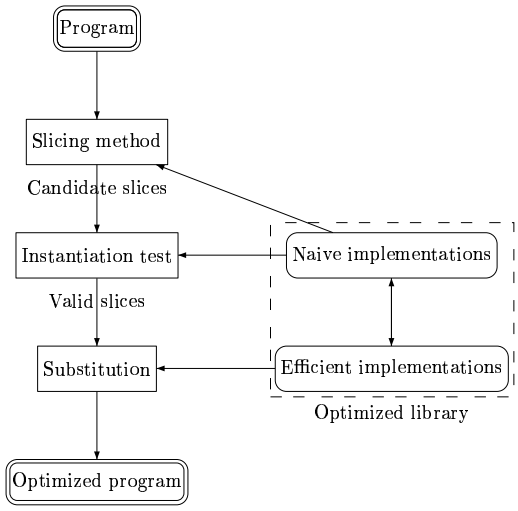



FIG. 4.1 – Main steps of our optimization framework

We provide thereafter a quick summary of these steps. The precise descriptions are in the following chapters. Chapter 5 describes the slicing method, Chapters 7 and 8 provides two instantiation test based on different theoretical frameworks. Finally, the substitution step is described in Chapter 9.

## 4.2 Motivating Example

Given a matrix $A$ of $M_n(\mathbb{R})$ and a vector $\mathbf{x} \in \mathbb{R}^n$, the *Krylov family* associated to $A$ and $\mathbf{x}$ is defined by:

$$\mathcal{K}_p(A, \mathbf{x}) = \{\mathbf{x}, A\mathbf{x}, A^2\mathbf{x}, \ldots, A^p\mathbf{x}\}$$

Where $p \leq n$. Krylov families are commonly used in iterative methods such as Arnoldi restarting method [10] to find eigen-values of large matrices. A quick and dirty program to compute $\mathrm{Kr}_p$ (left) and its implementation (right) could be written in the following way:

```
K_{0,.} = x
do i = 1,p
| K_{i,•} = A × K_{i-1,•}
enddo
```

```
K(0) = x
do i = 1,p
| do i' = 1,n
| | K(i,i') = 0
| | do j' = 1,n
| | | K(i,i') = K(i,i') + A(i',j')*K(i-1,j')
| | enddo
| enddo
enddo
```

Where $\times$ denotes a matrix-vector product, which is computed by the `do` loops with counters `i'` and `j'`. We aim at rewrite this program by using the BLAS library [67] (Basic Linear Algebra Subroutines), an efficient library to implement linear algebra algorithms. As stated above, we assume that a naive implementation is provided for each BLAS function.

These naives implementations include particularly the following functions, where $A$ is a squared matrix, $\vec{x}$ and $\vec{y}$ are two vectors, and $\alpha$ is a scalar:

| | |
|---|---|
| dscal | $\vec{x} \leftarrow \alpha\vec{x}$ |
| daxpy | $\vec{y} \leftarrow \alpha\vec{x} + \vec{y}$ |
| ddot | $\alpha \leftarrow \vec{x}.\vec{y}$ |
| dtrmv | $\vec{x} \leftarrow A\vec{x}$ |

Within the running example, we should obviously find the matrix-vector product and substitute it by dtrmv. We summarize thereafter the steps of recognition (*slicing method* and *instantiation test*) and substitution.

## 4.3 Slicing

We first apply an *approximate* matching, the slicing method, which provides the different possible occurrences of the library functions within the program. We obtain a set of slices containing right occurences, but also wrong occurences, which will be filtered during the next step. The results are given in the following table.

| BLAS function | Occurrences found |
|---|---|
| dscal | none |
| daxpy | $\{S_2\}$ |
| ddot | $\{S_1\}$, $\{S_1, S_2\}$ |
| dtrmv | $\{S_1\}$, $\{S_1, S_2\}$ |

The slicing method yields a set of statements over-approximating the slice. The exact slice – in the meaning defined in chapter 2 – will be computed during the instantiation test. Additionally, the surrounding loops to handle are not specified. This important information will be recovered during the next step, the instantiation test.

The daxpy slice is a correct but trivial detection, constituted of an instance of $S_2$ with $\alpha = A(i', j')$, $\vec{x} = [K(j', j-1)]$ and $\vec{y} = [K(i', j)]$. Such a detection is obviously not interesting to replace, and will be discarded during the substitution pass.

Consider ddot. The slice reduced to $\{S_1\}$ is also a trivial detection, assuming that 0-dimension vectors are handled. The slice $\{S_1, S_2\}$ is a good candidate, restricted to $S_1$ and $S_2$ with the surrounding `do` loop with counter `j'`.

Consider dtrmv. $\{S_1\}$ is a wrong detection, and $\{S_1, S_2\}$ is a good candidate including the `do` loop with counters `i'` and `j'`. Remark that this slice overlaps with the dot product found at $\{S_1, S_2\}$. The overlapping between slices induces several substitution sets, leading to choose the best one during the substitution pass.

## 4.4 Template Matching

Once the candidate slices are found in the program, it remains to check whether they really match the naive implementations of library functions, by using an *exact* matching, the *instantiation test*. In addition to (semi)-decide whether the slice matches the naive

implementation, the instantiation test provides also the exact slices in case of success. For the dot product ddot, we obtain the following slice:

$$\begin{array}{ll} \langle S_1, i, i' \rangle & i \text{ and } i' \text{ are parameters} \\ \langle S_2, i, i', j' \rangle & 1 \leq j' \leq n \end{array}$$

In the same manner the following slice is obtained for the matrix-vector product dtrmv:

$$\begin{array}{ll} \langle S_1, i, i' \rangle & i \text{ is a parameter and } 1 \leq i' \leq n \\ \langle S_2, i, i', j' \rangle & 1 \leq i', j' \leq n \end{array}$$

The trivial instances are also detected, and will be discarded during the substitution pass. Remark that the efficiency of our optimization framework directly depends on the capacity of our instantiation test to handle many program variations.

## 4.5 Substitution

Once the equivalent slices are found, it remains to substitute them by a call to the optimized library. The flow-dependences between a slice and the interleaved operations can forbid a substitution. Here is an example of program where a daxpy has been discovered. The slice is constituted of operations $\langle S, 1 \rangle$ to $\langle S, n \rangle$:

```
      do i = 1,n
S     |  y(i) = y(i) + a*x(i)
      |  x(i+1) = 2*y(i)
      enddo
```

Since the slice and the interleaved operations depends on each other, we cannot *separate* the slice from the interleaved operations to substitute it by a call to daxpy. Hence, a first step is to select the slices which can be extracted by using a reaching definition analysis.

Since several slices may overlap, we have to select the set of slices whose substitution will lead to the best performance. A first approach is to associate to each library function a number quantifying its capacity to increase program performances. For example, we could assume that we have the following gains:

| BLAS function | Gain |
|---|---|
| dscal | 1 |
| daxpy | 2 |
| ddot | 3 |
| dtrmv | 4 |

The gains are purely arbitrary, and will be used to make a choice whenever several slices overlap. The optimal set of slices to substitute is then selected by solving a 0-1 program whose constraints express the overlappings between slices. It will leads here to choose the slice matching dtrmv. This approach is straighforward, and could be improved be using more sophisticated performance prediction tools [21, 51, 41].

Once a relevant set of slices is selected, we generate the code with the substitution. Here, we would generate the following code, where the line 3 maps the program variables to the function library inputs:

```
1 K(0) = x
2 do i = 1,p
3 |  I(1:n) = K(i-1,1:n)
4 |  K(i,1:n) = dtrmv(A,I)
5 enddo
```

Experimental results on SpecFP 2000 [54] and Perfect Club [39] benchmarks provided in Chapter 10 will show that in most cases, the code to construct the inputs can be avoided, and replaced by adding a *stride* in the call to the BLAS function.

# Chapter 5

# Slicing

Given an instantiation test, a naive solution for template recognition would be to enumerate all the program slices, and to compare them to the template. Due to the important number of program slices, and the cost of comparison, such a solution is unrealistic. Consequently, a slicing pass providing a reasonnable amount of program parts which *possibly* match with the template is needed.

Several studies were achieved to find self-contained parts in a program, which can be potentially factored in a function. In the literature, program slicing is more often described as a program analysis technique, which aims to extract from a program the minimum part able to compute the value of a variable on a given program statement. Most often, the slicing techniques step the use-def chains from the output statement to built the slices [98, 70, 48, 90]. Metzger [76] introduces the notion of *computational confluence*, which aims to characterize the program parts which can potentially be factored in a function. As Weiser, Metzger basically computes computationally confluent parts by extracting strongly connected parts of the program dependence graph. Unfortunately, these methods do not take slice semantics into account, and may provide a too large amount of incorrect candidates.

In this chapter, we present a fast and efficient method to find in the program the candidate slices which *possibly* match with the template *w.r.t.* Herbrand equivalence. These slices will be checked by using the exact instantiation tests described in Chapters 7 and 8. Since the instantiation test is expensive, our slicing method must find all the right slices with a small amount of false positives. In addition, one must keep a reasonnable complexity. The slicing method presented in this chapter tries to find a balance between these two opposite constraints.

This chapter is organized as follows: Section 5.1, introduces the notions needed to understand our algorithm. In particular, we present the cartesian product over tree automata, which will be widely used in this thesis. Sections 5.3 and 5.4 present a formalization of the template recognition problem that is approximated in Section 5.5 to obtain the slicing algorithm. Section 5.6 executes the algorithm on a simple example providing a more intuitive explanation. Section 5.7 provides a formal complexity study of the algorithm, showing its linearity in the program size. Section 5.8 describes some related works about program slicing techniques. We finally discuss the advantages and the drawbacks of the algorithm in Section 5.9.

## 5.1 Background

This section introduces some important notions used in our slicing method. Particularly, we present the *tree automata*, an extension of finite-state automata to bound terms which is widely used in this thesis.

### 5.1.1 Approximated Reaching Definitions

According to definition 2.3 page 44, while executing a program $P$ on an input $I$, the *reaching definition* of the variable $v$ in the *operation* $\omega : s = f(\dots v \dots)$ is the last *operation* $\tau$ executed before $\omega$ that writes $v$:

$$\begin{array}{ll} \tau: & v = \dots \\ & \dots \\ \omega: & s = f(\dots v \dots) \end{array}$$

When such an operation exists, we write $\mathsf{RD}^I_\omega(v) = \tau$ ; otherwise, $\mathsf{RD}^I_\omega(v)$ is undefined. Denoting $\Omega^I$ the set of operations executed by $P$ on $I$, and writing $\omega : S$ to mean that the operation $\omega$ is an *instance* of the statement $S$ ; an *approximated reaching definition* is an application $\alpha$ dealing with statements:

$$\exists I\ \exists \omega, \tau \in \Omega^I \text{ with } \omega : S,\ \tau : T \text{ and } \mathsf{RD}^I_\omega(v) = \tau \implies T \in \alpha_S(v)$$

where $S$ and $T$ are statements of $P$ and $v$ is a variable read by $S$. In other words, an approximated reaching definition of a variable $v$ read by a statement $S$ is a set *containing* the statements $T$ such that there exists an *execution path* (characterized by $I$), where an *instance* of $T$ is the reaching definition of an *instance* of $S$.

In this thesis, we will focus on the approximated *scalar* reaching definitions defined in [1], page 683, that we will denote RDA. Basically, RDA handles array references $a[u(\vec{i})]$ as *scalar variables* $a$, and is unaware of conditionals and loop iteration numbers. For instance, figure 5.1 gives an example of program (a) and the corresponding approximated scalar reaching definitions (b).

| | | |
|---|---|---|
| $S_1$ | `s = a(0)` | $\mathsf{RDA}_{S_1}(a) = \emptyset$ |
| | `do i = 1,n` | |
| | `  if i mod 2 = 0 then` | $\mathsf{RDA}_{S_2}(s) = \{ S_1 , S_3 \}$ |
| $S_2$ | `  s = s + 1` | |
| | `  endif` | $\mathsf{RDA}_{S_3}(s) = \{ S_1 , S_2 , S_3 \}$ |
| $S_3$ | `  s = s + a(i)` | $\mathsf{RDA}_{S_3}(a) = \emptyset$ |
| | `enddo` | |
| $S_4$ | `r = s` | $\mathsf{RDA}_{S_4}(s) = \{ S_1 , S_3 \}$ |

(a) Original program (b) Approximated scalar reaching definitions

FIG. 5.1 – Approximated *scalar* reaching definitions

### 5.1.2 Tree Automata

Our slicing algorithm uses a powerful extension of the finite state word automata called *tree automata* [24], and defined as follows:

**Definition 5.1 (Tree automaton).** *A* tree automaton *is a tuple:*

$$\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$$

*where* $\Sigma$ *is a signature,* $Q$ *the set of states,* $Q_f \subset Q$ *the set of final states, and* $\Delta$ *a set of transition rules of the following type:*

$$f(q_1 \dots q_n) \longrightarrow q$$

*where* $n \geq 0$, $f/n \in \Sigma$ *and* $q, q_1, \dots, q_n \in Q$.
*Additionally,* $\longrightarrow$ *is a* congruence *over terms on* $\Sigma \cup Q$*:*

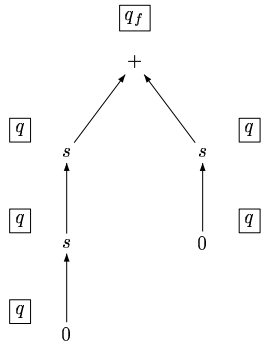
$$t \longrightarrow t' \Longrightarrow f(t_1 \dots t \dots t_n) \longrightarrow f(t_1 \dots t' \dots t_n)$$

*for each* $f \in \Sigma$, *and* $t, t', t_1 \dots t_n \in \mathcal{T}(\Sigma \cup Q)$.

Additionally, a term $t$ over $\Sigma$ is *accepted* by $\mathcal{A}$ if $t \xrightarrow{*} q_f$, where $q_f \in Q_f$. The set of terms accepted by $\mathcal{A}$ is called the *language recognized* by $\mathcal{A}$, and denoted by $\mathcal{L}(\mathcal{A})$.
For instance, consider the tree automaton $\mathcal{A}$ provided in the figure 5.2.(a). The term $+(s(0), s(s(0)))$ – represented in (b) by a tree – is accepted by $\mathcal{A}$. Indeed, (b) indicates the different steps of the recognition, putting the states reached at each node in a frame.

$$\Sigma = \{\ 0/0\ ,\ s/1\ ,\ +/2\ \}$$
$$Q = \{\ q\ ,\ q_f\ \},\ Q_f = \{\ q_f\ \}$$
$$\Delta = \left\{ \begin{array}{lcl} 0 & \longrightarrow & q \\ s(q) & \longrightarrow & q \\ +(q,q) & \longrightarrow & q_f \end{array} \right\}$$

(a) Tree automaton

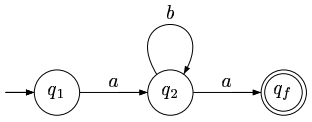



(b) A term recognized

FIG. 5.2 – A term recognized by a tree automaton

One can remark that tree automata have no initial states. Indeed, it is useless since the recognition starts from leaves (here 0) up to a final state.

Tree automata are a generalization of finite-state word automata. For example, the word automaton given in figure 5.3.(a) can be translated in the tree automaton given in (b), where unary functions $f/1$ represent alphabet letters $f$. A special 0-ary symbol $\Box$ has been added to start the recognition. Here, the word automaton recognizes *abba*, while the tree automaton recognizes the term $a(b(b(a(\Box))))$.




(a) Word automaton

$$\begin{array}{lcl} \Box & \longrightarrow & q_1 \\ a(q_1) & \longrightarrow & q_2 \\ b(q_2) & \longrightarrow & q_2 \\ a(q_2) & \longrightarrow & q_f \end{array}$$

(b) Corresponding tree automaton

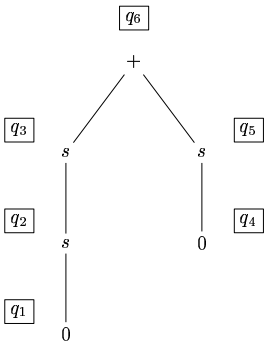
FIG. 5.3 – Tree automata generalize word automata

One may easily generalize this construction to every word automaton, identifying the set of word automata over the alphabet $\Sigma = \{a_1 \dots a_n\}$ to a subset of the tree automata over the signature $\{a_1/1 \dots a_n/1, \Box/0\}$.

We give now two important examples of tree automata, that will be used in our slicing algorithm. We first provide a tree automaton recognizing *exactly* a term, then we give a tree automaton recognizing *all the terms* over a signature $\Sigma$.

### Example A. Tree automaton recognizing exactly a term

Figure 5.4 gives an example of term (a), and provides the tree automaton recognizing it *exactly*. Basically, one associates to each sub-term a state $q_i$, then we trivially generate the transitions to construct a sub-term from its immediate sub-sub-terms (b).




(a) Term

$$\Sigma = \{\ 0/0\ ,\ s/1\ ,\ +/2\ \}$$
$$Q = \{\ q_1\ ,\ q_2\ ,\ q_3\ ,\ q_4\ ,\ q_5\ ,\ q_6\ \}$$
$$Q_f = \{\ q_6\ \}$$

$$\Delta = \left\{ \begin{array}{lcl} 0 & \longrightarrow & q_1 \\ s(q_1) & \longrightarrow & q_2 \\ s(q_2) & \longrightarrow & q_3 \\ 0 & \longrightarrow & q_4 \\ s(q_4) & \longrightarrow & q_5 \\ +(q_3, q_5) & \longrightarrow & q_6 \end{array} \right\}$$

(b) Corresponding tree automaton

FIG. 5.4 – Tree automaton recognizing *exactly* a term

### Example B. Tree automaton recognizing all terms over $\Sigma$

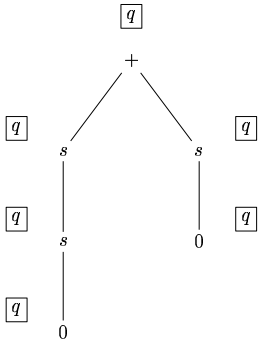
A tree automaton recognizing *all terms* over a signature $\Sigma$ is:

$$f(q \dots q) \longrightarrow q$$

for each $f \in \Sigma$, where $q$ is taken as final state. For instance, figure 5.5 gives the tree automaton recognizing all the terms over $\Sigma = \{0/0, s/1, +/2\}$ (a), and details the recognition for the term $+(s(s(0)), s(0))$ (b). We can easily show by induction on term depth that all the terms over $\Sigma$ are recognized.

$$\Sigma = \{\ 0/0\ ,\ s/1\ ,\ +/2\ \}$$
$$Q = \{\ q\ \},\ Q_f = \{\ q\ \}$$
$$\Delta = \left\{ \begin{array}{lcl} 0 & \longrightarrow & q \\ s(q) & \longrightarrow & q \\ +(q,q) & \longrightarrow & q \end{array} \right\}$$

(a) Tree automaton




(b) A term recognized

FIG. 5.5 – Tree automaton recognizing *all* the terms over $\Sigma$

## Cartesian product

Most of usual operations on word automata such as determinization, minimization or cartesian product extend naturally to tree automata [24]. Our slicing algorithm uses the cartesian product over tree automata, which allows basically to walk through two tree automata simultaneously.

**Definition 5.2 (Cartesian product).** *The* cartesian product *of two tree automata* $\mathcal{A}_1 = (\Sigma_1, Q_1, Q_{f1}, \Delta_1)$ *and* $\mathcal{A}_2 = (\Sigma_2, Q_2, Q_{f2}, \Delta_2)$ *is the tree automaton* $\mathcal{A}_1 \times \mathcal{A}_2 = (\Sigma, Q, Q_f, \Delta)$ *defined by:* $\Sigma = \Sigma_1 \cap \Sigma_2$, $Q = Q_1 \times Q_2$, $Q_f = Q_{f1} \times Q_{f2}$, *and* $\Delta = \Delta_1 \times \Delta_2$, *where:*

$$\Delta_1 \times \Delta_2 = \{\ f((q_1, q'_1) \ldots (q_n, q'_n)) \longrightarrow (q, q') \mid f(q_1 \ldots q_n) \longrightarrow q \in \Delta_1 \text{ and } f(q'_1 \ldots q'_n) \longrightarrow q' \in \Delta_2\ \}$$

Consider for instance the tree automata $\mathcal{A}$ (left) and $\mathcal{A}'$ (right):

$$\begin{array}{lcl} 0 & \longrightarrow & q \\ s(q) & \longrightarrow & q \\ +(q,q) & \longrightarrow & q_f \end{array} \qquad \begin{array}{lcl} 0 & \longrightarrow & q' \\ s(q') & \longrightarrow & q' \\ \times(q',q') & \longrightarrow & q'_f \end{array}$$

The cartesian product $\mathcal{A} \times \mathcal{A}'$ is then:

$$\begin{array}{lcl} 0 & \longrightarrow & (q,q') \\ s((q,q')) & \longrightarrow & (q,q') \end{array}$$

In the same manner as for word automata, the cartesian product of two tree automata $\mathcal{A}_1$ and $\mathcal{A}_2$ recognizes the intersection of their languages $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$. This interesting property will be used thereafter, making of the cartesian product the main mechanism of our slicing method.

## Quotient tree automata

We finish this section by introducing an important property of tree automata, which will be used to establish the conservativity of our method. We first define the notion of quotient tree automaton.

**Definition 5.3 (Quotient tree automaton).** *Consider a tree automaton* $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ *and* $\equiv$*, an equivalence relation over* $Q$*. The* quotient *of* $\mathcal{A}$ *by* $\equiv$ *is the tree automaton:*

$$\mathcal{A}/_{\equiv} = \left(\Sigma, Q/_{\equiv}, Q_f/_{\equiv}, \Delta/_{\equiv}\right)$$

*with:*

$$\Delta/_{\equiv} = \left\{ f([q_1] \dots [q_n]) \longrightarrow [q] \mid f(q_1 \dots q_n) \longrightarrow q \in \Delta \right\}$$

*where* $[q]$ *denotes the equivalence class of* $q$ w.r.t. $\equiv$.

Remark that this definition is *not* restricted to automata congruences, and can be applied to all possible equivalence relations over states. The following proposition establishes that a quotient tree automaton $\mathcal{A}/_{\equiv}$ recognizes *at least* the language of $\mathcal{A}$:

**Proposition 5.1 (Conservativity of the quotient).** *Consider a tree automaton* $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ *and* $\equiv$*, an equivalence relation on the states of* $\mathcal{A}$*. Then* $\mathcal{A}/_{\equiv}$ *recognizes* at least *the language of* $\mathcal{A}$*. More formally:*

$$\mathcal{L}(\mathcal{A}) \subset \mathcal{L}\left(\mathcal{A}/_{\equiv}\right)$$

*Proof.* Consider $t \in \mathcal{L}(\mathcal{A})$ obtained by a sequence of transitions:

$$t \longrightarrow_1 \dots \longrightarrow_i T_i(f(q_1 \dots q_n)) \longrightarrow_{i+1} T_i(q) \longrightarrow_{i+2} \dots \longrightarrow_p q_f$$

where $T_i(f(q_1 \dots q_n))$ denotes a term of $\mathcal{T}(\Sigma \cup Q)$ with a subterm $f(q_1 \dots q_n)$ to be rewrited in $q$. We can obviously have the same rewriting in $\mathcal{A}/_{\equiv}$ with:

$$t \longrightarrow_1 \dots \longrightarrow_i T_i(f([q_1] \dots [q_n])) \longrightarrow_{i+1} T_i([q]) \longrightarrow_{i+2} \dots \longrightarrow_p [q_f]$$

which leads to the result. □

Consider for example the following tree automaton $\mathcal{A}$, which recognizes the terms $\cos^n \sin^p \pi$ for $n, p \in \mathbb{N}$:

$$\begin{array}{lcl} \pi & \longrightarrow & S \\ \sin(S) & \longrightarrow & S \\ S & \longrightarrow & C \\ \cos(C) & \longrightarrow & C \\ C & \longrightarrow & q_f \end{array}$$

where $q_f$ is the final state. Consider the smallest equivalence relation verifying $S \equiv C$. The quotient of the state set is clearly $\{\{S, C\}, \{q_f\}\}$. Writing $Q = \{S, C\}$ and $Q_f = \{q_f\}$, this leads to write $\mathcal{A}/_{\equiv}$ as follows:

$$\begin{array}{lcl} \pi & \longrightarrow & Q \\ \sin(Q) & \longrightarrow & Q \\ Q & \longrightarrow & Q \\ \cos(Q) & \longrightarrow & Q \\ Q & \longrightarrow & Q_f \end{array}$$

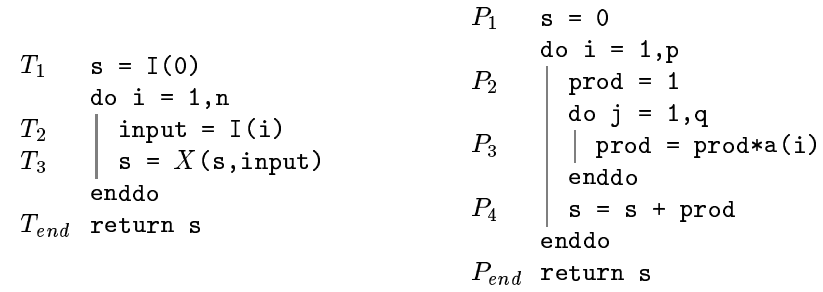
We have obviously $\mathcal{L}(\mathcal{A}) \subset \mathcal{L}(\mathcal{A}/_{\equiv})$. Since $\equiv$ is *not* an automaton congruence, $\mathcal{L}(\mathcal{A}/_{\equiv})$ also contains «bad» terms such as $\sin(\cos(\sin(\pi)))$.

## 5.2 Overview of the Method

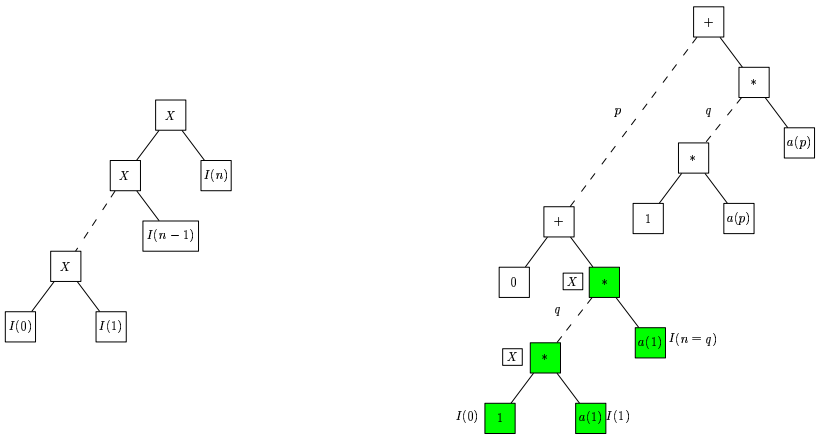
Figure 5.6 provides a simple example of template (a) to match in a program (b). The template and the program are assumed to be normalized with one operator per statement (at most). According to the definition of Herbrand equivalence, such a problem is equivalent to find all the instances of the template term (c) in the program term (d). Colored nodes represent a match with $X = *$, $n = q$, $I(0) = 1$ and $I(k) = a(1)$ for $1 \leq k \leq q$. This corresponds to the full slice:

$$\widehat{\mathcal{S}}^a_P(\langle P_3, 1, q\rangle) = \{\langle P_2, 1\rangle, \langle P_3, 1, 1\rangle \ldots \langle P_3, 1, q\rangle\}$$

Notice that other slices matches, including the whole program with $X = +$, $n = p$, $I(0) = 0$ and $I(k) = a(k)^q$.




(a) Template (b) Program



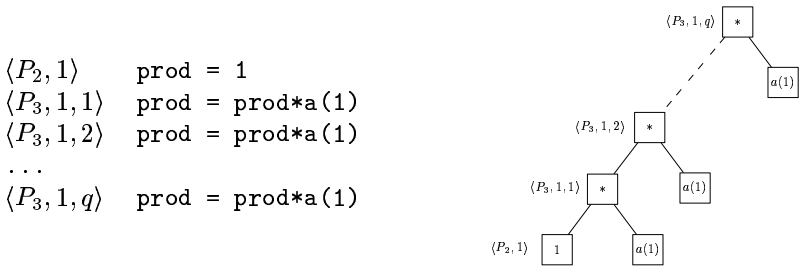

(c) Term computed by the template (d) Term computed by the program

FIG. 5.6 – Overview of the slicing

The main idea of our slicing method is to associate to the program and the template a *tree automaton* accepting their computed terms (Section 5.3). The recognition is then achieved by computing the cartesian product of the two tree automata (Section 5.4). Finally, Section 5.5 desribes our slicing method, based on an approximation of the construction rules. The approximation is shown to be *conservative*, ensuring the completeness of the method.

## 5.3 A Formalization of Template Matching

A convenient way to deal with the term computed by an algorithm is to built a tree automaton recognizing its computed term and only it, given an input. Consider the execution of the program slice $S = \widehat{\mathcal{S}}^a_P(\langle P_3, 1, q\rangle)$ detailed above. Figure 5.7.(a) provides the sequence of *operations* executed by the slice, and detail their contribution on the computed term (b). Each node is annoted by the operation that computes it.




(a) Execution trace of the (colored) slice (b) Computed term

FIG. 5.7 – Sub-terms computed by the operations of the colored slice

Handling each operation as a *state*, a tree automaton $\mathcal{A}_S(a)$ recognizing *exactly* the term computed by the slice is:

$$\begin{array}{lcl} 1 & \longrightarrow & \langle P_2, 1\rangle \\ *(\langle P_2, 1\rangle, a(1)) & \longrightarrow & \langle P_3, 1, 1\rangle \\ *(\langle P_3, 1, 1\rangle, a(1)) & \longrightarrow & \langle P_3, 1, 2\rangle \\ \ldots & & \\ *(\langle P_3, 1, q-1\rangle, a(1)) & \longrightarrow & \langle P_3, 1, q\rangle \end{array}$$

More generally, for each operation $\omega$:

$$\omega : \quad s = f(s_1 \ldots s_n)$$

where $s, s_1 \ldots s_n$ are scalar variables or array references, we generate the transition:

$$f(\mathsf{RD}^a_\omega(s_1) \ldots \mathsf{RD}^a_\omega(s_n)) \longrightarrow \omega$$

Taking the last operation as final state, we obtain a tree automaton $\mathcal{A}_S(a)$ called *exact tree automaton* of $S$ on $a$, that recognizes *exactly* the term computed while executing the slice on the input $a$.

In the general case, this tree automaton may have an infinite number of rules, which make them impossible to handle directly. Indeed, when the slice does not stops on $a$, $\mathcal{A}_S(a)$ is obviously infinite. Static control programs always execute the same (finite) sequence of operations for each input instance, providing a finite tree automaton independent of the input instance. However, static control *slices* may involve external variables (such as $n$ in the example) leading to a parametric number of rules, which are also impossible to handle directly.

In the same manner, we associate to the *template* a tree automaton recognizing *exactly* the terms computed by all its possible instances *in the program*. The construction follows the same principle than for the program, and allows template variables (here $X$) and template inputs (here $I$) to describe all possible expressions built with program operators.

As stated in background section, a tree automaton recognizing all possible terms over a signature $\Sigma$ is $f(q \dots q) \to q$ for each $f \in \Sigma$. Handling each program input $a(i)$ as a functional symbol, we have here:

$$\Sigma = \{a(0)/0 \ \dots \ a(p)/0 \ , \ 0/0 \ , \ 1/0 \ , \ +/2 \ , \ */2 \ \}$$

which leads to the following tree automaton:

$$\begin{array}{lcll} a(k) & \longrightarrow & q & \text{for each } 0 \leq k \leq p \\ 0 & \longrightarrow & q & \\ 1 & \longrightarrow & q & \\ +(q,q) & \longrightarrow & q & \\ *(q,q) & \longrightarrow & q & \end{array}$$

Consequently we can recognize all the instances of $X(x, y)$ by adding the *input transitions* $x \to q$ and $y \to q$. More generally, from an operation:

$$\omega : \quad s = X(s_1 \dots s_n)$$

where $s, s_1 \dots s_n$ are scalar or array references, we generate the transitions:

$$\begin{array}{lcll} \mathsf{RD}^I_\omega(s_1) & \longrightarrow & \omega & \\ \dots & & & \\ \mathsf{RD}^I_\omega(s_n) & \longrightarrow & \omega & \\ f(\omega \dots \omega) & \longrightarrow & \omega & \text{for each } f \in \Sigma \end{array}$$

Similarly, the *template inputs* are handled as 0-ary template variables. As a consequence, from an operation with a template input:

$$\omega : \quad s = I(i)$$

we generate the transitions:

$$f(\omega \dots \omega) \longrightarrow \omega \quad \text{for each } f \in \Sigma$$

Figure 5.8 summarizes the construction rules of the exact tree automaton that can be applied to templates and programs, for a given input $I$. Rule 1 can be applied to templates and programs, whereas rule 2 is restricted to templates. The transitions $\mathsf{RD}^I_\omega(s_i) \to \omega$ are called *input transitions*. Since the transitions $f(\omega \dots \omega) \to \omega$ loop on $q$, we call them *looping transitions*. Additionally, there is no rule for template inputs, since they are assimilated to 0-ary template variables.

Consider now the matching problem between the template $T$ and the slice $S$: $T \stackrel{?}{=} S$. We have the following equivalences:

$$\begin{array}{rcl} S \text{ is an instance of } T \text{ } w.r.t. \equiv_{\mathcal{H}} & \Longleftrightarrow & \forall a : \ \mathcal{T}(S) \in \mathcal{L}(\mathcal{A}_T(a)) \\ & \Longleftrightarrow & \forall a : \ \{\mathcal{T}(S)\} \cap \mathcal{L}(\mathcal{A}_T(a)) \neq \emptyset \\ & \Longleftrightarrow & \forall a : \ \mathcal{L}(\mathcal{A}_S(a)) \cap \mathcal{L}(\mathcal{A}_T(a)) \neq \emptyset \end{array}$$

1. For each *operation*:

$$\omega : \ s = f(s_1 \dots s_n)$$

emit the transition:

$$\boxed{f(\mathsf{RD}^I_\omega(s_1) \dots \mathsf{RD}^I_\omega(s_n)) \longrightarrow \omega}$$

2. For each *operation* with a template variable:

$$\omega : \ s = X(s_1 \dots s_n)$$

emit the transition:

$$\boxed{\begin{array}{lcl} \mathsf{RD}^I_\omega(s_1) & \longrightarrow & \omega \\ \dots & & \\ \mathsf{RD}^I_\omega(s_n) & \longrightarrow & \omega \\ f(\omega \dots \omega) & \longrightarrow & \omega \end{array}}$$

for each operator $f$ used in the program.

FIG. 5.8 – Construction rules of the *exact tree automaton*

A classical solution to check the intersection emptiness of $\mathcal{L}(\mathcal{A}_T(a))$ and $\mathcal{L}(\mathcal{A}_S(a))$ is to construct the cartesian product $\mathcal{A}_T(a) \times \mathcal{A}_S(a)$, and to verify that the final state is reachable [24]. As a consequence, we have:

$$\boxed{S \text{ is an instance of } T \iff \forall a : \text{the final state of } \mathcal{A}_T(a) \times \mathcal{A}_S(a) \text{ is reachable}}$$

This provides an *important expression of the matching problem w.r.t.* $\equiv_{\mathcal{H}}$ that will used in this chapter, and in the instantiation test described in Chapter 8. The following section extends this formalization to the *template recognition problem.*

## 5.4 Application to Template Recognition

Consider again the template recognition problem presented in figure 5.6. The following lemma states that the exact tree automaton of a *full slice w.r.t.* an operation $\omega$ is the tree automaton of the program where $\omega$ is taken as final state.

**Lemma 5.1.** *Consider an operation $\omega$ obtained while executing a program $P$ on a given input $a$. Writing $\mathcal{A}_P(a)[\omega]$ the tree automaton of $P$ where $\omega$ is taken as final state, we have:*

$$\mathcal{L}(\mathcal{A}_P(a)[\omega]) = \mathcal{L}(\mathcal{A}_{\widehat{\mathcal{S}}^a_P(\omega)}(a)) = \{\mathcal{T}(\widehat{\mathcal{S}}^a_P(\omega))\}$$

*Proof.* By construction, the state $\omega$ recognizes exactly the term computed by the operation $\omega$, and is reached by operations defining the full slice and only them (see figure 5.8, rule 1). The result follows. □

Assume now that we have to discover in $P$ all the full slices $\widehat{\mathcal{S}}^a_P(\omega)$ instances of $T$. Combining the lemma and the formulation given in the previous section, $\widehat{\mathcal{S}}^a_P(\omega)$ is an instance of $T$ iff the final state of $\mathcal{A}_T(a) \times \mathcal{A}_P(a)[\omega]$ is reachable.

A naive solution would be to construct the cartesian product for each operation $\omega$ executed by the program. Since $\mathcal{A}_P(a)[\omega]$ is $\mathcal{A}_P(a)$ where $\omega$ is taken as final state, it is enough to *built* $\mathcal{A}_T(a) \times \mathcal{A}_P(a)$ *and to collect the program operations* $\omega$ *together with* $\langle T_{end}, \rangle$ (states $(\langle T_{end}, \rangle, \omega)$). Since the instantiation tests presented in Chapters 7 and 8 just need the last operation $\omega$ of the slice, this information is enough for our purpose.

## 5.5 An Approximation

This section presents our *slicing method* which is a (conservative) approximation of the formulation given in the two previous sections. The algorithm is provided, and followed by a proof of conservativity that uses quotient of tree automata.

One can remark that we cannot compute $\mathcal{A}_A(I)$ for a general input $I$, since it requires an exact dataflow information, which is not computable for arbitrary programs. We choose to relax the dataflow information, replacing the *exact* reaching definitions $\mathsf{RD}^I_\omega(s)$ by *approximated* reaching definitions dealing at *statement* level $\mathsf{RDA}_S(s)$. We then obtained an *approximated tree automaton*, whose construction algorithm is given in figure 5.9. Since data-flow constraints are relaxed, the approximated tree automaton no longer depends on inputs, and can be denoted by $\mathcal{A}_A$.

---

**Algorithm** *Build_Automaton*

---

**Input:** *The template or the program.*
**Output:** *The corresponding* approximated *tree automaton.*

1. Associate a new state to each assignment *statement.*
2. For each *statement*:
$$S: \ s = f(s_1 \dots s_n)$$
add the transitions:
$$\boxed{f(S_1 \dots S_n) \longrightarrow S}$$
for each $S_i \in \mathsf{RDA}_S(s_i)$
3. For each *statement* with a template variable:
$$S: \ s = X(s_1 \dots s_n)$$
add the transitions:
$$\boxed{\begin{array}{l} S_1 \longrightarrow S \\ \dots \\ S_n \longrightarrow S \end{array}}$$
for each $S_i \in \mathsf{RDA}_S(s_i)$ (*input transitions*)
And:
$$\boxed{f(\omega \dots \omega) \longrightarrow \omega}$$
for each operator $f$ used in the program (*looping transitions*)

---

FIG. 5.9 – *Build_Automaton*

Since there is a finite number of statements, the approximated tree automata associated to the template ($\mathcal{A}_T$) and the program ($\mathcal{A}_P$) will be always finite. As a consequence, the construction algorithm will always terminate.

Following the formalization given in the previous section, it remains now to compute the cartesian product $\mathcal{A}_T \times \mathcal{A}_P$ and to collect the program *statements* together with the final state of the template to have a the candidates slices. This is stated in the algorithm described in figure 7.10.

---

**Algorithm** *Output_Slices*

---

**Input:** $\mathcal{A}_T$ *and* $\mathcal{A}_P$*, template and program* approximated *tree automata.*
**Output:** $\{O_1 \dots O_n\}$*, the output statements of each candidate slice.*

1. Compute the Cartesian product $\mathcal{A} = \mathcal{A}_T \times \mathcal{A}_P$.
2. Mark the nodes with a final state of $\mathcal{A}_T$, and emit the $\mathcal{A}_P$ part of marked states.

---

FIG. 5.10 – *Output_Slices*

We show now that our slicing algorithm provides an over-approximation of the matching slices. The following (technical) lemma exhibits a relation between the exact and the approximated tree automata of $A$. It is a consequence of the definition of $\mathcal{A}_A$ which will be used to prove lemma 5.3.

**Lemma 5.2.** *Consider an algorithm $A$, and denote $\mathcal{A}_A(I)$ its tree automaton for a given input $I$, and $\mathcal{A}_A$ its approximated tree automaton. Defining an equivalence relation $\equiv$ over the operations of $A$ by:*

$$\langle S_1, \vec{i_1} \rangle \equiv \langle S_2, \vec{i_2} \rangle \Longleftrightarrow S_1 = S_2$$

*We have:*

$$\mathcal{A}_A = \cup_I \mathcal{A}_A(I) \Big/ \equiv$$

*Proof.* Consider again the statement $\langle S, \vec{i} \rangle : s = f(s_1 \dots s_n)$, and denote by $D_S^I$ the iteration domain of the statement $S$ while executing $A$ on the input $I$. By construction, the rules of $\cup_I \mathcal{A}_A(I)$ leading to $S$ can be written:

$$f(\mathsf{RD}^I_{\langle S,\vec{i}\rangle}(s_1) \dots \mathsf{RD}^I_{\langle S,\vec{i}\rangle}(s_n)) \longrightarrow \langle S, \vec{i} \rangle$$

where $\vec{i}$ describes the iteration domain $D_S^I$ of $S$, and $I$ describes all possible inputs. Writing $\mathsf{Stmt}\langle S, \vec{i} \rangle = S$ and $S^I_{k,\vec{i}} = \mathsf{Stmt}\ \mathsf{RD}^I_{\langle S,\vec{i}\rangle}(s_k)$ the statement of the $k$-th argument of $f$, the corresponding rules of $\cup_I \mathcal{A}_A(I)/\equiv$ are:

$$f(S_1 \dots S_n) \longrightarrow S$$

where $S_k \in \cup_I \cup_{\vec{i}} S^I_{k,\vec{i}}$. Additionally, the approximate reaching definition of the $S_i$ can be written:

$$\mathsf{RDA}_S(S_i) = \cup_I \cup_{\vec{i}} S^I_{i,\vec{i}}$$

which leads to emit the same rules than for $\cup_I \mathcal{A}_A(I) \Big/ \equiv$. □

As stated in Section 5.1.1, the approximated reaching definition RDA provides the right data-flow information, among false one, since it is unaware of conditional and iterations executed by in loops. As a consequence, the approximated tree automaton recognizes the term computed by the algorithm, among «parasitic» terms. This is stated by the following lemma, which is an immediate consequence of the previous lemma and the conservativity of the quotient stated in Proposition 5.1.

**Lemma 5.3.** *Consider an algorithm $A$, and denote $\mathcal{A}_A(I)$ its tree automaton for a given input $I$, and $\mathcal{A}_A$ its approximated tree automaton. Then $\mathcal{A}_A$ recognizes the terms recognized by each $\mathcal{A}_A(I)$. More formally:*

$$\mathcal{L}(\mathcal{A}_A(I)) \subset \mathcal{L}(\mathcal{A}_A)$$

*For each input $I$ of $A$.*

A conservative approximation of the instantiation test between a slice $S$ and a template $T$ would be to check if the two approximated tree automata $\mathcal{A}_T$ and $\mathcal{A}_S$ can accept the same term. Indeed, if $S$ is an instance of $T$ *w.r.t.* Herbrand equivalence, they recognize the same term, which will be in the language of their exact rewrite system, but also in the language of their approximated tree automaton (previous lemma). Thus their intersection will be non-empty. This would ensure the detection of the matching. The following proposition states formally this property, and provides a proof.

**Proposition 5.2 (Conservativity).** *Consider a template $T$ and a program slice $S$ with approximated tree automata $\mathcal{A}_T$ and $\mathcal{A}_S$. Thus:*

$$S \text{ is an instance of } T \implies \mathcal{L}(\mathcal{A}_T) \cap \mathcal{L}(\mathcal{A}_S) \neq \emptyset$$

*Proof.* Assuming $S$ is an instance of $T$, we have: $\mathcal{L}(\mathcal{A}_S(I)) \cap \mathcal{L}(\mathcal{A}_T(I)) \neq \emptyset$. According to previous lemma, $\mathcal{L}(\mathcal{A}_S(I)) \subset \mathcal{L}(\mathcal{A}_S)$ and $\mathcal{L}(\mathcal{A}_T(I)) \subset \mathcal{L}(\mathcal{A}_T)$. The result follows. □

## 5.6 An Example

This section executes our slicing algorithm on a simple example, and detail all the steps described above. We also provide a more intuitive explanation of the algorithm.

Consider the template matching problem given in figure 5.11. The template (a) matches the reductions ; its input array has been substituted by a 0-ary template variable $X_I$ which will match the program part computing the corresponding input values. The program (b) computes $1+2+\ldots+k$ by using a counter c. The slice with the statements $S_1$ and $S_2$ matches with $X(x,y) = +(x,y)$ and $X_I = \mathsf{c}$ ; while the slice with $C_1$ and $C_2$ matches with $X(x,y) = +(x,y)$ and $X_I = 1$. Sub-figures (c) and (d) provides the approximated reaching definitions which will be used to compute the approximated tree automata $\mathcal{A}_T$ and $\mathcal{A}_P$.

```
T1    s = 0
      do i = 1,n
TI    | input = XI
T2    | s = X(s,input)
      enddo
Tend  return s
```

(a) Template

```
S1    s = 0
C1    c = 0
      while c <= k
S2    | s = s + c
C2    | c = c + 1
      enddo
```

(b) Program

$\mathsf{RDA}_{T_2}(s) = \{T_1, T_2\}$
$\mathsf{RDA}_{T_2}(input) = \{T_I\}$

$\mathsf{RDA}_{T_{end}}(s) = \{T_1, T_2\}$

(c) Template RDA

$\mathsf{RDA}_{S_2}(s) = \{S_1, S_2\}$
$\mathsf{RDA}_{S_2}(c) = \{C_1, C_2\}$

$\mathsf{RDA}_{C_2}(c) = \{C_1, C_2\}$

(d) Program RDA

FIG. 5.11 – A template recognition problem

### Construction of $\mathcal{A}_T$ and $\mathcal{A}_P$

Applying *Build_Automaton* to the template and the program, we obtain the tree automata given in figure 5.12. Rules emitted for each statement are separated by an empty line.

In the template automaton, rules 2,3,4 allow to build for $X_I$ all possible expressions using program operators (0, 1 and +). Rules 5,6,7 are input transitions built from RDA, and rules 8,9,10 are looping transitions for $X$, allowing to build all terms using program operators.

In the program automaton, each packet of rule is obtained by apply the step 2 of *Build_Automaton* that enumerates all combinations of approximated reaching definitions.

$$
\begin{array}{rcl}
0 & \xrightarrow{1} & T_1 \\
\\
0 & \xrightarrow{2} & T_I \\
1 & \xrightarrow{3} & T_I \\
+(T_I, T_I) & \xrightarrow{4} & T_I \\
\\
T_1 & \xrightarrow{5} & T_2 \\
T_2 & \xrightarrow{6} & T_2 \\
T_I & \xrightarrow{7} & T_2 \\
0 & \xrightarrow{8} & T_2 \\
1 & \xrightarrow{9} & T_2 \\
+(T_2, T_2) & \xrightarrow{10} & T_2 \\
\\
T_2 & \xrightarrow{11} & T_{end}
\end{array}
$$

$$
\begin{array}{rcl}
0 & \xrightarrow{1} & S_1 \\
\\
0 & \xrightarrow{2} & C_1 \\
\\
+(S_1, C_1) & \xrightarrow{3} & S_2 \\
+(S_1, C_2) & \xrightarrow{4} & S_2 \\
+(S_2, C_1) & \xrightarrow{5} & S_2 \\
+(S_2, C_2) & \xrightarrow{6} & S_2 \\
\\
+(C_1, 1) & \xrightarrow{7} & C_2 \\
+(C_2, 1) & \xrightarrow{8} & C_2
\end{array}
$$

FIG. 5.12 – Approximated tree automata of the template (left) and the program (right)

**Construction and Analysis of $\mathcal{A}_T \times \mathcal{A}_P$**

It remains now to analyze $\mathcal{A}_T$ and $\mathcal{A}_P$ to find the candidates slices. Intuitively, the idea is to step simultaneously the two automata starting from *identical leaves* (0 or 1), and firing the reached transitions with the *same operators*. When the final state of the template is reached, the corresponding program state corresponds to the *output statement* of the slice. This information is enough for the following step of the recognition process (the instantiation test). However, the whole slice could be computed by taking all program states along the paths from leaves to final state.

For the sake of clarity, we do not provide $\mathcal{A}_T \times \mathcal{A}_P$. Instead, figure 5.13 provides a graphical representation of the rules of $\mathcal{A}_T \times \mathcal{A}_P$ leading to detect the slice with statements $S_1$ and $S_2$.

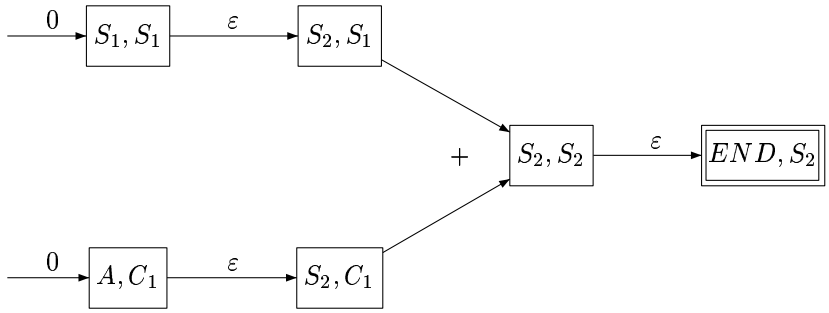



FIG. 5.13 – Rules of $\mathcal{A}_T \times \mathcal{A}_P$ leading to detect a candidate slice

Although the slicing method is conservative and provides all candidate slices *w.r.t.* Herbrand-equivalence, several false positives are yield. The false detections are due to the approximation made while computing the reaching definition RDA. All variables are handled as scalars, which induces false dependences, and thus false transitions. In addition, the loop iteration number, and the predicates in conditionals are ignored. An experimental study measuring the accuracy, and highlighting the main causes of false detections can be found in Chapter 10, Section 10.2.1.

## 5.7 Complexity Issues

We focus now on the theoretical complexity of the slicing method. This study is completed by experimental results on SpecFP 2000 [54] and Perfect Club benchmarks [39] which can be found in Chapter 10 Section 10.2.1.

Let us assume that the significant operation of our algorithm is the creation of a transition, and consider a program $P$, with a statement:

$$S: \quad s = f(s_1 \ldots s_n)$$

where $s, s_1 \ldots s_n$ are scalar variables or array references. The number of transitions created by *Build_Automaton* is:

$$|\mathsf{RDA}_S(s_1)| \times \ldots \times |\mathsf{RDA}_S(s_n)|$$

Thus the total number of transitions can be written $P \times d^a$ where $d$ is the average number of RDA by reference, and $a$ denotes the average arity of program operators. Experimental results on SpecFP 2000 has shown that in average $d = 4$ and $a = 2$. Indeed, $f$ is more often standard arithmetic operators such as $+$, $-$, $\times$ or $/$. Thus, the average complexity of *Build_Automaton* is $16P = \mathcal{O}(P)$.

In the worst case, the number of transitions of $\mathcal{A}_T \times \mathcal{A}_P$ is $T(\mathcal{A}_T) \times T(\mathcal{A}_P)$ where $T(\mathcal{A})$ denotes the number of transitions of $\mathcal{A}$. This case occurs when $T$ and $P$ uses only one operator, which never occurs in real life examples. Thus the worst-case complexity can be written:

$$(d^a)^2 T \times P = 256T \times P = \mathcal{O}(P)$$

As a consequence, the slicing method is linear in the program size. Furthermore, experimental results on SpecFP 2000 and Perfect Club benchmarks confirms the linearity and demonstrates the scalability of our method.

## 5.8 Related Work

This section present several significative approaches to locate functions within a program. We first focus on slicing-based approaches [66, 23, 101], then we present two methods based on metrics [11], and exact syntactic matching on program source [84].

The original definition of a program slice [98] involves *slicing criterion*, which is a pair $(p, V)$, where $p$ is a program point and $V$ a subset of program variables. A program slice on the slicing criterion $(p, V)$ is a subset of program statements that preserves the behavior of the original program at the program point $p$ with respect to the program variables in $V$. There exists several extensions of this definition to find *functions* among source code including the work of Lanubile and Visaggio [66], Cimitile *et al.* [23] and Wilde *et al.* [101], that we present thereafter.

Lanubile and Visaggio [66] added the set of input variables to the slice criterion. They introduced the notion of *transform slice*, as the slice that computes the values of the output variables at a given point, from the values of the input variables. Basically, the computation of the slice stops as soon as statement that defines values for the input variables are included in the slice.

Cimitile *et al.* [23] defined a method to identify slices verifying given pre-conditions and post-conditions. They first compute a symbolic execution of the program, which assign to each statement its pre-condition, then they use a theorem prover to extract the slices. They need user interaction to associate post-condition variables to program variables. Moreover, as the problem of finding invariant assertions is in general *undecidable*, symbolic execution can require user interaction in order to prove some assertions and assert some invariants. No practical evaluation of their method, or theoretical study of complexity is given, but their method seems to be costly. Moreover, the need of user interaction makes the method inappropriate in a fully automatic framework.

Wilde *et al.* [101] proposes a method based on test-cases to locate specific functionalities in code. They use deterministic and probabilistic techniques to analyze the traces resulting from the program execution. In general, this method outputs too large components, including more functionalities than the one sought.

Kontogiannis *et al.* [11] propose an approach to detect clones within large programs. They describe a dynamic programming based approach which focus on whole sequence of instructions (begin-end block and functions) and allow the detection of similar blocks by using an edition distance. The distance between a pair of blocks is defined as the least sequence of insert and deletes to rewrite one block by another. The underlying hypothesis is that pairs with a small distance are likely to be clones caused by copy-paste. This method is purely syntactic, and does not take into account of the program semantics. Moreover, the variations supported seems to be limited to basic organization variations. However, distance measures and heuristic metrics seems to be a low cost method interesting to explore an a future work.

Paul and Prakash [84] proposes an extension of `grep` to find *program patterns* in source code. They use a pattern language with wildcards on syntactics entities *e.g.* declaration, type, variable, function, expression and statement which allow to find patterns with specific sequences and imbrications of control structures. For example, here is pattern used to find the maximum in an array of integers:

```
{*
  @[while|dowhile|for] {*
    if($v_1[#] > $v_2) {*
      $v_2 = $v_1[#]
    *}
  *}
*}
```

The wildcard `{* *}` means a statement with an arbitrary nesting depth. The expression `@[while|dowhile|for]` means either `while`, `dowhile` or `for`. `$v_{name}` means a variable labeled by `name` and `#` means an expression. Their algorithm produces and interprets a Code Pattern Automaton (CPA), which traverse the program's AST according to the pattern, and decides if it is an instance or not. They argue that the complexity of their algorithm is $O(n^2)$ with $n$ the number of AST nodes. Obviously, their approach is limited to one programming language, and forces the user to make strong assumptions in the implementation of patterns. It seems this approach cannot handle many other program variations than variable renaming (`$v_{name}`) and garbage code (`{* *}`). Thus candidate slices will be quite limited. Moreover, their algorithm is more expensive than ours.

## 5.9 Discussion

We have described a new and efficient slicing method, able to find in *any* program the potential instances of a given template. Our approach has been validated by recognizing BLAS 1 and 2 functions in several kernels of the SpecFP and Perfect Club benchmarks. Experimental results has established that the amount of bad detections remains reasonnable (about 36 % of detections). Moreover, all Herbrand-equivalent slices were found, which confirms the result of conservativity stated by Proposition 5.2. Particularly, organization, data-structure and control variations are detected. We provide thereafter an intuitive explanation.

**Organization variations** Our algorithm works on the def-use graph, which avoids the artificial precedence constraints due to the text representation of the program. This allows our algorithm to handle legal permutations and garbage code. Moreover, our method compares two by two the operators used in the template and the program handling the temporary assignments `tmp = a` as an $\varepsilon$-transition.

**Data structure variations** Scalar RDA applies an aggressive normalization by handling each variable as an array. The resulting data-flow information is over-approximated and thus contains the exact one. The flow-dependence are stepped without taking types of variables into account, allowing to handle data-structure variations.

**Control variations** A control variation may affect the execution order of operations (tiling), or the program text (unroll), but the final expressions computed are still the same, leading the def-use chains to describe the same sequence of operators. As a consequence, all control variations which do not affect the final expression computed are handled. This includes most of loop transformations (tiling, skewing, loop fusion, interchange, etc).

**Semantic variations** Our method is able to detect all slices computing the same expression than the template, *syntactically*. Semantic variation changes the syntax of the computed expression, and make it impossible to detect it. Semantic variations represents all correct but undetected slices in the program.

In an other hand, we provide a complexity study establishing that our slicing method is linear in the program size. This theoretical result has been confirmed by the experimental results, which provides a reasonnable execution time on real-life applications (about 6 minutes on a kernel with 1264 lines of code). This demonstrates the scalability of our approach.

Notice that other efficient methods with a different approximation level could be used, it will not affect the correctness and the completness of the approach, according to Proposition 2.3, page 54. Possible extensions may include type of variables (scalar, arrays), or more accurate data-flow information.

Once the candidates slices are found, it remains to check whether they are effectively instances of the template. This corresponds to step 2 in the decomposition provided in Chapter 2, Proposition 2.3. The next chapters propose two different instantiation tests to cope with this problem. They follow the principle of the equivalence test of Barthou *et al.* [13], which we describe in the next chapter.

# Chapter 6

# Program Equivalence

The previous chapter presents an algorithm able to find within a program the slices which *possibly* match a given template. To ensure the correctness of the substitution, a *may* test is obviously not enough, and must be completed by an exact, *must* matching test.

Before describing our two matching procedures in the next chapters, we propose to study here the particular case where the template to look for is actually not a template, but a simple program. This difficult issue is better known as *semantic equivalence*, a well-known undecidable problem, which has been intensively addressed in several domains such as program verification or program slicing.

Necula [80] proposes an equivalence test in the scope of program validation, and more precisely in translation validation for the optimization passes of compilers. He basically rely on a symbolic evaluation which produces equivalence constraints, that needs a theorem prover to be solved. His framework is based on the gcc front-end, and has been able to validate optimized transformations on the gcc compiler and the linux kernel. Zuck *et al.* [111] proposes another theoretical framework for translation validation of optimizing compilers. Their tool, VOC, works on a low-level representation of the program near to assembly language, and rely on a theorem prover to check proof obligations. Further works aims to detect software clones and code duplication [11, 65]. They most often rely on a metric based on edition distance, and syntactic properties of programs.

In this chapter, we present an algorithm due to Barthou *et al.* [13] to semi-decide whether two static-control programs are Herbrand-equivalent. The method rely on a simultaneous symbolic execution of both programs, and is able to detect all the variations raised in introduction, except semantics variations. This method is not able to decide the *matching problem*. However, it provides interesting ideas, which will be exploited within our two instantiations tests, described in the two next chapters.

This chapter is organized as follows: Section 6.1 presents some preliminary notions needed to understand the algorithm, including exact data-flow analysis, and Presburger relations. Section 6.2 provides a motivating example which will be addressed during the algorithm description. Section 6.3 sums up the main steps of the algorithm, then Sections 6.4 and 6.5 describe the two steps of the algorithm. Section 6.6 presents some approaches related to the semantics equivalence. We finally provide a discussion in Section 6.7.

## 6.1 Background

### 6.1.1 Exact Reaching Definitions

The equivalence test described in this chapter needs an exact dataflow information in the meaning defined in Chapter 2. Recall that the *reaching definition* of a variable $v$ read by an operation $\langle S, \vec{i} \rangle$ is the last operation $\langle T, \vec{j} \rangle$ executed before $\langle S, \vec{i} \rangle$ which write $v$. It is usually denoted $\mathsf{RD}_{\langle S,\vec{i}\rangle}(v)$. Given the following example:

```
S1  s = a(0)
    do i =1,n
S2  | s = s + a(i)
    enddo
```

The reaching definition of `s` read by $\langle S_2, \vec{i} \rangle$ is:

$$\mathsf{RD}_{\langle S_2,\vec{i}\rangle}(s) = \begin{cases} i = 1 : & \langle S_1, \rangle \\ 2 \leq i \leq n : & \langle S_2, i-1 \rangle \end{cases}$$

Even if the reaching definition computation is undecidable in general, Feautrier provides a solution for the particular case of static control programs [43], that we describe below.

#### Sequencing predicate

Consider two *operations* $\langle S_i, \vec{i} \rangle$ and $\langle S_j, \vec{j} \rangle$ of a static control program. The part of $\vec{i}$ with loop counters common to $S_i$ and $S_j$ is denoted by $\vec{i}_{|i,j}$. In the following example:

```
do i = ...
| do j = ...
| | do k = ...
| | | S1
| | enddo
| | S2
| enddo
enddo
```

We would have $[i, j, k]_{|1,2} = [i, j]$, since the common loop nest of $S_1$ and $S_2$ is built from loops over $i$ and $j$.

The following proposition shows that the execution order between two operations $\langle S_i, \vec{i} \rangle$ and $\langle S_j, \vec{j} \rangle$ of a static control program can be expressed by using the lexicographic order between $\vec{i}$ and $\vec{j}$. A more detailed explanation can be found in [43].

**Proposition 6.1 (Sequencing predicate).** *Consider two operations $\langle S_i, \vec{i} \rangle$ and $\langle S_j, \vec{j} \rangle$ of a static control program. $\langle S_i, \vec{i} \rangle$ is executed before $\langle S_j, \vec{j} \rangle$ iff the following condition is achieve:*

$$\vec{i}_{|ij} \ll \vec{j}_{|ij} \text{ or } \left(\vec{i}_{|ij} = \vec{j}_{|ij} \text{ and } S_i \text{ is before } S_j \text{ in the program text}\right)$$

*Where $\ll$ denotes the lexicographic order between integer vectors.*
*In this case, we write $\langle S_i, \vec{i} \rangle \prec \langle S_j, \vec{j} \rangle$*

Consider now an *operation* $\langle S_i, \vec{i} \rangle$ of the program, which reads an array $a$:

$$S_i : \ \ldots = \ldots a[u(\vec{i})] \ldots$$

In order to compute the last operation writing $a[u(\vec{i})]$ before $\langle S_i, \vec{i} \rangle$, the idea is to compute from each statement assigning $a$, its last instance executed *before* $S_i$. These operations are called *direct dependences*. The exact reaching definition of $a$ is then obtained by computing the maximum of these operations in the meaning of the execution order $\prec$. These two steps are described thereafter.

### Computing the Direct Dependences

The first step is to compute from each statement writing $a$ the last operation executed before $\langle S_i, \vec{i} \rangle$. Consider a statement $S_j$ assigning $a$:

$$S_j : \ a[v(\vec{j})] = \ldots$$

In order to compute its last instance executed before $\langle S_i, \vec{i} \rangle$, Feautrier proposes to write the following integer program:

$$\underline{\max}_{\ll} \quad \boxed{\vec{j}}$$

$$\underline{\text{s.t.}}$$

- $\langle S_j, \vec{j} \rangle$ is a valid operation:

$$\boxed{\vec{j} \in D_j}$$

- $S_j$ write the array cell read by $S_i$:

$$\boxed{u(\vec{i}) = v(\vec{j})}$$

- $\langle S_j, \vec{j} \rangle$ is executed before $\langle S_i, \vec{i} \rangle$:

$$\boxed{\langle S_j, \vec{j} \rangle \prec \langle S_i, \vec{i} \rangle}$$

Where $D_i$ denotes the iteration domain of the statement $S_i$. Unfortunately, it is not a true integer program since the definition of $\prec$ involves lexicographic order, which involves disjunctions $\underline{\text{or}}$. Indeed, recall that the definition of $\ll$ is:

$$\begin{pmatrix} i_1 \\ \vdots \\ i_n \end{pmatrix} \ll \begin{pmatrix} j_1 \\ \vdots \\ j_n \end{pmatrix} \iff \begin{array}{ll} & i_1 < j_1 \\ \underline{\text{or}} & (i_1 = j_1 \text{ and } i_2 < j_2) \\ \underline{\text{or}} & (i_1 = j_1 \text{ and } i_2 = j_2 \text{ and } i_3 < j_3) \\ & \vdots \\ \underline{\text{or}} & (i_1 = j_1 \text{ and } \ldots \text{ and } i_n < j_n) \end{array}$$

Feautrier handles this problem by building one integer program by clause of the lexicographic order. The intuitive explanation is that the disjunctions of the sequencing predicate leads to consider the search domain as a finite union of disjoint polyhedra $Q_1 \cup \ldots \cup Q_n$ (one polyhedra by clause). And since:

$$\max_{\ll}(Q_1 \cup \ldots \cup Q_n) = \max_{\ll}\left\{\max_{\ll} Q_1 \ldots \max_{\ll} Q_n\right\}$$

we can consider the $\max_{\ll} Q_i$ as a direct dependence, whose $\max_{\ll}$ will be computed during the next step. Since $Q_i$ may depends on parameters, usual solvers such as CPLEX [57] or LP_SOLVE [15] cannot be used. Feautrier has also designed an algorithm to solve integer programs whose search domains depends on a parameter, this algorithm has been implemented in the PIP tool [42].

Since the lexicographic maximum may depend on the parameter values, PIP presents its results as selection trees called QUASTs (for QUasi-Affine Selection Tree). The syntax of a quast is naturally defined as follows:

$$\begin{array}{lll} \texttt{quast ::=} & \bot & \\ & \mid \vec{i} & \\ & \mid \textbf{if } f(\vec{p}) \geq 0 \textbf{ then } \texttt{quast} \textbf{ else } \texttt{quast} & \end{array}$$

Where $\bot$ means that no solution was found, $\vec{i}$ is a solution vector, $\vec{p}$ is a vector built from the parameters, and $f$ is an *affine* function. As a consequence, the direct dependence is not a simple operation, but a quast providing the last writings for different parameters values. The next steps aims to combine these quasts in order to obtain the quast giving the last writing of $a$.

### Combining the Direct Dependences

Once the direct dependences are computed, it remains to combine them in order to obtain the quast giving the last writing of $a$. It can be achieved by applying the following rules:

1. $\max_{\prec}(\ \bot\ ,\ Q\ ) = Q$
2. $\max_{\prec}(\ \langle S, \vec{s}\rangle\ ,\ \langle T, \vec{t}\rangle\ ) = \textbf{if } \langle S, \vec{s}\rangle \prec \langle T, \vec{t}\rangle \textbf{ then } \langle T, \vec{t}\rangle \textbf{ else } \langle S, \vec{s}\rangle$
3. $\max_{\prec}(\ \textbf{if } f(\vec{p}) \textbf{ then } Q_{\text{then}} \textbf{ else } Q_{\text{else}}\ ,\ Q\ ) =$

$$\textbf{if } f(\vec{p}) \textbf{ then } \max_{\prec}(\ Q_{\text{then}}\ ,\ Q\ ) \textbf{ else } \max_{\prec}(\ Q_{\text{else}}\ ,\ Q\ )$$

The $\max_{\prec}$ can then be used to combine the direct dependences $\{Q_1 \ldots Q_n\}$. Since the maximum operator is associative and commutative, the application order has no importance. Rules 1 and 3 can be directly applied. Since quast needs an atomic condition on parameters (no conjunctions or disjunctions), the quast obtained from rule 2 needs to be rewritten by using the rules:

- **if** $a$ and $b$ **then** $Q_1$ **else** $Q_2$ =

$$\underline{\textbf{if}}\ a\ \underline{\textbf{then}}\ (\textbf{if } b \textbf{ then } Q_1 \textbf{ else } Q_2)\ \underline{\textbf{else}}\ Q_2$$

- **if** $a$ or $b$ **then** $Q_1$ **else** $Q_2$ =

$$\begin{array}{l}\underline{\textbf{if}}\ a\ \underline{\textbf{then}}\ (\textbf{if}\ b\ \textbf{then}\ Q_1\ \textbf{else}\ Q_1)\\ \underline{\textbf{else}}\ (\textbf{if}\ b\ \textbf{then}\ Q_1\ \textbf{else}\ Q_2)\end{array}$$

Applying the algorithm to compute the reaching definition of `s` in `S2` in the above example, we obtain the following direct dependences:

$$\begin{array}{ll}S_1: & \langle S_1, \rangle\\ S_2: & \textbf{if}\ i \geq 2\ \textbf{then}\ \langle S_2, i-1\rangle\ \textbf{else}\ \bot\end{array}$$

Applying the combination rules, we obtain the following rewriting:

$$\begin{array}{ll} & \max_{\prec}\ \boxed{\textbf{if}\ i \geq 2\ \textbf{then}\ \langle S_2, i-1\rangle\ \textbf{else}\ \bot\ ,\ \langle S_1, \rangle}\\ \longrightarrow_3 & \textbf{if}\ i \geq 2\ \textbf{then}\ \max_{\prec}\boxed{\langle S_2, i-1\rangle\ ,\ \langle S_1, \rangle}\ \textbf{else}\ \max_{\prec}\boxed{\bot\ ,\ \langle S_1, \rangle}\\ \longrightarrow_2 & \textbf{if}\ i \geq 2\ \textbf{then}\ \langle S_2, i-1\rangle\ \textbf{else}\ \max_{\prec}\boxed{\bot\ ,\ \langle S_1, \rangle}\\ \longrightarrow_1 & \textbf{if}\ i \geq 2\ \textbf{then}\ \langle S_2, i-1\rangle\ \textbf{else}\ \langle S_1, \rangle\end{array}$$

which finally gives:

$$\boxed{\mathsf{RD}_{\langle S_2, i\rangle}(s) = \textbf{if}\ i \geq 2\ \textbf{then}\ \langle S_2, i-1\rangle\ \textbf{else}\ \langle S_1, \rangle}$$

Such a quast can easily be flattened in order to obtain a definition with a list of clauses of the form $\vec{i} \in D : \langle S, u(\vec{i})\rangle$.

### 6.1.2 Systems of Affine Recurrence Equations

The equivalence test described in this chapter uses an intermediate representation called System of Affine Recurrence Equations (SARE), and defined as follows.

**Definition 6.1 (SARE).** *A* System of Affine Recurrence Equations (SARE) *is a tuple* $\mathcal{S} = (\Sigma, \mathbb{A}, O, I, \mathcal{E})$*, where* $\Sigma$ *is a signature of functional symbols,* $\mathbb{A}$ *is a set of arrays of terms over* $\Sigma$*,* $O \in \mathbb{A}$ *is the* output *of* $\mathcal{S}$*,* $I \subset \mathbb{A}$ *is the set of* inputs *of* $\mathcal{S}$*, and* $\mathcal{E}$ *is a set of equations, called* clauses*, which define the value of* $O$ *from the inputs of* $I$*. Each clause of* $\mathcal{E}$ *is defined as follows:*

$$\vec{i} \in D : A[\vec{i}] = f(B_1[u_1(\vec{i})] \ldots B_n[u_n(\vec{i})])$$

*where* $D$ *is a* $\mathbb{Z}$*-polyhedron,* $A \in \mathbb{A} - I$*, each* $B_i \in \mathbb{A}$*, and the* $u_i$ *are* affine *functions. Moreover, a SARE must satisfy the* single assignment property i.e. *each array cell* $A[\vec{i}]$ *must be defined one time at most.*

```
S1  s = a(0)
    do i = 1,n
S2  | s = s + a(i)
    enddo
    return s
```

(a) Original program

$$\begin{cases} & S_1 & = & a(0) \\ i \in \{1\}: & S_2[i] & = & S_1 + a(i) \\ i \in \{2 \dots n\}: & S_2[i] & = & S_2[i-1] + a(i) \\ & O & = & S_2[n] \end{cases}$$

(b) Corresponding SARE

FIG. 6.1 – Representation of a program by a SARE

Figure 6.1 provides an example of program, and the corresponding SARE, where $\Sigma = \{+/2\}$, $\mathbb{A} = \{a, S_1, S_2, O\}$, $a$ is an input and $O$ is the output. The SAREs are a convenient intermediate representation for programs since they provide the exact data-flow information, allowing to traverse the *exact* use-def chains easily. This important property is exploited by the equivalence test described in this chapter.

The static control programs can be translated automatically in SAREs by applying the algorithm described by Feautrier in [43]. The principle of the translation is summarized thereafter.

### Translating a static control program into a SARE

Consider a static control program with statements $S_1 \dots S_n$. For sake of clarity, we assume without lost of generality that each statement uses at most one operator, but the algorithm could also work without this constraint. Given a statement $S_i$:

$$S_i : \ ... = f(v_1 \dots v_n)$$

We first compute the exact reaching definitions of the $v_j$ by using the algorithm described in Section 6.1.1. Let us write the reaching definition of the variable $v_j$ in the following manner:

$$\mathsf{RD}_{\langle S_i, \vec{i} \rangle}(v_j) = \begin{cases} \vec{i} \in D_{j,1} : & \langle S_{j,1}, \vec{i}_{j,1} \rangle \\ \dots & \\ \vec{i} \in D_{j,n_j} : & \langle S_{j,n_j}, \vec{i}_{j,n_j} \rangle \end{cases}$$

The notation $S_{j,1}$ is a simplification for $S_{u(j,1)}$, where $u : [\![1, n]\!] \times \mathbb{N}^* \to [\![1, n]\!]$. The same simplification is achieved for the iteration vectors $\vec{i}$. The corresponding SARE is then obtained by generating the following clauses:

$$\vec{i} \in D_{1,k_1} \cap \dots \cap D_{n,k_n} : \ S_i[\vec{i}] = f(S_{1,k_1}[\vec{i}_{1,k_1}] \dots S_{n,k_n}[\vec{i}_{n,k_n}])$$

For each statement $S_i$ and each $1 \leq k_j \leq n_j$, where the $\vec{i}$ of $S_i[\vec{i}]$ denotes the symbolic iteration vector of $S_i$, build from the surrounding loop counter of $S_i$.

Consider the static control program given in figure 6.1. The statement $S_1$ involves an input array which has, by definition, no reaching definition. In this case, we keep $a(0)$ and we just emit the clause:

$$S_1 = a(0)$$

Consider now the statement $S_2$. In the same manner, $a(i)$ has no reaching definitions. The reaching definition of $s$ is:

$$\mathsf{RD}_{\langle S_2,i\rangle}(s) = \begin{cases} i \in \{1\} : & \langle S_1, \rangle \\ i \in \{2 \ldots n\} : & \langle S_2, i-1 \rangle \end{cases}$$

This leads to generate the following clauses:

$$\begin{array}{lrcl} i \in \{1\} : & S_2[i] & = & S_1 + a(i) \\ i \in \{2 \ldots n\} : & S_2[i] & = & S_2[i-1] + a(i) \end{array}$$

Consider finally the statement `return s`. It could be handled as $O = s$, where $O$ denotes a special output variable. Since the reaching definition analysis provides $\mathsf{RD}_{\langle O,\rangle}(s) = \langle S_2, n\rangle$, we generate the following clause:

$$O = S_2[n]$$

Finally, we obtain the following SARE:

$$\begin{cases} & S_1 & = & a(0) \\ i \in \{1\} : & S_2[i] & = & S_1 + a(i) \\ i \in \{2 \ldots n\} : & S_2[i] & = & S_2[i-1] + a(i) \\ & O & = & S_2[n] \end{cases}$$

which corresponds to the SARE given in figure 6.1.

### 6.1.3 Presburger Relations

The algorithm described in this chapter, and the instantiation tests given in the two following chapters make an intensive usage of Presburger relations, that we present in this section.

Presburger arithmetic is the first-order theory of the natural numbers with the addition equality, and inequalities $(\mathbb{N}, +)$. It is defined by logical formulas built from $\neg$, $\vee$ and $\wedge$, equality and inequality between integer affine contraints, and quantifiers $\exists$ and $\forall$. An example of Presburger formula is:

$$\forall x \ (\exists y \ x = 2y \vee \exists y \ x = 2y + 1)$$

Testing the satisfiability of a Presburger formula is know as *integer linear programming* and is decidable, but NP-complete [77]. Rabin and Fisher [45] have shown that the worst case complexity of the satisfiability test is $2^{2^{kn}}$, where $n$ is the number of variables of the Presburger formula, and $k$ is a constant. Computing exact solutions for large integer programs is still an open problem. We can now define Presburger relations.

**Definition 6.2 (Presburger relation).** *A Presburger formula $\phi$ involving variables $i_1 \ldots i_n$ and $j_1 \ldots j_p$ can be seen as a relation called* Presburger relation *or* affine relation *from $i_1 \ldots i_n$ to $j_1 \ldots j_p$, and written:*

$$\{ \ [i_1 \ldots i_n] \rightarrow [j_1 \ldots j_p] \mid \phi \ \}$$

Presburger relations concisely summarize many informations required in program analysis. For instance, the following example captures the incrementation step of a loop counter:

$$\{\ [i] \rightarrow [i'] \mid i' = i + 1 \wedge 1 \leq i \leq n\ \}$$

In the same manner as any relation, one can define union $\cup$ and composition $\circ$ operations between two affine relations. The *transitive closure* of an affine relation $R$ is classically defined by:

$$R^* = \bigcup_{k=0}^{\infty} R^k$$

where $R^0$ denotes the identity relation, and $R^k = R \circ R^{k-1}$ for $k \geq 1$. The computation of transitive closures in *undecidable* in the general case, since they allow to simulate a multiplication, and thus to cover the well known undecidable Peano arithmetic $(\mathbb{N}, +, \times)$. However, Kelly *et al.* [63] provide an efficient heuristic to solve this problem in some specific cases. Their heuristic is implemented in the Omega library [62], that we use in our implementation.

## 6.2 Motivating Example

Consider that we have to check the equivalence between the two following programs:

```
s = a(0)
do i = 1,n
| s = s + a(i)
enddo
return s
```

```
s = a(0)
do i = 1,n-1
| s = s + a(i)
enddo
s = s + a(n)
return s
```

The two programs compute the sum of the elements of the array $a$, and the last iteration of the right program has been peeled. In order to simplify the presentation, we will assume that $n \geq 1$. Since $\mathcal{T}_{left}(a) = \sum_{i=0}^{n} a(i) = \mathcal{T}_{right}(a)$ for any input array $a$, the two programs are Herbrand-equivalent. We present thereafter the SAReQ algorithm due to Barthou *et al.* [13], to check whether two static control programs are Herbrand-equivalent.

## 6.3 Overview of the Method

Figure 6.2 sums up the main steps of the SAReQ algorithm. Given two programs to be compared, the first step is to build an automaton allowing to step them simultaneaously as soon as the operators found are the same. In a way, the automaton represents a simultaneous symbolic execution of the two programs. It remains to analyze the automaton, called *unification automaton* in order to decide whether the two programs are Herbrand-equivalent. This step will basically achieve the symbolic execution of the two programs and relies on a semi-decision procedure. These important steps are described in the following sections.

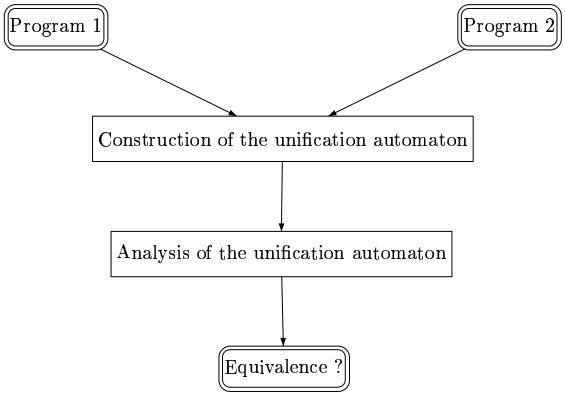



FIG. 6.2 – Main steps of the SAReQ algorithm

## 6.4 Construction of the Unification Automaton

The main idea of the algorithm is to unroll simultaneaously the two programs starting from their outputs (the `return` statement) as long as the head operators are the same. The unrolling stops whenever a constant ($1()$, $2.5()$) or an input element $a(i)$ is reached. When two inputs arrays are reached $a(i) \stackrel{?}{=} a(j)$ we have two verify that $i = j$. By analogy with the unification theory, SAReQ solves the *word problem* over the terms computed by the two algorithms: $\mathcal{T}_{left}(a) \stackrel{?}{=} \mathcal{T}_{right}(a)$.

In order to step the terms computed by the two algorithms, we need an *exact* data-flow information. SAReQ works on a dataflow representation described in Section 6.1.2 called System of Affine Recurrence Equations (SARE). Applying the translation algorithm to the running example, we obtain the two following SAREs:

$$\begin{cases} & S_1 = a(0) \\ i \in \{1\} : & S_2[i] = S_1 + a(i) \\ i \in \{2 \dots n\} : & S_2[i] = S_2[i-1] + a(i) \\ & O = S_2[n] \end{cases} \qquad \begin{cases} & S_1 = a(0) \\ i \in \{1\} : & S_2[i] = S_1 + a(i) \\ i \in \{2 \dots n-1\} : & S_2[i] = S_2[i-1] + a(i) \\ & S_3 = S_2[n-1] + a(n) \\ & O = S_3 \end{cases}$$

Following the standard rules of unification theory, the authors proposes to build an unification tree from the two programs by using the following construction rules:

**Decompose** (D )

$$\boxed{f(t_1 \dots t_n) \stackrel{?}{=} f(t'_1 \dots t'_n)} \longrightarrow \boxed{t_i \stackrel{?}{=} t'_i}$$

For each $1 \leq i \leq n$.

**Left Compute** (LC)

$$\boxed{S[\vec{i}] \stackrel{?}{=} t} \xrightarrow{\vec{i} \in D} \boxed{t_i \stackrel{?}{=} t}$$

For each clause $\vec{i} \in D : S[\vec{i}] \stackrel{?}{=} t_i$ of the left SARE.

**Right Compute** (RC)

$$\boxed{t \stackrel{?}{=} S[\vec{i}]} \xrightarrow{\vec{i} \in D} \boxed{t \stackrel{?}{=} t_i}$$

For each clause $\vec{i} \in D : S[\vec{i}] \stackrel{?}{=} t_i$ of the right SARE.

From the above SAREs, we obtain the unification tree given in the figure 6.3. For the sake of clarity, Decompose, Left Compute and Right Compute are respectively denoted by D , LC and RC. In addition, LC and RC have been applied simultaneously (LC/RC) to reduce the number of states.

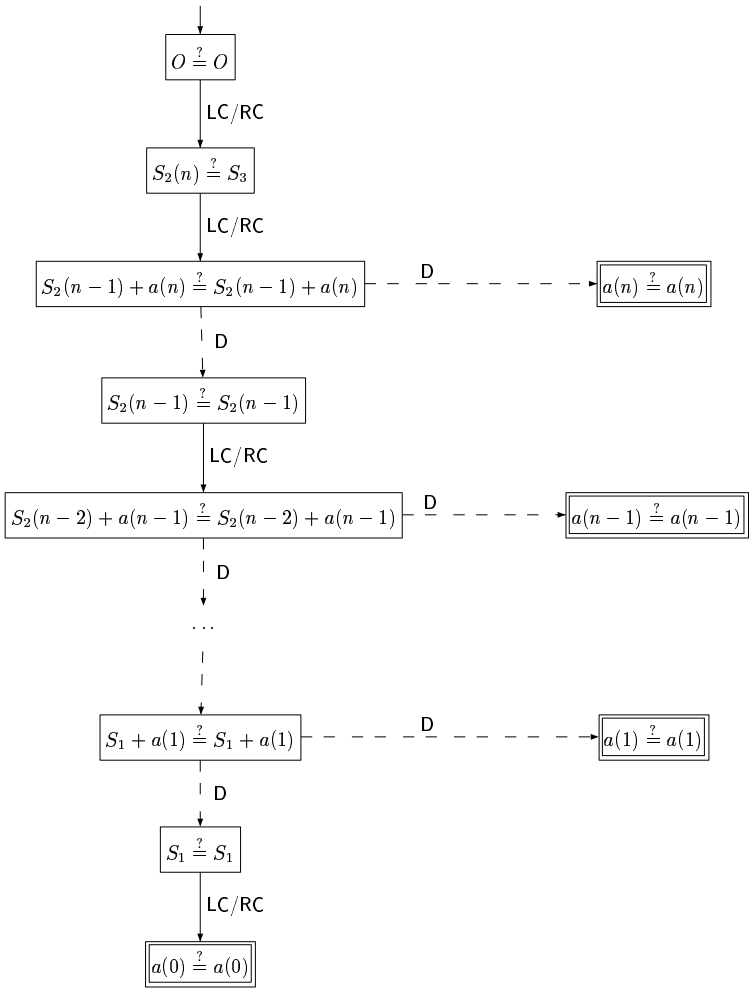



FIG. 6.3 – Unification tree of left and right SAREs

Unfortunately, the unification tree has a parametric number of states, which makes it impossible to handle directly. The authors solve this problem by introducing a *generalization rule* allowing to handle the SARE's array $s_2(i)$ in the general case, given a symbolic $i$. Figure 6.4 provides the complete system of construction rules used in the SAReQ algorithm. Left Generalize (LC) and Right Generalize (RG) rules allow to substitute the current index vector represented by $u(\vec{i})$ by the symbolic vector $\vec{i}$. The transitions are labelled by the assignement $\vec{i} \leftarrow u(\vec{i})$. In addition, Left Compute (LC) and Right Compute (RC) rules can be fired only on generalized SARE's arrays $S[\vec{i}]$, which imposes to apply a generalization rule before. Following the rules of unification theory, a failure occurs whenever the head operators of the left and right hand side are differents, which leads to add the Conflict rules.

**Decompose** (D)

$$\boxed{f(t_1 \dots t_n) \stackrel{?}{=} f(t'_1 \dots t'_n)} \longrightarrow \boxed{t_i \stackrel{?}{=} t'_i}$$

For each $1 \leq i \leq n$.

**Left Compute** (LC)

$$\boxed{S[\vec{i}] \stackrel{?}{=} t} \xrightarrow{\vec{i} \in D} \boxed{t_i \stackrel{?}{=} t}$$

For each clause $\vec{i} \in D : S[\vec{i}] \stackrel{?}{=} t_i$ of the left SARE. $\vec{i}$ *must* be a *symbolic* index vector generated by the **Left Generalize** rule.

**Right Compute** (RC)

$$\boxed{t \stackrel{?}{=} S[\vec{i}]} \xrightarrow{\vec{i} \in D} \boxed{t \stackrel{?}{=} t_i}$$

For each clause $\vec{i} \in D : S[\vec{i}] \stackrel{?}{=} t_i$ of the right SARE. $\vec{i}$ *must* be a *symbolic* index vector generated by the **Right Generalize** rule.

**Left Generalize** (LG)

$$\boxed{S[u(\vec{i})] \stackrel{?}{=} t} \xrightarrow{\vec{i} \leftarrow u(\vec{i})} \boxed{S[\vec{i}] \stackrel{?}{=} t}$$

Where $\vec{i}$ is a *symbolic* index vector of $S$, and $u \neq Id$.

**Right Generalize** (RG)

$$\boxed{t \stackrel{?}{=} S[u(\vec{i})]} \xrightarrow{\vec{i} \leftarrow u(\vec{i})} \boxed{t \stackrel{?}{=} S[\vec{i}]}$$

Where $\vec{i}$ is a *symbolic* index vector of $S$, and $u \neq Id$.

**Conflict 1**

$$\boxed{f(t_1 \dots t_n) \stackrel{?}{=} g(t'_1 \dots t'_p)} \longrightarrow \boxed{\text{FAILURE}}$$

Where $f \neq g$.

**Conflict 2**

$$\boxed{f(t_1 \dots t_n) \stackrel{?}{=} a(\vec{i})} \longrightarrow \boxed{\text{FAILURE}}$$

Where $a$ is an *input array* of the right SARE.

**Conflict 3**

$$\boxed{a(\vec{i}) \stackrel{?}{=} g(t'_1 \dots t'_p)} \longrightarrow \boxed{\text{FAILURE}}$$

Where $a$ is an *input array* of the left SARE.

FIG. 6.4 – Construction rules of the unification automaton

Applied from the initial state $O \stackrel{?}{=} O$, these constructions rules produce a finite automaton, as stated in the following proposition proved by the authors of SAReQ in [13]. Remark the *fixed*, which means that the number of states of the unification automaton does not depends on a parameter *e.g.* 10 or 15, but not $n-2$ or $2n+1$.

**Proposition 6.2 (Finitude of the unification automaton).** *Let $S_1$ and $S_2$ be two SAREs, and $\mathcal{A}_{S_1 \stackrel{?}{=} S_2}$ be there unification automaton. Then $\mathcal{A}_{S_1 \stackrel{?}{=} S_2}$ has a* fixed *number of states.*

Applying these rules on the above example, we obtain the automaton given in figure 6.5. Once again, compute rules (LC and RC) and generalization rules (LG and RG) are applied simultaneously on the left and right hand sides when it is possible, in order to reduce the number of states. Starting from the initial state $O \stackrel{?}{=} O$, the left hand side is first generalized by applying the LG rule. LG basically generates a *symbolic* index vector $i_L$, and achieve the assignation $i_L \leftarrow n$. The right hand side is then computed. Since the value of the symbol $i_L$ is unknown during the construction, the compute rules (here LC) have to explore all the possibilities. This leads to generate *unreachable* states. For presentation reasons, we have not provided the unreachable states. A consequence of the generalization rules is to produce cycles such as LC/RC $\rightarrow$ D $\rightarrow$ LG/RG allowing to capture the parametric-length branches of unification tree.

## 6.5 Analysis of the Unification Automaton

Once the unification automaton is built, it remains to analyze it in order to decide whether the two programs are Herbrand-equivalent.

Basically, the two programs are equivalent if no failure state is *reached* from the initial state. Unfortunately, the systematic application of **Left Compute** (LC) and **Right Compute** (RC) rules creates unreachable states. For instance, see figure 6.6, which provides the detail of the transitions fired from the state $s_2(i_L) \stackrel{?}{=} s_2(i_R)$. Due to transitions LG$[i_L \leftarrow n]$ and RG$[i_R \leftarrow n]$, $i_L$ and $i_R$ are initialized with the same value. Moreover they are decremented simultaneously (see transitions LG$[i_L \leftarrow i_L - 1]$ and RG$[i_R \leftarrow i_R - 1]$). Consequently, they have always the same value, and the transition detailed in figure 6.6 is never fired, making the following states *unreachable*. Therefore, a first step of the analysis is to detect the unreachable states, and to remove them.

We first introduce the definition of Memory-State Automata (MSA), allowing to express the unification automata in a formal framework. MSA were introduced by Boigelot and Wolper in [18]. The definition provided by Feautrier *et al.* in [13] is the following:

**Definition 6.3 (MSA).** *The state of an MSA has two parts: an element of a finite set and a vector of integers. The vector associated to state $p$ is denoted $v_p$ and the full state is $\langle p, v_p \rangle$. The dimension of $v_p$ is determined by $p$ and is noted $n_p$. A transition in an MSA has three elements: a start state, $p$, an arrival state $q$, and a firing relation $F_{pq}$ in $\mathbb{N}^{n_p} \times \mathbb{N}^{n_q}$. A transition from $\langle p, v_p \rangle$ to $\langle q, v_q \rangle$ can occur only if $\langle v_p, v_q \rangle \in F_{pq}$. There is an edge from $p$ to $q$ in an MSA iff $F_{pq} \neq \emptyset$.*

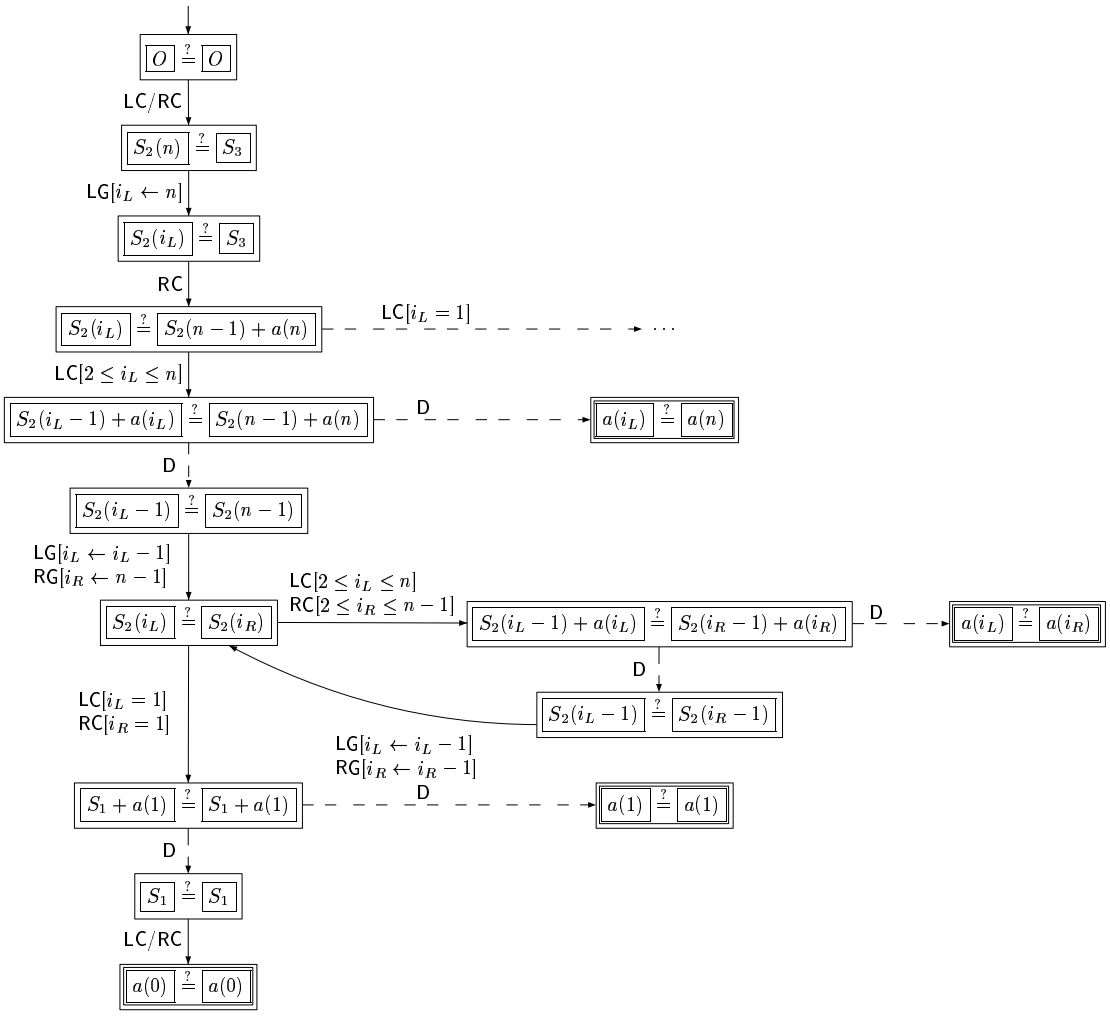



FIG. 6.5 – Unification automaton of the two programs

Let $\langle p_0, v_{p_0} \rangle$ be the initial state of the automaton. A state $\langle p, v_p \rangle$ is reachable iff there exists a finite sequence of transitions from the initial state to $\langle p, v_p \rangle$:

$$\exists p_1 \dots p_n \;\; v_{p_1} \dots v_{p_n} : (p_n = p \wedge \langle v_{p_{i-1}}, v_{p_i} \rangle \in F_{p_{i-1},p_i}).$$

The *reaching set* of $p$, noted $A_p$, is the set of vectors $v_p$ such that $\langle p, v_p \rangle$ is reachable from the initial state.

In order to compute the reaching set of $p$, a method is to compute the regular expression recognizing all paths from the initial state to $p$, by using Arden's lemma. On the motivating example, the regular expression recognized by handling $s_2(i_L) \stackrel{?}{=} s_2(i_R)$ as a final state is:

$$[i_L \leftarrow n] \cdot [2 \leq i_L \leq n] \cdot \begin{bmatrix} i_L \leftarrow i_L - 1 \\ i_R \leftarrow n - 1 \end{bmatrix} \cdot \left( \begin{bmatrix} 2 \leq i_L \leq n \\ 2 \leq i_R \leq n-1 \end{bmatrix} \cdot \varepsilon \cdot \begin{bmatrix} i_L \leftarrow i_L - 1 \\ i_R \leftarrow i_R - 1 \end{bmatrix} \right)^*$$

which leads to the reaching set $\{ (i_L, i_R) \mid i_L = i_R \text{ and } 1 \leq i_L, i_R \leq n-1 \}$. Using this method, the reaching set obtained from the states of the branch detailed in the figure 6.6 is $\emptyset$, leading to remove them.

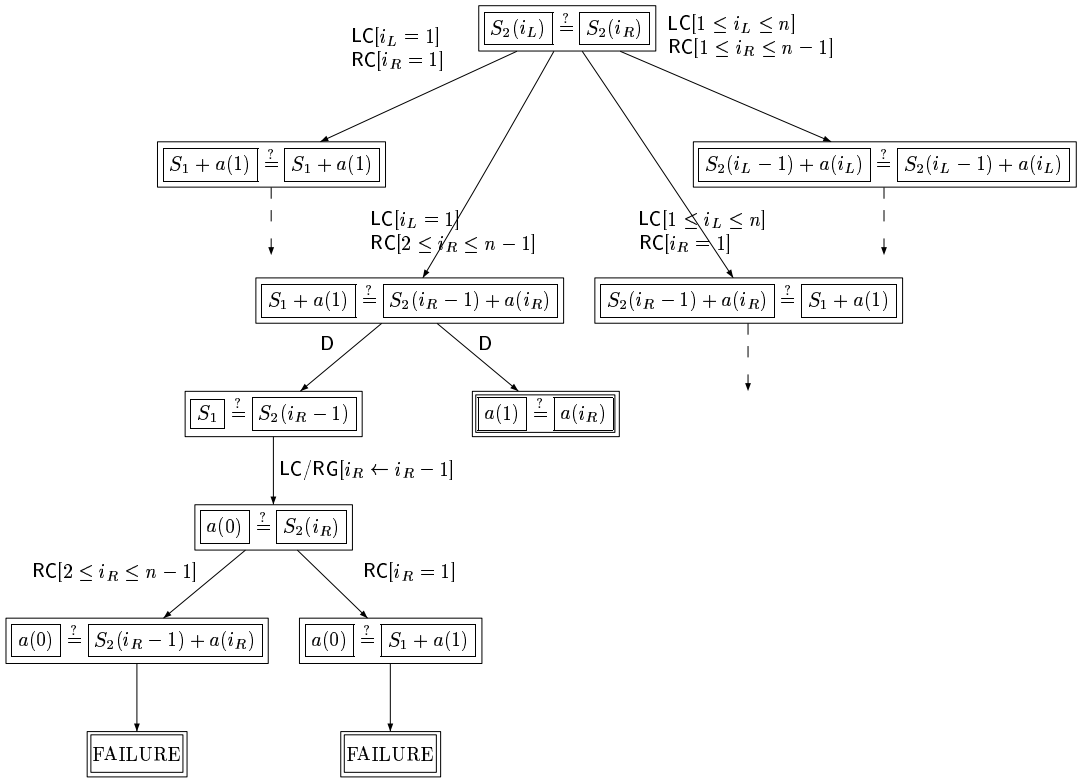



FIG. 6.6 – Detail of the transitions fired from the state $S_2(i_L) \stackrel{?}{=} S_2(i_R)$

Because of cycles, the reaching set computation is *undecidable* in general. The authors use the semi-decision procedure described in [63], and implemented in the Omega library [62]. The algorithm computes the reaching sets for each node, and remove a node when its reaching set is empty. Whenever the semi-decision procedure cannot achieve its computation, the algorithm report a failure.

Once the unreachable nodes are removed, we obtain an unification automaton which represents exactly the unification tree of the two SAREs to compare. It remains to check the nodes involving the SAREs inputs $a(i_L) \stackrel{?}{=} a(i_R)$ by verifying that $i_L = i_R$ for each possible path from the initial state $O \stackrel{?}{=} O$. It is achieved by checking whether the reaching set of $a(i_L) \stackrel{?}{=} a(i_R)$ is included in $\{ (i_L, i_R) \mid i_L = i_R \}$. If $a(i_L) \stackrel{?}{=} a(i_R)$ does not verify this condition, a transition is fired from $a(i_L) \stackrel{?}{=} a(i_R)$ to a $\boxed{\text{FAILURE}}$ state.

Following the rules used to solve the word problem in unification theory, the two SAREs are Herbrand-equivalent iff no failure state is reachable. Thus, it remains to check that the resulting MSA does not contain a $\boxed{\text{FAILURE}}$ state. The algorithm described in figure 6.7 summarizes the main steps of the analysis.

## 6.6 Related Work

We present now several approaches related to equivalence checking. The first approches are concerned with *translation validation*, and aims to certify the code produced by a

**Algorithm** *Analyze_Unification_Automaton*

**Input:** $\mathcal{A}$, *an unification-automaton obtained by applying the rules of figure 6.4*
**Output:** *Are the two input SAREs equivalent? (*$\boxed{YES}$ *or* $\boxed{?}$*)*

1. For each node $n$ of $\mathcal{A}$:
   - Compute the reaching set $r(n)$ of $n$ [semi-decision procedure]
   - If the semi-decision procedure fails, emit $\boxed{?}$
   - If $r(n) = \emptyset$, remove $n$ from $\mathcal{A}$
2. If a failure state of $\mathcal{A}$ remains reachable, emit $\boxed{?}$.
3. For each node $a(\vec{i}_L) \stackrel{?}{=} a(\vec{i}_R)$ where $a$ denotes the common input of the two SAREs:
   - If $r\left(a(\vec{i}_L) \stackrel{?}{=} a(\vec{i}_R)\right) \not\subset \{\ (\vec{i}_L, \vec{i}_R) \mid \vec{i}_L = \vec{i}_R\ \}$, emit $\boxed{?}$.
4. emit $\boxed{\text{YES}}$.

FIG. 6.7 – *Analyze_Unification_Automaton*

compiler [111, 81, 27]. Then we introduce the *model checking*, which aims to check whether a program verifies properties specified by a temporal logic formula [96, 50, 49, 37, 32]. Finally, we present two slicing-based approaches to locate equivalent slices in a program [65, 108].

*Translation validation* aims to verify that the code produce by a compiler is semantically equivalent to the source code. Such a problem is quite difficult since *source-to-source* optimizations achieved by the compiler may be aggressive and deeply change the control and data structures used in the program.

Zuck *et al.* [111] presents a theoretical framework for translation validation of optimizing compilers. They describe VOC, a tool to validate several translations performed by the SGI Pro-64 compiler. VOC works on a low-level representation near to assembly language, and rely on a theorem prover to validate proof obligations. The loop invariants are derived from inductions variables. VOC seems to be able to handle most control variations, including most loop transformations such as reversal, interchange, tiling and skewing, but does not deal with pointers, aliasing and procedure calls.

Necula [81] proposes a translation validation infrastructure, based on a simultaneous symbolic execution of the source program and its translation. The symbolic execution produces a *simulation relation* which link the variables used in both programs. The equivalence is finally decided by solving the contraints produced by simulation relation. The main drawback of this approach is its inability to cope with variations which changes deeply the control structure of the program.

Currie *et al.* [27] propose an automatic approach to compare assembly-languages routines for DSPs. It basically achieves a symbolic execution of both assembly programs. Since DSP codes contains numerous fixed-count loops, they can be unrolled completely during the symbolic execution, leading to produce the (large) terms computed by both programs. The comparison is then achieved by checking whether both terms are syntactically equals modulo associativity and commutativity of usual operators. Such an approach

is close to the equivalence test described in this chapter, without the loop constraint.

Jaramillo, Gupta and Soffa [53] propose an comparison checker to debug unsafe optimizations. Their tool, COP, executes the optimized and unoptimized programs and compare the execution traces to get the earliest point of execution when the results differ. The comparison needs a mapping between the *operations* of the original and the optimized program that must be specified by the optimizer.

*Model-checking* aims to check whether a program verifies a property specified by a formula of linear temporal logic (LTL). Model checking can be decided over *finite-state* programs *i.e.* programs whose variables range over a finite domain. The program is viewed as a finite-state machine (the *Kripke structure*), which represents the different memory states during the execution. Such a structure can be easily translated into a finite state automaton $A_P$ recognizing all possible sequence of memory states (*Büchi automaton*) [96]. A linear temporal logic formula $\phi$ aims to valid a class of memory-states sequences verifying a given property. Standart algorithms [106, 50, 49] allow to compute the Buchi automaton $A_\phi$ recognizing exactly these sequences. The model-checking problem is thus reduced to check whether all sequences accepted by the automaton $A_P$ are also accepted by the automaton $A_\phi$. This leads to check that the automaton accepting $L(A_P) \cap \overline{L(A_\phi)}$ is empty. Such a problem is decidable, and can be checked in linear time [37]. Since the Büchi automaton of the property $\phi$ has $\mathcal{O}(2^{|\phi|})$ states and the automaton representing the program have $\mathcal{O}(|P|)$ states, the size of the product automaton $A_P \times A_\phi$, which determines the overall complexity of the method is $\mathcal{O}(|P|.2^{|\phi|})$. In practice, the exponent blowup of the number of states is the main obstacle to model-checking [32].

Komondoor and Horwitz [65] propose a method to identify duplicated code segments in C programs. They rely on a slicing techique over the program dependence to find isomorphic parts that represent clones. Their method is able to detect the equivalence of non-contiguous slices, and slices involving variable renaming and statement reordering. Despite a low cost, their method can only cope with organization variations in the meaning defined in introduction.

The method proposed by Yang, Horwitz and Reps [108] also deals with a program dependence graph. They propose a sequence-congruence algorithm for detecting program components that exhibits identical execution behaviours. Their algorithm consists in two passes. The initial partition puts vertices with the same operators into the same classes. Then they apply an algorithm inspired of Alpern [8] to find the coarsest partition coherent with the initial partition.

## 6.7 Discussion

In this chapter, we have described a method due to Barthou *et al.* to check the equivalence between two programs, and we have pointed-out its ability to cope with Herbrand-equivalence. By construction, this method would be able to decide Herbrand-equivalence if the transitive closure would be decidable, which is not the case, unfortunately. One can say that Barthou *et al.* equivalence test is a semi-decision procedure for Herbrand-equivalence, whose undecidable part is the computation of transitive closures of affine relations.

Since Herbrand-equivalence is (partially) handled, the method described in this chapter is able to detect a wide range of program variations from organization to control variations. Informally, this is mostly due to the data-flow representation (SARE), which does not depends on data-structures and control structures. We provide thereafter a detailed explanation for each kind of program variation.

**Organization variations** Since the SARE are an exact data-flow representation, they allow the algorithm to describe exactly the statements involved in the computed term without reaching the other ones. This leads to handle organization variations.

**Data structure variations** SAReQ performs a symbolic execution of both programs, and check whether the operators used are the same without taking data-structures into account. Moreover, the SARE representation normalizes data-structure with multidimensional arrays. As a consequence, data-structures variations are handled.

**Control variations** Most of control variations do not affect the dependences, and consequently the term computed by the program. These variations include particularly most of loop transformations (peeling, skewing, tiling, etc), and also variations with conditionals such as if-conversion.

**Semantics variations** Since SAReQ has been designed to check a *syntactic* equivalence between the terms computed by both programs, it does not take account of operators properties such as associativity or commutativity. This unfortunately leads to discard semantics variations.

As mentionned above, the critical operation of the equivalence test is the computation of transitive closures for affine relations. In our implementation, this is achieved by using the Omega library [62]. This operation is widely used in the algorithm which makes it expensive in time.

In this thesis, we propose to detect *generic* libraries, involving templates. Generic libraries is a new emerging library paradigm, which provides algorithms expressed in a data-structure neutral fashion. This leads to reduce drastically the number of library operations. Typically, for a library with $D$ data-structures sharing $O$ operations, a standard library will provides $D \times O$ operations, whereas a generic library will only provides $O$ generic operations, which can be instantiated on $D$ data-structures. Such a feature reduces considerably the number of library functions to learn, and thus the readability of the code using the library. Another important fact is the increasing amount of generic libraries which have emerged in the last years [95, 93, 9, 97, 61, 28].

In order to address this problem, the two next chapters presents two different procedures able to (semi-)decide whether a program slice is an *instance* of a template. The first procedure is an extension of SAReQ , while the other one follows the principle of the slicing method with an exact data-flow information.

# Chapter 7

# Template Matching with Semi-Unification

The slicing method given in Chapter 5 provides a set of program slices that *may* match a given template. The next step is to keep automatically the slices that *must* matches with the template, and to provide the solutions in case of success.

The template matching problem is connected with the semi-unification problem, which consists basically in resolving equations in term structures, where one term is bound. This problem is known to be decidable when the type of free variables is limited to the fourth-order. Huet and Lang [56] propose a second-order matching algorithm motivated by an application to program schema recognition on functional programs. De Moor and Sittampalam [31] propose an extension of Huet and Lang's algorithm to third-order matching. Their method aims to apply optimizing transformations on functional program, such as list promotion. Yokoyama et al. [109] restrict Huet and Lang's algorithm to the class of deterministic second-order patterns, and provide an algorithm in $\mathcal{O}(T^2 \times P)$ where $T$ and $P$ denotes the size of the template and the program. Unfortunately, none of these methods are adaptable to imperative programs.

In the previous chapter, we have described an algorithm due to Barthou *et al.* to semi-decide the *equivalence problem* between two programs, that we propose to extend in this chapter to the *matching problem* between a template and an imperative static control program (slice). As the matching procedures described in unification theory, our algorithm is able to provide the *unifiers*, that is the values of the template variables that ensure the equivalence with the program slice.

This chapter is structured as follows: Section 7.1 presents the simply typed $\lambda$-calculus and its high-order matching problem. Section 7.2 presents the motivating example, which will be used to describe our algorithm. Section 7.3.1 shows the connection between the template matching problem and the second-order matching problem, and points out some limitations of the existing matching procedures. Section 7.3.2 sums the main steps of the algorithm, described in Sections 7.4 and 7.5. Section 7.6 provides a complexity study of the method, and reveals a time complexity in the same order than for the equivalence test for a small number of template variables. Finally, Section 7.7 analyzes the capabilities and the limitations of our method.

## 7.1 Background

We first introduce some elements of simply typed $\lambda$-calculus, then we present high-order matching problem, and particularly second-order matching problem that we propose to extend in this chapter in order to (semi)-decide the matching problem between a template and a program.

### 7.1.1 Simply Typed $\lambda$-calculus

The $\lambda$-calculus is a theoretical functional programming language introduced by Alonzo Church in 1936 in order to study the computable functions. It is surprisingly Turing-complete, despite a minimal syntax ; and provides a convenient theoretical framework for the high-order unification, and particularly the second-order matching problem addressed in this chapter. This section presents briefly the basic notions needed to introduce the matching problem.

#### Types

As most functional languages, simply typed $\lambda$-calculus works with typed functions. In a general manner, one could design a functional language working without types, applying a simple rewriting until normalization. Types can be seen as a facility increasing readability of a program, and allowing to track bugs. They are introduced here to have the notion of *order*.

**Definition 7.1 (Type).** *Given a countable set $\mathcal{B}$ whose elements are called base types, we define the set $\tau(\mathcal{B})$ of types by:*

- $\mathcal{B} \subset \tau(\mathcal{B})$
- $T$ *and* $U \in \tau(\mathcal{B}) \Longrightarrow T \to U \in \tau(\mathcal{B})$

For example, $\mathsf{char} \to \mathsf{int}$ and $\mathsf{int} \to \mathsf{int} \to \mathsf{char}$ are types of $\tau(\{\mathsf{char}, \mathsf{int}\})$. It may seem at first glance that $\tau(\mathcal{B})$ does not types functions with several parameters. This problem is solved by typing an $n$-ary function $f : T_1 \times \ldots \times T_n \to T$ as:

$$T_1 \to (T_2 \to \ldots \to (T_n \to T))$$

Such a type transformation is called *Currying*. In order to handle naturally *Currying*, we will simply write $T_1 \to \ldots \to T_n \to T$, assuming the right associativity of $\to$.

**Definition 7.2 (Order).** *The* order *of a type $T \in \tau(\mathcal{B})$ is defined as follows:*

- *if $T$ is a base type of $\mathcal{B}$, then $o(T) = 1$*
- *if $T = U \to V$, then $o(T) = \max\{1 + o(U), o(V)\}$*

Basically, the order represent the functional level of a type. first-order types are constants, second-order types are functions of base types, third-order types are functions taking second-order functions in parameter, etc.

### $\lambda$-terms

The following definition specifies the syntax of the $\lambda$-calculus. It basically rely on two simple operations, which are function definition (abstraction) and function evaluation (application).

**Definition 7.3 ($\lambda$-term).** *Given a countable set $\Sigma$ whose elements are called* functional symbols*, and a countable set $V$ whose elements are called variables we define inductively the set $\Lambda(\Sigma, V)$ of* $\lambda$-terms *over $\Sigma$ and $V$ by:*

- $\Sigma \cup V \subset \Lambda(\Sigma, V)$
- $x \in V$ *and* $t \in \Lambda(\Sigma, V) \Longrightarrow \lambda x.t \in \Lambda(\Sigma, V)$ (Abstraction)
- $t_1$ *and* $t_2 \in \Lambda(\Sigma, V) \Longrightarrow (t_1 t_2) \in \Lambda(\Sigma, V)$ (Application)

For instance, $\lambda x.\lambda y.y$ and $\lambda x.(fx)$ are $\lambda$-terms with functional symbols in $\{f\}$ and variables in $\{x, y\}$. For the sake of clarity, we will write $\lambda x_1 \dots x_n.t$ for $\lambda x_1.\lambda x_2 \dots \lambda x_n.t$. Additionaly, we will write $t_1 \dots t_n$ for $((t_1 t_2) \dots t_n)$.

Within simply-typed $\lambda$-calculus, a type of $\tau(B)$ is assigned to each fonctional symbol of $\Sigma$ and each variable of $V$. The type of a $\lambda$-term is thus defined in the following manner:

**Definition 7.4 (Type of a $\lambda$-term).** *The* type $Type(t)$ *of a $\lambda$-term $t \in \Lambda(\Sigma, V)$ is defined by:*

- $Type(f)$ *is given for each $f \in \Sigma$*
- $Type(v)$ *is given for each $v \in V$*
- $Type(\lambda x.t) = Type(x) \to Type(t)$
- $Type((t_1 t_2)) = V$ *if* $Type(t_1) = U \to V$ *and* $Type(t_2) = U$.

*If the last condition is always verified, we note $t : Type(t)$; and we say that $t$ is* well typed.

For example, assuming $x : \mathsf{int}$ and $y : \mathsf{int} \to \mathsf{char}$, and $f : \mathsf{char} \to \mathsf{char}$ we should have $\lambda x.f(yx) : \mathsf{int} \to \mathsf{char}$. In this chapter, we will consider only well-typed $\lambda$-terms. Additionally, we will talk about the *order* of a $\lambda$-term $t$ for the order of its type $o(Type(t))$.

The free variables of a $\lambda$-term $t$ are the variables of $t$ which are not in the scope of a $\lambda$. In a formal manner:

**Definition 7.5 (Free variables).** *Given a $\lambda$-term $t$, the set $\mathcal{FV}(t)$ of* free variables *occurring in $t$ is defined by:*

- $\mathcal{FV}(f) = \emptyset$ *for any functional symbol $f$.*
- $\mathcal{FV}(x) = \{x\}$ *for any variable $x$.*
- $\mathcal{FV}(\lambda x.t) = \mathcal{FV}(t) - \{x\}$
- $\mathcal{FV}((uv)) = \mathcal{FV}(u) \cup \mathcal{FV}(v)$

The variables of $t$ which are not free are said to be *bound*. If all variables of $t$ are bound, $t$ is said to be *ground*, or *closed*. For example, $\lambda x.f(yx)$ have one free-variable $y$, its bound variable is $x$.

A substitution is an application which replace all the occurences of variables by given $\lambda$-terms. In a more formal manner:

**Definition 7.6 (Substitution).** *Given a $\lambda$-term $t$ and a variable $x$. We define the* substitution *as an application $\sigma = [x_1/t_1 \ \ldots \ x_n/t_n] : \Lambda(\Sigma, V) \longrightarrow \Lambda(\Sigma, V)$ verifying:*

- $\sigma(f) = f$ *for any functional symbol* $f$
- $\sigma(x_i) = t_i$
- $\sigma(y) = y$ *if* $y \in V - \{x_1 \ldots x_n\}$
- $\sigma(uv) = (\sigma(u)\sigma(v))$
- $\sigma(\lambda y.t) = \lambda y.\sigma(t)$ *if* $y \in V - \{x_1 \ldots x_n\}$
- $\sigma(\lambda x_i.t) = \lambda y.\sigma(t_y)$, *where $y$ is a fresh variable, and $t_y$ is the term obtained by substituting all* free *occurrences of $x_i$ in $t$ by $y$*

An example of substitution and its application is $[x/y](x(\lambda x.x)) = y(\lambda x.x)$.

**Execution of a $\lambda$-term**

Once the syntax of the $\lambda$-calculus is specified, it remains to provide an execution paradigm. The execution of a $\lambda$-term mainly rely on a syntactic evaluation the functions defined by the abstraction rule. The following definition provides the execution rules of $\lambda$-terms.

**Definition 7.7 ($\beta$-reduction).** *We define the $\beta$-reduction by:*

$$(\lambda x.t)t' \longrightarrow_\beta [x/t'](t)$$

*The $\beta$-reduction is moreover compatible with the application and the abstraction:*

$$t \longrightarrow_\beta t' \Longrightarrow \begin{cases} (tu) \longrightarrow_\beta (t'u) \\ (ut) \longrightarrow_\beta (ut') \quad \textit{where } u \in \Lambda(\Sigma, V) \\ \lambda x.t \longrightarrow \lambda x.t' \end{cases}$$

The reflexive and transitive closure of $\longrightarrow_\beta$ is denoted by $\longrightarrow_\beta^*$. The following example provides the execution of a simple projection:

$$\begin{array}{lll} ((\lambda x.\lambda y.y)t_1)t_2 & \longrightarrow_\beta & [x/t_1](\lambda y.y)t_2 \\ & = & (\lambda y.y)t_2 \\ & \longrightarrow_\beta & [y/t_2]y \\ & = & t_2 \end{array}$$

Since $\lambda$-calculus is Turing-complete, it is not limited to simple functions and can express any computable function.

A $\lambda$-term is said to be in *normal form* if it can no more be rewritten by using $\beta$-reduction. In the above example, $t_2$ is in normal form. In addition, a $\lambda$-term $t$ is said to be *normalizable* if there exists a $\lambda$-term $u$ in normal form such that:

$$t \longrightarrow_\beta^* u$$

In this case, $u$ can be see as the result of the computation described by $t$.

Depending on rewriting strategy, there may exists different paths to $\beta$-reduce $t$ to a normal form. For example, figure 7.1 provides all possible derivations of $((\lambda x.x)(\lambda x.x))((\lambda y.y)t)$ to the normal form $t$.

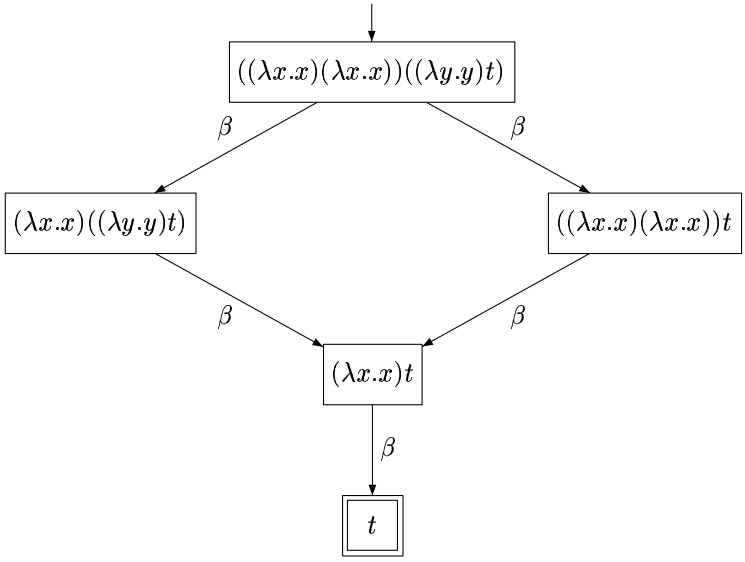



FIG. 7.1 – All possible sequences of $\beta$-reductions on a $\lambda$-term

The following theorem demonstrates the unicity of the normal form by proving the *confluence* of $\beta$-reduction.

**Theorem 7.1 (Church-Rosser).** *Let $t$, $u$ and $v \in \Lambda(\Sigma, V)$ such that $t \longrightarrow_\beta^* u$ and $t \longrightarrow_\beta^* v$. Then there exists a $\lambda$-term $w \in \Lambda(\Sigma, V)$ with:*

$$u \longrightarrow_\beta^* w \text{ and } v \longrightarrow_\beta^* w$$

As a consequence, a normalizable $\lambda$-term $t$ has a unique normal form, that we will write $t \downarrow_\beta$.

**Proposition 7.1 (Head normal form).** *Let $t$ be a normalized term well-typed term with type $T_1 \to \ldots T_n \to U$, where $U \in \mathcal{B}$. Then the term $t$ has the form:*

$$t = \lambda x_1 \ldots \lambda x_p (y u_1 \ldots u_q)$$

*where $y \in \Sigma \cup V$.*

When $y \in \Sigma$, $t$ is said to be a *rigid* $\lambda$-term, and when $y \in V$, $t$ is said to be a *flexible* $\lambda$-term.

### 7.1.2 High-Order Matching

High-order matching is a particular case of high-order unification, which consists in solving equations on term structures, where one term is closed. These two important problems are briefly presented in this section.

**Definition 7.8 (Unification problem, Unifier).** *A* unification problem *is a system of equations:*

$$\begin{cases} u_1 & \stackrel{?}{=} & v_1 \\ \dots & & \\ u_n & \stackrel{?}{=} & v_n \end{cases}$$

*where the $u_i$ and $v_i$ are $\lambda$-terms of $\Lambda(\Sigma, V)$.*
*A* unifier *is a substitution $\sigma$ such that:*

$$\sigma(u_i) \downarrow_\beta = \sigma(v_i) \downarrow_\beta$$

*for each $1 \leq i \leq n$, where $=$ denotes the* syntactic *equivalence between $\lambda$-terms.*

We consider here unification problems with one equation ($n = 1$). For the sake of clarity, the free variables of $u_i$ and $v_i$ will be denoted by using capital letters. Bound variables and fonctionals symbols will be denoted by small letters. An example of unification problem is:

$$\lambda x.X(fx) \stackrel{?}{=} \lambda xy.f(Xx)$$

which has an infinite set of unifiers, which can be written $\{[X/\lambda x.f^n x],\ n \in \mathbb{N}\}$.

The unification problem is unfortunately undecidable. A proof using a reduction of the tenth Hilbert's problem can be found in [35], page 1025. In order to find interesting decidable sub-cases, the idea commonly investigated in the litterature is to restrict the order of the free-variables. For a given order $n$, the corresponding unification problem is called $n$-th unification problem. Without such restrictions, we will talk about high-order unification problem, or simply unification problem. Unfortunately, the unification problem remains undecidable beyond the second-order [52].

We will now introduce an extension of the unification problem allowing to take into account semantic properties of functional symbols.

**Definition 7.9 (Equational theory).** *An* equational theory *is a set $E$ of equalities between $\lambda$-terms of $\Lambda(\Sigma, V)$. We built from $E$ the least equivalence relation $=_E$ which contains $E$.*

Taking $\Sigma = \{+\}$, an example of equational theory should be:

$$(+xy) \rightarrow_E (+yx)$$
$$(+(+xy)z) \rightarrow_E (+x(+yz))$$

For any $\lambda$-terms $x$, $y$ and $z \in \Lambda(\Sigma, V)$ . This important equational theory specify the commutativity and the associativity of $+$. The equational unification, or $E$-unification, consists in solving an unification problem modulo the equivalence relation induced by a given equational theory, as stated in the following definition.

**Definition 7.10 (Equational unification problem).** *An* equational unification problem w.r.t. an equational theory $E$, *or* $E$-unification problem *is a system of equations:*

$$\begin{cases} u_1 & \stackrel{?}{=} & v_1 \\ \cdots & & \\ u_n & \stackrel{?}{=} & v_n \end{cases}$$

*where the* $u_i$ *and* $v_i$ *are* $\lambda$*-terms of* $\Lambda(\Sigma, V)$.
*A* unifier *is a substitution* $\sigma$ *such that:*

$$\sigma(u_i) \downarrow_\beta =_E \sigma(v_i) \downarrow_\beta$$

*for each* $1 \leq i \leq n$.

Since equational unification is a generalization of unification, it is also undecidable. We will now address an important restriction of the unification problems where a term is bound in each equation. Such a unification problem is called (high-order)*matching problem*, or *semi-unification* problem. Its unifiers are called *semi-unifiers*.

**Definition 7.11 (Matching problem).** *A* semi-unification problem *or* matching problem *is a unification problem:*

$$\begin{cases} u_1 & \stackrel{?}{=} & v_1 \\ \cdots & & \\ u_n & \stackrel{?}{=} & v_n \end{cases}$$

*where the* $v_i$ *are* bound $\lambda$*-terms of* $\Lambda(\Sigma, V)$. *A* unifier *of matching problem, or* semi-unifier *is a substitution* $\sigma$ *over* $\Lambda(\Sigma, V)$ *such that:*

$$\sigma(u_i) \downarrow_\beta = v_i$$

*for each relevant* $i$.

Hopefully, the matching problem has been proven decidable for the orders two [56], three [34] and four [82]. The decidability of the high-order matching problem is still open. This chapter aims to extends the second-order matching procedure due to Huet and Lang [56] in order to (semi-)decide the matching problem between a template and a program.

## 7.2 Motivating Example

We present now the running example that we will handle along this chapter. Consider the following matching problem between a template (left) and a candidate program slice (right):

```
s = a_T(0)                    s = a(0)
do i = 1,n                    do i = 1,n
| s = X(s,a_T(i))             | s = s + a(i)
enddo                         enddo
return s                      return s
```

The program computes the sum of the array $a$, while the template represents the reductions over the array $a_T$ with respect to the operator $X$. The program obviously matches the template with the unifier:

$$\begin{array}{ll} a_T(i) & = a(i) \;\; 1 \leq i \leq n \\ X(x,y) & = +(x,y) \end{array}$$

In this chapter, we present an extension of the equivalence test described in Chapter 6, able to semi-decide the matching problem between a template and a program.

## 7.3 Principle of the Algorithm

The equivalence algorithm described in Chapter 6 basically solves the *word problem* over the terms computed by the two programs to be compared. In the same manner, the matching problem between a template $T$ and a program $P$ can be written as a *semi-unification problem* between the terms computed by $T$ and $P$:

$$\lambda a_T(0) \ldots a_T(n).\ \mathcal{T}_T(a_T) \stackrel{?}{=} \lambda a(0) \ldots a(n).\ \mathcal{T}_P(a)$$

where the input array cells are handled as bound variables, and template variables as free variables. Taking $n = 2$ in the motivating example, the matching problem between the template and the program could be written as follows:

$$\lambda a_T(0)a_T(1)a_T(2).\ X(X(a_T(0), a_T(1)), a_T(2)) \stackrel{?}{=} \lambda a(0)a(1)a(2).\ +(+(a(0), a(1)), a(2))$$

A simple solution would be to apply one of the unification procedures provided in the litterature [56, 35]. Unfortunately they are not able to handle the case where the terms to unify have a parametric size, which is often the case while matching a program to a template. This chapter presents a matching algorithm obtained by combining the equivalence test presented in chapter 6 with the Huet and Lang's unification procedure [56], that we describe thereafter.

### 7.3.1 Huet and Lang's Procedure

Figure 7.2 provides Huet and Lang's procedure [56], to solve the semi-unification problem. Huet and Lang's procedure basically inspects the left and right terms as long as the head operators are the same (**Decompose**). If the head operators are different (**Conflict**), the procedure stops and yields a failure $\perp$. The inspection stops whenever the left hand side starts with a free variable $X$. Huet and Lang's procedure will then try to define $X$ by all possible sub-terms starting from the top of the right hand side *e.g.* for the rhs $\lambda xyz.+(*(x,y),z)$, the procedure would try the definitions $\Box$, $+(\Box,\Box)$, $+(*(\Box,\Box),\Box)$, where $\Box$ denotes an argument of $X$. These definitions are built by choosing in a non-deterministic manner to absorb the head operator of the rhs (**Imitate**), or to stop the unrolling of the rhs (**Project**). The **Decompose** rule is then be applied to find further free-variables to compute. When all free-variables are defined, the procedure stops by firing the **Delete** rule, and provides the definitions in $\sigma$.

**Delete**

$$\frac{\left\{\lambda\vec{x}.t \stackrel{?}{=} \lambda\vec{x}.t\right\} \cup \mathcal{E}, \sigma}{\mathcal{E}, \sigma}$$

Where $t$ is a term without free variables, $t \in \mathcal{T}(\Sigma)$.

**Decompose**

$$\frac{\left\{\lambda\vec{x}.f(t_1 \dots t_n) \stackrel{?}{=} \lambda\vec{x}.f(t'_1 \dots t'_n)\right\} \cup \mathcal{E}, \sigma}{\left\{\lambda\vec{x}.t_1 \stackrel{?}{=} \lambda\vec{x}.t'_1 \ , \ \dots \ , \ \lambda\vec{x}.t_n \stackrel{?}{=} t'_n\right\} \cup \mathcal{E}, \sigma}$$

Where $f$ is a functional symbol of $\Sigma$.

**Conflict 1**

$$\frac{\left\{\lambda\vec{x}.f(t_1 \dots t_n) \stackrel{?}{=} \lambda\vec{x}.g(t'_1 \dots t_p)\right\} \cup \mathcal{E}, \sigma}{\emptyset, \bot}$$

If $f$ and $g \in \Sigma$, and $f \neq g$.

**Conflict 2**

$$\frac{\left\{\lambda\vec{x}.f(t_1 \dots t_n) \stackrel{?}{=} \lambda\vec{x}.x_i)\right\} \cup \mathcal{E}, \sigma}{\emptyset, \bot}$$

If $f$ is functional symbol of $\Sigma$.

**Conflict 3**

$$\frac{\left\{\lambda\vec{x}.x_i \stackrel{?}{=} \lambda\vec{x}.g(t'_1 \dots t_p)\right\} \cup \mathcal{E}, \sigma}{\emptyset, \bot}$$

If $g$ is a functional symbol of $\Sigma$.

**Project**

$$\frac{\left\{\lambda\vec{x}.X(t_1 \dots t_n) \stackrel{?}{=} \lambda\vec{x}.t\right\} \cup \mathcal{E}, \sigma}{\left\{\lambda\vec{x}.t_i \stackrel{?}{=} \lambda\vec{x}.t\right\} \cup \theta\mathcal{E}, \theta}$$

Where $X$ is a free variable of $\mathcal{X}$, and $\sigma$ is expanded in $\theta$ by defining $X$ as a projection on its $i^{th}$ parameter:

$$\theta = \sigma \circ [X(x_1 \dots x_n) = x_i]$$

For a given $1 \leq i \leq n$.

**Imitate**

$$\frac{\left\{\lambda\vec{x}.X(t_1 \dots t_n) \stackrel{?}{=} \lambda\vec{x}.f(t'_1 \dots t'_p)\right\} \cup \mathcal{E}, \sigma}{\left\{\lambda\vec{x}.X_1(\theta t_1 \dots \theta t_n) \stackrel{?}{=} t'_1 \ , \ \dots \ , \ \lambda\vec{x}.X_p(\theta t_1 \dots \theta t_n) \stackrel{?}{=} t'_p\right\} \cup \theta\mathcal{E}, \theta}$$

Where $X$ is a free variable of $\mathcal{X}$, $f$ is a functional symbol of $\Sigma$ and $\sigma$ is expanded in $\theta$ by defining $X$ in the following manner:

$$\theta = \sigma \circ [X(\vec{x}) = f(X_1(\vec{x}) \dots X_p(\vec{x}))]$$

where the $X_i$ are fresh free-variables.

FIG. 7.2 – Huet and Lang's semi-unification procedure

Applying Huet and Lang's procedure on the above example, we would obtain the *unification tree* provided in figure 7.3. Following the rules described in figure 7.2, we first try to compute the value of $X$ by testing all possible projections (Project 1 and Project2), and the imitation of the rhs. The projections obviously lead to a conflict, since they lead to compare a bound variable ($a(0)$ or $a(2)$) to a term starting by a functional symbol of $\Sigma$. The imitation assumes $X(x,y) = +(X_1xy, X_2xy)$, and leads to compute the values of $X_1$ and $X_2$. In the same manner, the projections and the imitation will be tested. For presentation reasons, we just provide the choices leading to a solution, which are $X_1(x,y) = x$ and $X_2(x,y) = y$. This will lead to emit $X(x,y) = +(x,y)$ as a solution of the matching problem between the template and the program. The correspondance between $a_T$ and $a$ can moreover be found on the leaf states ; this important fact will be exploited later to recover the mapping between template and program inputs.

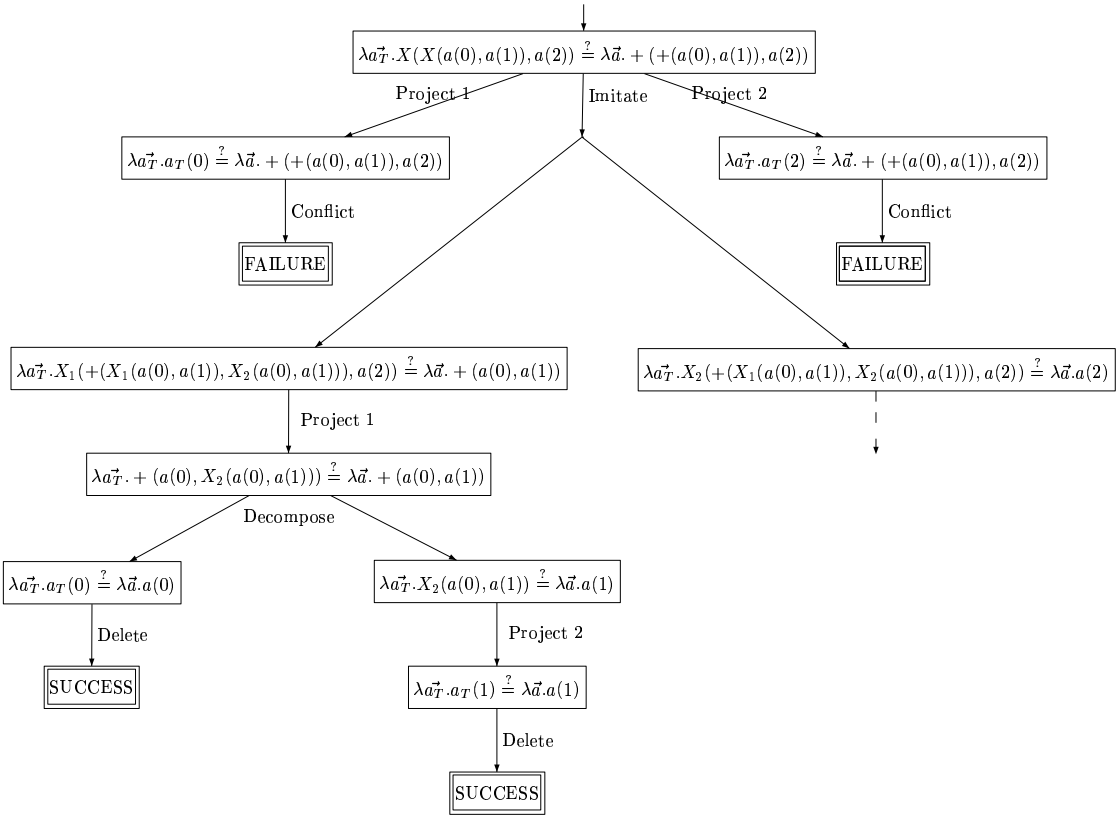



FIG. 7.3 – Unification tree of the template and the program for $n = 2$

Unfortunately, the terms computed by the template and the program have most often a parametric size, which makes Huet and Lang's procedure impossible to apply directly. In the above example, we have for example $\mathcal{T}_T(a_T) = X(\ldots X(a(0),a(1))\ldots a(n))$. The idea investigated in this chapter is to extend the equivalence test described in Chapter 6 with the rules **Project** and **Imitate** to construct the value of the template variables.

### 7.3.2 Overview of the Method

Figure 7.4 sums up the main steps of our instantiation test. Following the equivalence test, we propose to capture the unification tree of the template and the program into a fixed size automaton, that we will analyze to check whether the program matches the template and to provide the unifiers in case of success. The following sections describe each of these important steps.

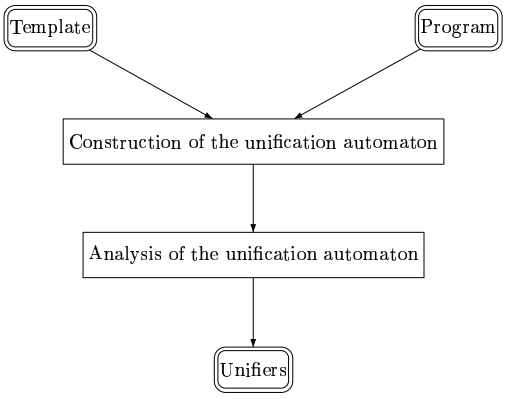



FIG. 7.4 – Overview of the instantiation test

## 7.4 Construction of the Unification Automaton

As stated in the previous section, the matching problem between a template $T$ and a program $P$ can be written as a semi-unification problem between their computed terms:

$$\lambda\vec{a_T}.\mathcal{T}_T(a_T) \stackrel{?}{=} \lambda\vec{a}.\mathcal{T}_P(a)$$

Since $\mathcal{T}_T(a_T)$ and $\mathcal{T}_P(a)$ can have a parametric size, a direct application of Huet and Lang's procedure would lead to a unification tree with a parametric number of nodes. Following the equivalence test described in Chapter 6, this section proposes to capture the unification tree into an unification automaton with a *non-parametric and finite* number of states.

In the same manner as the equivalence test, the template and the program are represented by a system of affine recurrence equations (SARE), a dataflow representation allowing to step easily their computed terms. The definition of a SARE, and a translation algorithm from a static control program can be found in Chapter 6. The SAREs obtained from the template (left) and the program (right) are:

$$\begin{cases} i \in \{0\} : & S_2[i] = a_T(0) \\ i \in \{1 \dots n\} : & S_2[i] = X(S_2[i-1], a_T(i)) \\ & O \quad = S_2[n] \end{cases} \qquad \begin{cases} i \in \{0\} : & S_2[i] = a(0) \\ i \in \{1 \dots n\} : & S_2[i] = S_2[i-1] + a(i) \\ & O \quad = S_2[n] \end{cases}$$

Figure 7.5 provides the construction rules of the unification automaton. Following Huet and Lang's procedure, the states are extended with the current unifier $\sigma$. The two SAREs are unfolded starting from the state $O \stackrel{?}{=} O$ with the empty unifier (identity mapping). Decompose, Left Compute, Right Compute, Left Generalize and Right Generalize are the same rules as in the equivalence test. The **Decompose** rule allows to inspect the *rigid-rigid* equations until a *flexible-rigid* equation is found, while the compute rules LC and RC allow to unfold on-demand the recurrence of both SAREs. In the same manner as for the equivalence test, the generalization rules LG and RG ensure the finitude of the automaton by folding the parametric length branches of the unification tree into cycles. When a *flexible-rigid* equation $X(\ldots) \stackrel{?}{=} t, \sigma$ is reached, $X$ is replaced by its value if it is already defined in the current unifier $\sigma$ (**Substitute**). Otherwise, we try several definitions for $X$ including the projections on its differents arguments (**Project**) and the imitation of the rigid term (**Imitate**). Each of these assumptions which will be checked during the analysis of the automaton in order to compute the unifiers of both template and program. Remark that no transitions can be fired from the states following the application of Project or Imitate to a previous state, since their current unifier are not the same. Consequently, we can consider that Project and Imitate construct a *search tree* whose leaves are assumptions on the template variables. For each leaf with a candidate unifier $\sigma$, an equivalence automaton is build by using SAReQ rules (decompose, compute and generalize) to check the equivalence between $\sigma(T)$ and $P$.

For presentation reasons, we have not provided the rules leading to a failure in the figure 7.5. They are the same than for the equivalence test, and basically occurs whenever the two head operators of a *rigid-rigid* equation are not the same. We briefly recall them thereafter:

**Conflict 1**

$$\frac{f(t_1 \ldots t_n) \stackrel{?}{=} g(t'_1 \ldots t'_p)}{\sigma} \longrightarrow \boxed{\text{FAILURE}}$$

Where $f \neq g$.

**Conflict 2**

$$\frac{f(t_1 \ldots t_n) \stackrel{?}{=} a(\vec{i})}{\sigma} \longrightarrow \boxed{\text{FAILURE}}$$

Where $a$ is an input array of the right SARE.

Applying the construction rules to the motivating example, we obtain the automaton given in figure 7.6. The value of the current unifier $\sigma$ is provided on the left side of important states, after a projection or an imitation. Following the construction rules, the two SAREs are unrolled until a *flexible-rigid* equation is obtained (first colored state).

Consider the **first colored state**. Since $X$ is not defined by the current unifier, Project and Imitate rules are applied. For presentation reasons, the two projections are not detailed, but would lead to a failure. The imitation assumes that $X(x, y) = X_1(x, y) + X_2(x, y)$, and leads to compute the values of $X_1$ and $X_2$. The right branch of the following Decompose is not detailed for sake of clarity. Since Project and Imitate require the heads symbol of the rhs to be a functional symbol of $\Sigma$ or an input, the Right Compute rule is applied, leading to the next colored state.

From the **next colored state**, Huet and Lang rules are applied to compute $X_1$. One

**Decompose (D)**

$$\underset{\sigma}{\boxed{f(t_1 \dots t_n) \stackrel{?}{=} f(t'_1 \dots t'_n)}} \longrightarrow \underset{\sigma}{\boxed{t_i \stackrel{?}{=} t'_i}}$$

For each $1 \leq i \leq n$.

**Left Compute (LC)**

$$\underset{\sigma}{\boxed{S[\vec{i}] \stackrel{?}{=} t}} \xrightarrow{\vec{i} \in D} \underset{\sigma}{\boxed{t_i \stackrel{?}{=} t}}$$

For each clause $\vec{i} \in D : S[\vec{i}] \stackrel{?}{=} t_i$ of the left SARE. $\vec{i}$ *must* be a *symbolic* index vector generated by the **Left Generalize** rule.

**Right Compute (RC)**

$$\underset{\sigma}{\boxed{t \stackrel{?}{=} S[\vec{i}]}} \xrightarrow{\vec{i} \in D} \underset{\sigma}{\boxed{t \stackrel{?}{=} t_i}}$$

For each clause $\vec{i} \in D : S[\vec{i}] \stackrel{?}{=} t_i$ of the right SARE. $\vec{i}$ *must* be a *symbolic* index vector generated by the **Right Generalize** rule.

**Left Generalize (LG)**

$$\underset{\sigma}{\boxed{S[u(\vec{i})] \stackrel{?}{=} t}} \xrightarrow{\vec{i} \leftarrow u(\vec{i})} \underset{\sigma}{\boxed{S[\vec{i}] \stackrel{?}{=} t}}$$

Where $\vec{i}$ is a *symbolic* index vector of $S$, and $u \neq Id$.

**Right Generalize (RG)**

$$\underset{\sigma}{\boxed{t \stackrel{?}{=} S[u(\vec{i})]}} \xrightarrow{\vec{i} \leftarrow u(\vec{i})} \underset{\sigma}{\boxed{t \stackrel{?}{=} S[\vec{i}]}}$$

Where $\vec{i}$ is a *symbolic* index vector of $S$, and $u \neq Id$.

**Substitute (S)**

$$\underset{\sigma}{\boxed{X(t_1 \dots t_n) \stackrel{?}{=} t}} \longrightarrow \underset{\sigma}{\boxed{\sigma(X(t_1 \dots t_n)) \stackrel{?}{=} t}}$$

When $X$ is defined in $\sigma$.

**Project (P)**

$$\underset{\sigma}{\boxed{X(t_1 \dots t_n) \stackrel{?}{=} t}} \longrightarrow \underset{\sigma \circ [X \mapsto \lambda \vec{x}.x_i]}{\boxed{t_i \stackrel{?}{=} t}}$$

For each $1 \leq i \leq n$. Where $t$ is a term of $\mathcal{T}(\Sigma)$, or an input array $a(\vec{i})$ of the right SARE. This rule is apply when $X$ is *not* defined in $\sigma$.

**Imitate (I)**

$$\underset{\sigma}{\boxed{X(t_1 \dots t_n) \stackrel{?}{=} f(t'_1 \dots t'_p)}} \longrightarrow \underset{\sigma \circ [X \mapsto \lambda \vec{x}.f(X_1(\vec{x}) \dots X_p(\vec{x}))]}{\boxed{f(X_1(t_1 \dots t_n) \dots X_p(t_1 \dots t_n)) \stackrel{?}{=} f(t'_1 \dots t'_p)}}$$

Where $X_1 \dots X_p$ are *fresh* free-variables. This rule is apply when $X$ is *not* defined in $\sigma$.

FIG. 7.5 – Construction rules of the unification automaton

can remark that the imitation would lead to a parametric-length branch, allowing $X$ to absorb the whole term computed by the right SARE. Consequently, we choose to *forbid the imitation*. A precise description of the forbidden imitations is provided thereafter.

Starting from the projection P1, a new state $\boxed{X(S_2[i_T-1], a[i_T]) \stackrel{?}{=} S_2[i_P-1] + a[i_P]}$ is created. It is different from the first colored state since the unifiers are not the same. Since $X$ is already defined in the current unifier, the substitution rule is applied. Finally, the cycle S → D → LC/RC captures the parametric-depth recurrence, while the projection P2 provides an (assumption) of final value for $X$, which is $X(x, y) = x + y$.

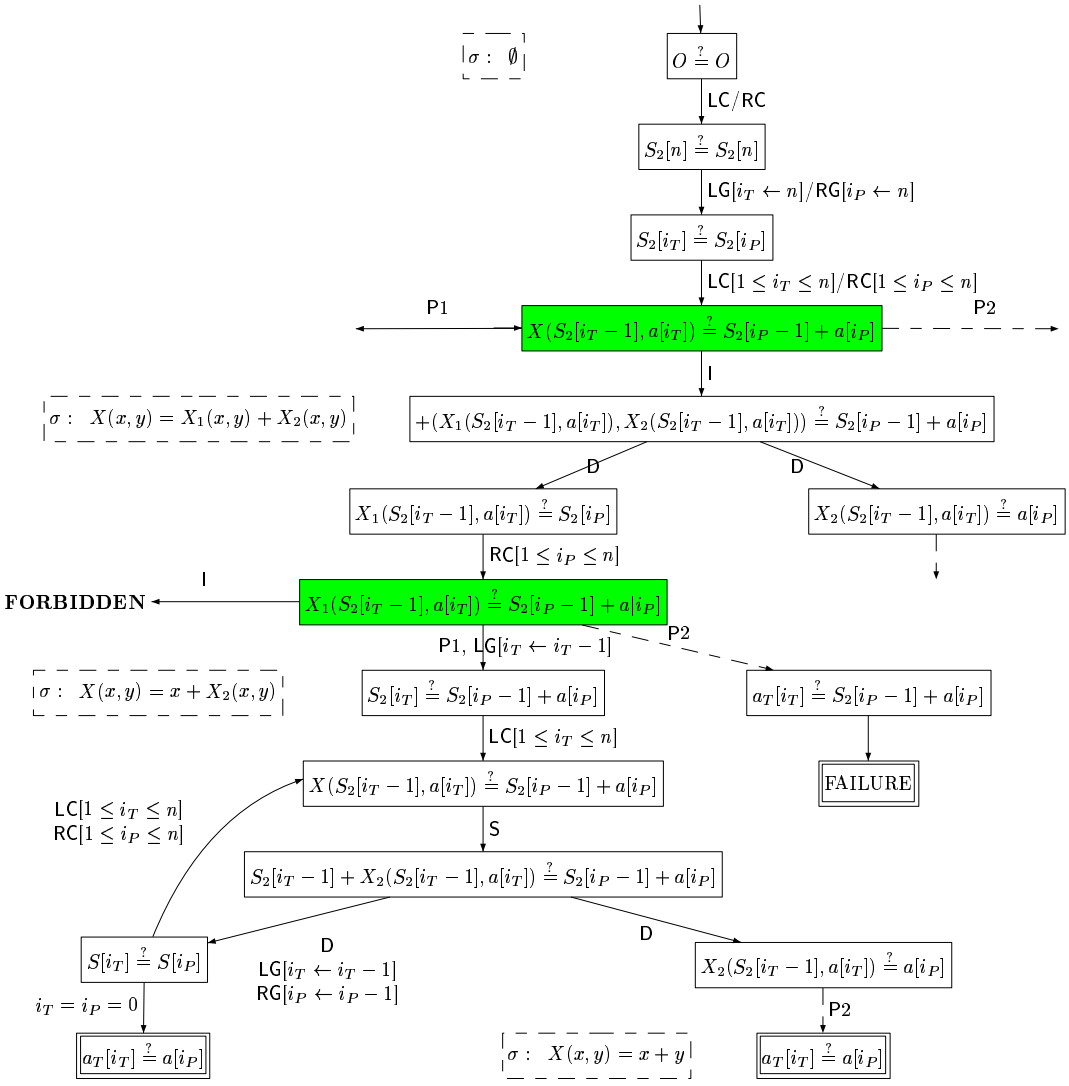



FIG. 7.6 – Unification automaton of the template and the program for any $n$

**Forbidden imitations** This example shows that the imitation rule has to be applied carefully to avoid parametric-length branches. Indeed, recall that we have to solve the following matching problem:

$$\lambda a_T(0) \ldots a_T(n).X(\ldots X(a_T(0), a_T(1)) \ldots) \stackrel{?}{=} \lambda a(0) \ldots a(n). + (\ldots + (a(0), a(1)) \ldots)$$

Each application of the imitation rule adds the head symbol + of the rhs to the current definition of $X$. Since there is a parametric number of +, we will obtained a parametric number of states, leading to absorb the whole rhs.

A solution is to limit the number of application of the imitation to define a given template variable. We choose to *forbid* the imitation of *flexible-rigid* states $\boxed{X'(t_1 \dots t_n) \stackrel{?}{=} t, \sigma}$ such that:

- There exists a predecessor $\boxed{X(t_1 \dots t_n) \stackrel{?}{=} t, \theta}$ of $\boxed{X'(t_1 \dots t_n) \stackrel{?}{=} t, \sigma}$ in the unification automaton *e.g.* colored states in figure 7.6.
- $X'$ is involved in the definition of $X$.

In other words, we forbid a template variable to absorb a term defined by a recurrence. As a consequence, we cannot handle unifiers defined by a loop. We will provide such an example in section 7.7, page 124. Under this condition, the following proposition establishes the finitude of the unification automaton.

**Proposition 7.2 (Finitude of the unification automaton).** *Consider a matching problem $T \stackrel{?}{=} P$, and $\mathcal{A}$ the corresponding unification automaton. Then $\mathcal{A}$ has a finite number of states.*

*Proof.* Let us denote by $S_T$ the template SARE and $S_P$ the program SARE. The states of $\mathcal{A}$ are of the form $t_T \stackrel{?}{=} t_P, \sigma$.

- $t_P$ is one of the possible subterms of the clauses defining the program SARE, which are in finite number.
- $t_T$ is either a subterm of $S_T$, or a free variable $X(t_1 \dots t_n)$ which takes subterms of $S_T$ as arguments. The number of free variables of the template is finite. Moreover the fresh free variable produced by the imitation rule is also finite. Indeed, a parametric number of fresh free-variables would be necessarly produced by the imitation of a recurrence of $S_P$, which is forbidden by the restriction on the imitation.

Finally, the finitude of each hand side ensures the finitude of $\mathcal{A}$. □

## 7.5 Analysis of the Unification Automaton

The unification automaton obtained is a *finite representation* of the unification tree *w.r.t.* the terms computed by the template and the program. Due to the representation, Huet's rules cannot be applied directly. An analysis pass is thus required to decide whether both template and program match, and to compute the unifiers in case of success. In this section, we present an extension of the analysis already described in Chapter 6 to achieve this task.

### 7.5.1 Overview

Figure 7.7 sums up the main steps of the analysis. The first step check whether the template and the program matches, and provides in case of success the values to put in the template variables ($X$ in the motivating example). We then find the relevant mapping between template and program inputs. Each of these steps are described thereafter.

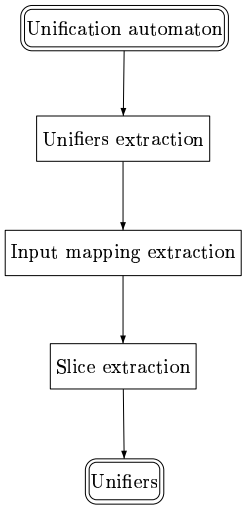



FIG. 7.7 – Overview of the analysis

### 7.5.2 Unifiers Extraction

In the same manner as the equivalence test, the generalization rules abstract array subscripts, and lead to handle the arrays $S[\vec{i}]$ in the general case with a symbolic $\vec{i}$. The compute rules LC and RC are then applied in a systematic manner, and generates *unreachable states*. Unfortunately, the unreachable branches of the automaton may lead to failure states even if the template and the program match. A first task is then to remove the unreachable states from the automaton. Following the method described in section 6.5, page 98, we compute the *reaching set* of each state, then we remove the states whose reaching set is empty. The resulting automaton represents exactly the unification tree of the template and the program, and can handled by applying classical composition rules of Huet and Lang's procedure.

The algorithm described in figure 7.8 summarizes the main steps of the analysis.

**Step 1** removes the unreachable states by applying the method described above. The reaching sets are computed by using the heuristic described in [63] and implemented in the Omega library [62]. It remains then to remove the nodes whose reaching set is empty.

**Steps 2 and 3** check whether the algorithm fails. Following Huet and Lang's algorithm [56], the failures are propagated in a bottom-up manner by handling projection and imitation rules as an OR-branching. Indeed, $S$ will fail if all assumptions made in the $S_i$ fail. In the other hand, the other rules are handled as AND-branching. Intuitively, decompose and compute rules allow to explore the terms computed by the template and

the program ; and it is enough that sub-terms do not match so that the whole terms do not match.

If the algorithm does not fail, the program matches the template. In this case, we compute the corresponding unifier(s) **(step 4)**. Following Huet and Lang algorithm, the idea is to built the set of unifiers in a bottom-up manner, by applying rules allowing to lift the set of unifiers up to the initial state. As stated above, a final state $t \stackrel{?}{=} p, \sigma$ contains a unifier $\sigma$ built from the assumptions made while applying projections and imitation rules. This justifies intuitively the first two rules. The unifiers obtained from the branches of decompose and compute rules have to be gathered with a composition $\circ$ to obtain the whole unifier. For instance, in figure 7.6, the branches of the first decompose rule define separately the values of $X_1$ and $X_2$. Since two branches may define the same variable, we have to check carefully that the definitions are the same. Projection and imitation are construction rules which make assumptions on the value of the unifier. It is thus natural to group the results obtained from each alternative by using union. Applying these rules on the motivating example, we lift the unifier $X(x, y) = x + y$ up to the initial state, and we finally emit it as a unifier **(step 5)**.

---

**Algorithm** *Extract_Unifiers*

---

**Input:** $\mathcal{A}$, *a unification-automaton obtained by applying the rules of figure 7.5*
**Output:** *The set of unifiers* $\mathcal{U}$

1. For each node $n$ of $\mathcal{A}$:
   - Compute the reaching set $r(n)$ of $n$ [semi-decision procedure]
   - If the semi-decision procedure fails, replace $n$ by FAILURE.
   - If $r(n) = \emptyset$, remove $n$ from $\mathcal{A}$
2. Propagate the failure nodes by applying the following rules:
   - If $S \xrightarrow{\text{D /LC/RC/LG/RG/S}}$ FAILURE,
     Replace $S$ by FAILURE.
   - If $S \xrightarrow{\text{P/I}} S_i$ for $1 \leq i \leq n$, and *all* the $S_i$ are a FAILURE,
     Replace $S$ by FAILURE.
3. If the failures are propagated up to the initial state $O \stackrel{?}{=} O$, emit $\boxed{?}$.
4. Remove the return arcs of $\mathcal{A}$, then compute the set of unifiers $\mathcal{U} = \mathcal{U}[\![O \stackrel{?}{=} O]\!]$ by applying the following rules:
   - $\mathcal{U}[\![f() \stackrel{?}{=} f(), \sigma]\!] = \{\sigma\}$
   - $\mathcal{U}[\![a[\vec{i_T}] \stackrel{?}{=} a[\vec{i_P}], \sigma]\!] = \{\sigma\}$
   - If $S \xrightarrow{\text{D /LC/RC}} S_i$ for $1 \leq i \leq n$: $\mathcal{U}[\![S]\!] = \{\sigma_1 \circ \ldots \circ \sigma_n \mid \sigma_i \in \mathcal{U}[\![S_i]\!]\}$
   - If $S \xrightarrow{\text{LG/RG/S}} S'$ for $1 \leq i \leq n$: $\mathcal{U}[\![S]\!] = \mathcal{U}[\![S']\!]$
   - If $S \xrightarrow{\text{P/I}} S_i$ for $1 \leq i \leq n$: $\mathcal{U}[\![S]\!] = \mathcal{U}[\![S_1]\!] \cup \ldots \cup \mathcal{U}[\![S_n]\!]$
5. emit $\boxed{\text{YES, with } \mathcal{U}}$.

---

FIG. 7.8 – *Extract_Unifiers*

### 7.5.3 Input Mapping Extraction

Once the unifiers are found, it remains to provide the relevant values of template inputs. Basically, the unification automaton allows to unroll simultaneously the template and the program until the template inputs are reached. We then obtain a leaf, which links the template input to a program value, which can be either a program input $a(\vec{i_T})$, or a more complex expression $t(\vec{i_P})$ depending on a counter $\vec{i_P}$. Consequently, the input mapping can be obtained by inspecting the leaves.

Figure 7.9 provides the algorithm to recover the input mapping. As stated above, we just collect the informations provided by reachable leaves. The last item expresses a general failure of the unification automaton analysis, we do not know whether the program matches the template.

---

**Algorithm** *Extract_Inputs*

---

**Input:** $\mathcal{A}$, *the unification automaton*
**Output:** $\delta$, *a mapping defining template inputs*

For each leaf $\boxed{a_T(\vec{i_T}) \stackrel{?}{=} t(\vec{i_P})}$:

- Compute the reaching set $R$ of $\boxed{a_T(\vec{i_T}) \stackrel{?}{=} t(\vec{i_P})}$
- For each $(\vec{i_T}, \vec{i_P}) \in R$,
  emit the mapping $\boxed{a_T(\vec{i_T}) \stackrel{\delta}{\mapsto} t(\vec{i_P})}$
- In case of functional incoherence (two different definitions for a given $a_T(\vec{i_T})$), then emit $\boxed{?}$

---

FIG. 7.9 – *Extract_Inputs*

In practice, reaching sets are expressed by a finite set of clauses, despite their parametric size. This guarantee that the mapping emitted has a finite representation. Consider again the unification automaton given in figure 8.5. On the leaf state $\boxed{a_T[i_T] \stackrel{?}{=} a[i_P]}$ at the bottom-left, we obtain the reaching set $\{(i_T = 0, i_P = 0)\}$, leading to emit emit the mapping:

$$a_T(0) \stackrel{\delta}{\mapsto} a_P(0)$$

### 7.5.4 Slice Extraction

The last step of the matching is to provide an exact description of the matched program slice, in the meaning defined in Chapter 2, Section 2.4.

Intuitevely, the unification automaton reaches exactly the operations of the program slice, starting from the operation assigning the output, to the first operations reading the inputs. Since the SAREs are obtained from the algorithm described in Chapter 6, each each array $S[.]$ corresponds to an assignment $S$ in the program.

Consequently, a simple algorithm to recover the program slice would be to compute for each state corresponding to a statement $S$, the set of the program iteration vectors $\mathcal{I}_S$ by using the reaching sets, then to emit $\{\langle S, \vec{i}\rangle, \vec{i} \in \mathcal{I}_S\}$. Such an algorithm is summarized in figure 7.10.

**Algorithm** *Extract_Slice*

**Input:** $\mathcal{A}$, *the unification automaton*
**Output:** $\mathcal{S}$, *the corresponding program slice*

For each state $\boxed{t \stackrel{?}{=} S[\vec{i_P}]}$:

- Compute its reaching set $R = \{(\vec{i_{T1}}, \vec{i_{P1}}) \dots (\vec{i_{Tn}}, \vec{i_{Pn}})\}$
- emit $\boxed{\{\langle S, \vec{i_{P1}}\rangle \dots \langle S, \vec{i_{Pn}}\rangle\}}$.

FIG. 7.10 – *Extract_Slice*

## 7.6 Complexity Issues

In the same manner as the equivalence test, we consider that the significative operation of the algorithm is the computation of reaching sets.

The unification automaton can be viewed as a unification tree chosing the different possible functions as unifiers, and whose leaves are unification automata like those built by the equivalence test. Assume there is $n$ template variables $X_1 \dots X_n$. Each $X_i$ is used in one node at most. Indeed, the project and imitate rules allow to define a value for $X_i$, which will not be redefined in the following transitions. The projections produce $|X_i|$ branches, where $|X_i|$ denotes the arity of $X_i$ ; and the imitation produces one transition. As a consequence, the number of leaves of the unification tree described above is $\Pi_{i=1}^{n}(|X_i| + 1)$.

In the same manner as for equivalence test, the MSA built on the "leaves" of the unification tree has $\mathcal{O}(T \times P)$ nodes at most, where $T$ resp. $P$ denotes the number of assignement in the template resp. the program. Consequently, the total number of nodes of the whole MSA, and thus the worst-case complexity of the instantiation test can be written:

$$\mathcal{O}\left(T \times P \times \prod_{i=1}^{n}(|X_i| + 1)\right)$$

For a given template of size $T$ with free variables $X_1 \dots X_n$, the complexity is thus *linear* in the program size. In most cases, we deal with templates with $n = 1$ free variable and with an arity $|X_1| = 2$, leading to write the complexity $3 \times T \times P$. In this case, the complexity remains of the same order as for the equivalence test. This theoretical study has been confirmed by experimental results obtained on SpecFP 2000 [54] and Perfect Club [39] benchmarks, and presented in Chapter 10.

## 7.7 Discussion

We have presented a new instantiation test based on the equivalence test described in Chapter 6, and the Huet and Lang's semi-unification procedure. In the same manner as the equivalence test, our instantiation test is able to solve partially the matching problem *w.r.t* Herbrand-equivalence, and can handle all program variations described in introduction, excepts the semantics variations. For instance, we can handle the following matching problem, and provide the unifier $X(x, y) = x + 3y + 1$.

```
s = a(0)
do i = 1,n
| s = X(s,a(i))
enddo
return s
```

(a) Template

```
s = a(0)
do i = 1,n
| b = 3*a(i)
| s = s + b + 1
enddo
```

(b) Program

Combining with data-structure and control variations, we are also able to handle the following matching problem, providing the same unifier:

```
s = a(0)
do i = 1,n
| s = X(s,a(i))
enddo
return s
```

(a) Template

```
s(0) = a(0)
do i = 1,n,2
| b = 3*a(i)
| s(i) = s(i-1) + b + 1
| b = 3*a(i+1)
| s(i+1) = s(i) + b + 1
enddo
```

(b) Program

Additionally, our method is able to provide the exact program slice corresponding to the recognized algorithm. At the opposite of many slicing methods, the slice obtained does not contain garbage. Moreover, our method is *linear* in the program size despite a big constant due to the cost of Presburger relations computation. Experimental results on the SpecFP 2000 and Perfect Club benchmarks have shown that the cost can be reduced of 50 % by avoiding the re-computations of Presburger relations, leading to a reasonnable execution time.

Unfortunately, the limitation on the application of the **Imitation** rule makes impossible to detect unifiers defined by a loop. Consequently, the following matching problem cannot be handled:

```
s = a(0)
do i = 1,n
| s = X(s,a(i))
enddo
return s
```

(a) Template

```
s = a(0)
do i = 1,10
| p = 1
| do j = 1,5
| | p = p * a(i)
| enddo
| s = s + p
enddo
```

(b) Program

Indeed, a solution would be:

$$X(x,y) \quad = \quad \begin{array}{l} \texttt{p = 1} \\ \texttt{do j = 1,5} \\ \quad \texttt{p = p * } y \\ \texttt{enddo} \\ \texttt{return } x \texttt{ + p} \end{array}$$

We can handle this problem by allowing systematically $k$ nested imitations, for a given $k$ ($k$-limiting). Even if it would works here for $k = 5$, such a solution has several drawbacks.

- $k$ has to be chosen in an arbitrary manner.
- We cannot handle loops *do* $i = 1, p$ with a parametric number of iterations.
- The $k$-limiting increases the size of the automaton, and thus the execution time of the algorithm.
- Our method will provide here the expression $X(x,y) = x + ((((y \times y) \times y) \times y) \times y)$ without recovering the loop.

The next chapter proposes another instantiation test able to provide unifiers expressed with general `do` loops, without these drawbacks. It will also be able to handle general programs, breaking the static control restriction.

# Chapter 8

# Template Matching with Tree-Automata

The previous chapter proposes a solution to the template matching problem by extending the equivalence test described in chapter 6 with the construction rules of Huet and Lang matching procedure. Though the obtained algorithm can detect a large amount of program variations, it cannot cope with matching problems whose unifiers are defined by a loop. Additionally, it is limited to static control programs.

In this chapter, we present an alternative instantiation test able to break these limitations. Following the slicing method, we associate to the template and the program a tree-automaton recognizing *exactly* their computed terms. The matching is then achieved by checking their intersection emptiness. The main difficulty comes from the parametric size of the two tree automata, that makes the intersection emptiness undecidable. A finite-size representation of the tree automata is proposed, and an heuristic is described, allowing to handle successfully a significant amount of instances of this difficult problem.

Particularly, we show that parametric-length branches of the cartesian product are repetitions of finite size schemas that can be captured using simple cycles. This allows to represent the cartesian product by a classical finite state automaton. Several reachability problems are also raised, and require the computation of transitive closures of Presburger relations. Experimental results will show that the heuristic used (Omega) can handle them in most cases.

This chapter is organized as follows: Section 8.1 presents the motivating example used to illustrate our algorithm, described precisely in the following sections. Section 8.3 presents the finite size representation of the exact tree automata, then Sections 8.4 and 8.5 describe the intersection emptiness test. Section 8.6 presents a complexity analysis of the instantiation test. Section 8.7 provides some experimental results with some practical improvements. Finally, Section 8.8 summarizes the advantages and the drawbacks of the method, and concludes by presenting possible improvements.

## 8.1 Motivating Example

Consider the following matching problem between a template (a) and a candidate program slice (b):

```
s = I(0)
do i = 1,n
| input = I(i)
| s = X(s,input)
enddo
return s
```

(a) Template

```
s = a(0)
do i = 1,10
| p = 1
| do j = 1,5
| | p = p * a(i)
| enddo
| s = s + p
enddo
```

($S_{end}$)

(b) Program slice

The output statement provided by the slicing method is denoted by $S_{end}$. The template matches the reductions over the input array `I` with a parametric size `n`, by using the generic operator $X$ (template variable). The program slice computes the expression $\sum_{i=1}^{10} a(i)^5$ by using two nested loops. Our instantiation test returns the relevant values of template parameters $X$, `I` and `n` making it semantically equivalent to the slice. Here, a solution would be:

$$X(x,y) = \begin{array}{l}\texttt{p = 1}\\ \texttt{do j = 1,5}\\ \texttt{| p = p * } y\\ \texttt{enddo}\\ \texttt{return } x \texttt{ + p}\end{array}$$

$$n = i \text{ (first loop counter of the program)}$$

$$I(k) = a(k) \text{ for } k \text{ between 1 and } i$$

## 8.2 Overview of the Method

Following the slicing method, the instantiation described in this chapter checks the intersection emptiness of the *exact* tree automata of the template and the program.

Roughly speaking, the instantiation test steps simultaneously the template and slice statements going up the data-flow dependences as long as the same operators are found. If the stepping leads to different operators, the instantiation test fails. Whenever a free variable is reached, program operators are absorbed (or not) in a non-deterministic manner, similarly to Huet and Lang's Project and Imitate rules.

The stepping use the *exact* tree-automata $\mathcal{A}_T$ and $\mathcal{A}_P$ built from the template and the slice (see section 8.3), and is perform by computing their cartesian product $\mathcal{A}_T \times \mathcal{A}_P$, backward starting from the final state (see section 8.4) . If the slice matches the template, the template variables values are built by extracting and analyzing the parts of $\mathcal{A}_T \times \mathcal{A}_P$ unrolling the free variables of the template on the slice (see section 8.5).

## 8.3 Construction of $\mathcal{A}_T$ and $\mathcal{A}_P$

In this section, we present the construction of the exact tree automata for the template and the program. The construction rules given in chapter 5 are used, and a simplification is applied for the sake of clarity.

Recall that the construction rules of the exact tree automata were given, and justified in figure 5.8, page 78.

Figure 8.2 gives the *symbolic* expression of the exact tree automata of the template (c), and the program (d), *without computing the reaching definitions* RD. Once the reaching definitions computed, we obtained the tree automata given in (e) and (f). In fact, they are a finite representation of tree automata with a parametric number of rules. For instance, the input transition $\langle T_2, i-1\rangle \to \langle T_2, i\rangle$, $i \in \{2 \dots n\}$ represents $n-1$ transitions.

Notice that these tree automata are similar to the SARE defined in section 6.1.2, page 91. Indeed, it is enough to swap left and right hand side of the rules to obtain the SAREs.

Unfortunately, to obtain (e) and (f), we need to compute the reaching definitions of the whole program, whereas those of the slice would be enough. This approach is not realistic, since the whole program has to be static control (to make the RD computable). Additionally, the RD computation is expensive in time. The idea investiguated in this chapter is to keep the RD unevaluated, and to compute them *on-demand*, while computing the cartesian product $\mathcal{A}_T \times \mathcal{A}_P$. Such an approach allows to handle *static control slices* in *general programs*, and reduces drastically the cost of RD computation. Indeed, only RD involved in the slice would need to be computed.

In order to simplify the notations (and particularly in the tree automata presented thereafter), we will denote the reaching definition by symbols $\Phi_i$, as shown in figure 8.1. $\Phi_X$ is the reaching definition that control the loop computing the composition of the template variables $X$. We use the same notation for $\Phi_+$ and $\Phi_\times$ in the program tree automaton. In both automata, $\Phi_V$ is the reaching definition giving the operation $\langle V, .\rangle$. $\Phi_{P_2}$ in another simple reaching definition giving the last instance of $P_2$ ($\langle P_2, i, 5\rangle$).

$$
\begin{array}{rcl}
I(0) & \longrightarrow_1 & \langle T_1, \rangle \\
I(i) & \longrightarrow_2 & \langle V, i\rangle \\
\Phi_X & \longrightarrow_3 & \langle T_2, i\rangle \\
\Phi_V & \longrightarrow_4 & \langle T_2, i\rangle \\
1 & \longrightarrow_5 & \langle T_2, i\rangle \\
+(\langle T_2, i\rangle, \langle T_2, i\rangle) & \longrightarrow_6 & \langle T_2, i\rangle \\
\times(\langle T_2, i\rangle, \langle T_2, i\rangle) & \longrightarrow_7 & \langle T_2, i\rangle \\
\langle T_2, n\rangle & \longrightarrow_8 & \langle T_{end}\rangle
\end{array}
$$

(a) Template automaton

$$
\begin{array}{rcl}
a(0) & \longrightarrow & \langle S_1, \rangle \\
1 & \longrightarrow & \langle P_1, i\rangle \\
a(i) & \longrightarrow & \langle V, i, j\rangle \\
\times(\Phi_\times, \Phi_V) & \longrightarrow & \langle P_2, i, j\rangle \\
+(\Phi_+, \Phi_{P_2}) & \longrightarrow & \langle S_2, i, j\rangle
\end{array}
$$

(b) Program automaton

FIG. 8.1 – Simplified notation for the RD

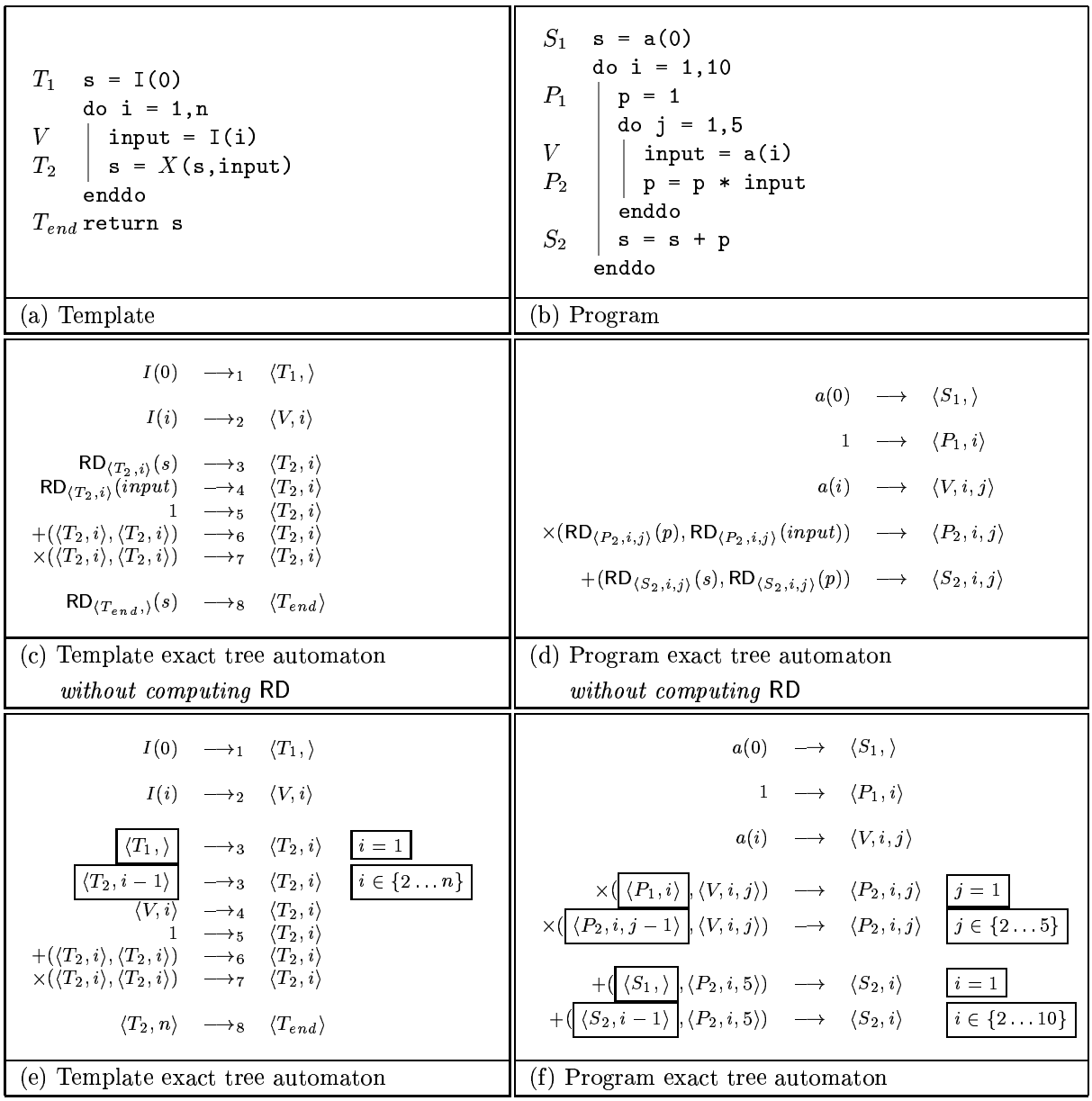



FIG. 8.2 – Exact tree automata of the template and the program

## 8.4 Construction of $\mathcal{A}_T \times \mathcal{A}_P$

Once $\mathcal{A}_T$ and $\mathcal{A}_P$ are built, it remains to step them simultaneously, in order to decide whether the program slice $P$ is an instance of the template $T$. In this section, we describe an algorithm to achieve this stepping. The cartesian product $\mathcal{A}_T \times \mathcal{A}_P$ summarizing the stepping is obtained, and will be analyzed by applying an algorithm described in the next section. We recall the following proposition, already given in chapter 5, page 78, that justifies the computation of $\mathcal{A}_T \times \mathcal{A}_P$ to decide whether $T$ is an instance of $P$.

**Proposition 8.1 (Correction).** *Consider a matching problem $T \stackrel{?}{=} P$ where $T$ is a template and $P$ is a program. The following assertions are equivalent:*

*(i) $T \stackrel{?}{=} P$ has a solution* w.r.t. $\equiv_{\mathcal{H}}$

*(ii) For each input $I_P$ of $P$, there exists an input $I_T$ of $T$ such that:*

$$\mathcal{L}(\mathcal{A}_T(I_T)) \cap \mathcal{L}(\mathcal{A}_P(I_P)) \neq \emptyset$$

*Proof.* Part (ii) $\Rightarrow$ (i). Assume (ii). Since $\mathcal{L}(\mathcal{A}_P(I_P)) = \{\mathcal{T}_P(I_P)\}$, $\mathcal{T}_P(I_P)$ is recognized by $\mathcal{A}_T(I_T)$. Hence, there exists an instance $T_I$ of $T$ such that $\mathcal{T}_{T_I}(I_T) = \mathcal{T}_P(I_P)$. Denoting by $\sigma$ the mapping from $I_P$ to $I_T$ stated by (ii), we can write: $\mathcal{T}_{T_I}(\sigma(I_P)) = \mathcal{T}_P(I_P)$. Thus $T_I \circ \sigma \equiv_{\mathcal{H}} P$. This leads to (i), modulo $\sigma$.
Part (i) $\Rightarrow$ (ii). Assume now (i). Consequently, there exists an instance $T_I$ of $T$ such that $T_I \equiv_{\mathcal{H}} P$. Then $\mathcal{T}_{T_I}(I) = \mathcal{T}_P(I)$ for each relevant input $I$ (definition 2.9, page 51). Since we have $\mathcal{T}_{T_I}(I) \in \mathcal{L}(\mathcal{A}_T(I))$, and $\mathcal{T}_P(I) \in \mathcal{L}(\mathcal{A}_P(I))$, the result follows with the identity mapping. □

The classical solution to check the intersection emptiness of $\mathcal{L}(\mathcal{A}_T(I_T))$ and $\mathcal{L}(\mathcal{A}_P(I_P))$ is to compute the cartesian product $\mathcal{A}_T(I_T) \times \mathcal{A}_P(I_P)$, and to verify that the final state is reachable.

In the general case, the template can output an array $O[\vec{i}]$ with several dimensions. In the tree-automaton, it appears on the final state, which depends on the output index $\vec{i}$. To be rigorous, we should consider the tree-automaton $\mathcal{A}(I)$ as a *family* of tree-automata $[\mathcal{A}(I)_{\vec{i}}]_{\vec{i} \in D}$, where each element $\mathcal{A}(I)_{\vec{i}}$ is the tree-automaton recognizing the $\vec{i}$-th element of the output array, and $D$ denotes the index domain of the output array. Consequently, the general case of our problem is to compare the families of tree-automata $[\mathcal{A}_T(I_T)_{\vec{i}}]_{\vec{i} \in D_T}$ and $[\mathcal{A}_P(I_P)_{\vec{i}}]_{\vec{i} \in D_P}$. An ideal solution should be to find a bijective mapping $\delta$ from $D_P$ to $D_T$ and to compare $\mathcal{A}_T(I_T)_{\vec{i}}$ to $\mathcal{A}_P(I_P)_{\delta(\vec{i})}$, for each relevant output index $\vec{i}$. This leads to handle the three following cases:

- $\underline{\dim D_T = \dim D_P}$ In this case, we try the identity mapping.
- $\underline{\dim D_T < \dim D_P}$ This case occurs when the output of the slice is nested in a number of loops greater than $\dim D_T$. In the motivating example, this case occurs with the loop $i$. In this case, we identify two-by-two the dimensions of $T$ to the last (more nested) dimensions of $P$. The remaining dimensions of $P$ are handled as structure parameters.

- $\underline{\dim D_T > \dim D_P}$ In this case, we cannot find a restriction of the program slice's output which have the same dimension of the template's output. Since the outputs are not compatibles, we consider that the template and the program do not match, and we yield a failure.

Once the mapping $\delta$ is found, we compute the cartesian product $\mathcal{A}_T(I_T)_{\vec{i}} \times \mathcal{A}_P(I_P)_{\delta(\vec{i})}$ for a *symbolic* $\vec{i}$ (as inputs $I_T$ and $I_P$). Unfortunately, the classical algorithm cannot be applied, since the two tree-automata can have a parametric number of rules. Figure 8.3 provides the general schema of the cartesian product, obtained while starting the construction backward, from the final state $\boxed{\langle T_{end}, \rangle, \langle S_2, [i] \rangle}$. The cartesian product is basically constituted of different parts unrolling $X$ on $I_P(i)^5$, then $I_P(i-1)^5$, then $I_P(i-2)^5$, until $I_P(1)^5$. Each part allowing $X$ to absorbs the needed $\times$ operations. Reaching definitions are computed *on-demand* during the construction. The arrows with dotted lines represent a decomposition rule, and detail the transitions needed to obtain each argument of an operator (here $+$ and $\times$).

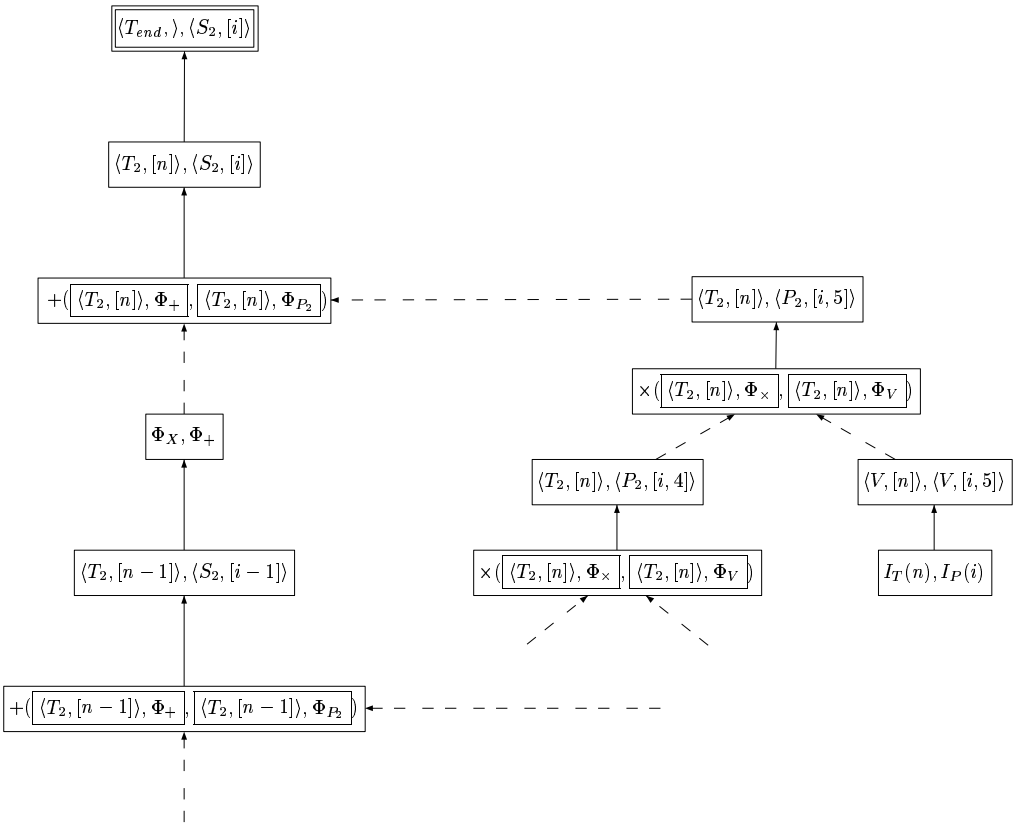



FIG. 8.3 – The last states of $\mathcal{A}_T(I_T)_{\vec{i}} \times \mathcal{A}_P(I_P)_{\delta(\vec{i})}$

Following the equivalence test, we handle the parametric size of the cartesian product by introducing a *generalization rule*, allowing to describe only one time, in the *general case*, the part unrolling $X$ on $I_P(k)^5$ with a *symbolic* $k$, $1 \leq k \leq i$. This allows to capture the parametric-size cartesian product into a finite-size MSA, whose construction rules are summarized thereafter.

- Template $\varepsilon$-compute: From a state $\boxed{\langle T, \vec{i_T}\rangle, \bullet}$, where $\bullet$ is a generalized state $\langle S, \vec{i_P}\rangle$ or a symbol $\Phi$ ; with an $\varepsilon$-transition $T' \longrightarrow \langle T, \vec{i_T}\rangle$ in $\mathcal{A}_T$, add the transition:

$$\boxed{\langle T, \vec{i_T}\rangle, \bullet} \longrightarrow \boxed{T', \bullet}$$

- Program $\varepsilon$-compute: From a state $\boxed{\bullet, \langle P, \vec{i_P}\rangle}$, where $\bullet$ is a generalized state $\langle T, \vec{i_T}\rangle$ or a symbol $\Phi$ ; with an $\varepsilon$-transition $P' \longrightarrow \langle P, \vec{i_P}\rangle$ in $\mathcal{A}_P$, add the transition:

$$\boxed{\bullet, \langle P, \vec{i_P}\rangle} \longrightarrow \boxed{\bullet, P'}$$

- Full compute: From a state $\boxed{\langle T, \vec{i_T}\rangle, \langle P, \vec{i_P}\rangle}$, with the transitions $f(\bullet_1 \dots \bullet_n) \longrightarrow \langle T, \vec{i_T}\rangle$ in $\mathcal{A}_T$ and $f(\circ_1 \dots \circ_n) \longrightarrow \langle P, \vec{i_P}\rangle$ in $\mathcal{A}_P$, add the transition:

$$\boxed{\langle T, \vec{i_T}\rangle, \langle P, \vec{i_P}\rangle} \longrightarrow f(\boxed{\bullet_1, \circ_1} \dots \boxed{\bullet_n, \circ_n})$$

- Decompose: From a state $f(\boxed{\bullet_1, \circ_1} \dots \boxed{\bullet_n, \circ_n})$, add the transitions:

$$f(\boxed{\bullet_1, \circ_1} \dots \boxed{\bullet_n, \circ_n}) \longrightarrow \boxed{\bullet_i, \circ_i}$$

  for $1 \leq i \leq n$.

- Template generalization: From a state $\boxed{\langle T, u(\vec{i_T})\rangle, \bullet}$ with $u \neq \mathrm{id}$, add a transition:

$$\boxed{\langle T, u(\vec{i_T})\rangle, \bullet} \xrightarrow{\vec{i_T} \leftarrow u(\vec{i_T})} \boxed{\langle T, \vec{i_T}\rangle, \bullet}$$

- Program generalization: From a state $\boxed{\bullet, \langle P, u(\vec{i_P})\rangle}$ with $u \neq \mathrm{id}$, add the transition:

$$\boxed{\bullet, \langle P, u(\vec{i_P})\rangle} \xrightarrow{\vec{i_P} \leftarrow u(\vec{i_P})} \boxed{\bullet, \langle P, \vec{i_P}\rangle}$$

- Template $\Phi$-compute: From a state $\boxed{\Phi, \bullet}$ apply if possible the exact algorithm described in [43] to get the exact definition of $\Phi$:

$$\Phi = T_k \quad \vec{i_P} \in D_k$$

  If the algorithm cannot be applied, yield a $\boxed{\text{FAILURE}}$.

  Else, add the transitions:

$$\boxed{\Phi, \bullet} \xrightarrow{\vec{i_T} \in D_k} \boxed{T_k, \bullet}$$

  For each $k$.

- Program Φ-compute: From a state $\boxed{\bullet, \Phi}$ apply if possible the exact algorithm described in [43] to get the exact definition of $\Phi$:

$$\Phi = P_k \ \ \vec{i_P} \in D_k$$

If the algorithm cannot be applied, yield a $\boxed{\text{FAILURE}}$.

Else, add the transitions:

$$\boxed{\bullet, \Phi} \xrightarrow{\vec{i_T} \in D_k} \boxed{\bullet, P_k}$$

For each $k$.

Since we want to check whether the final state of the cartesian product is reachable (intersection emptiness), we construct the cartesian product *backward*, from the final state to the initial states (constant leaves $f()$, and states with a template input $\boxed{a_T[.], \bullet}$).

Rules Template $\varepsilon$-compute, Program $\varepsilon$-compute, Full compute and Decompose are classical construction rules of cartesian product. Rules Template generalization and Program generalization to capture the parametric-length branches with a cycle. As stated below, these rules guarantee the finiteness of the MSA. Finally, the rules Template Φ-compute and Program Φ-compute allow to compute *on-demand* the reaching definitions needed to achieve the comparison. These rules are quite important since they allow to compute *only* the reaching definitions of the slice. The remainder of the program may contain uncomputable reaching definitions, it will not hurt the matching. As a consequence, our instantiation test is able to handle programs with uncomputable reaching definitions, including programs with `while` loops and non-affine conditionals.

We will consider that a reaching definition is computable whenever Feautrier's algorithm can be applied. The following definition presents a simple test to check this property.

**Definition 8.1 (Static control-dependent statement).** *A statement is said to be* static control-dependent *whenever all its surrounding control structures verify the constraints defined in the polytope model. More precisely:*

- `do` *loops with affine bounds*
- *Conditionals involving affine constraints over loop counters and structure parameters.*

Consider the example given on figure 8.4. The static control-dependent statements are `S1`, `S3`, `S4` and `S5`. `S2` is not static control-dependent since it depends on a `while` loop. While computing a reaching definition of a variable $v$, Feautrier's algorithm will handle all statements writing $v$. It will be appliable whenever each of these statements are static control-dependent. On the example given figure 8.4, the reaching definition $\mathsf{RD}_{\langle S_5,i,j,k\rangle}(s1)$ of `s1` in $S_5$ is computable since the statements writing `s1`, $S_1$ and $S_3$ are static control-dependent. However, the reaching definition $\mathsf{RD}_{\langle S_5,i,j,k\rangle}(s2)$ of `s2` in $S_5$ is not computable, since it needs $S_2$, which is *not* static control-dependent. Indeed, it depend on the definitions in the `while` loop.

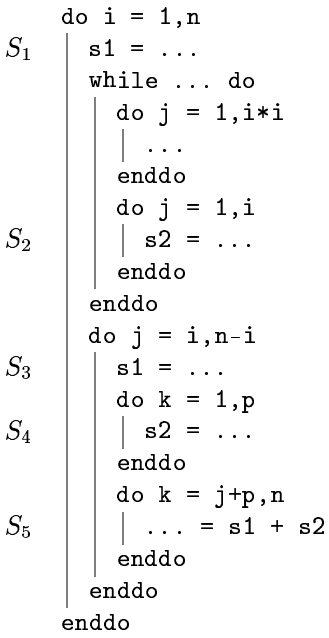



FIG. 8.4 – General loop nest: *static control dependent* statements

The following proposition shows that the number of states of the MSA is always finite.

**Proposition 8.2 (Finitude of the MSA).** *Let $T \stackrel{?}{=} P$ be matching problem where $T$ is a template and $P$ is a program. Then the MSA of $\mathcal{A}_T \times \mathcal{A}_P$ has a finite number of states.*

*Proof.* The states of the MSA are either as $f(\boxed{\bullet_1, \circ_1} \dots \boxed{\bullet_n, \circ_n})$, or as $\boxed{\bullet, \circ}$. The states $f(\boxed{\bullet_1, \circ_1} \dots \boxed{\bullet_n, \circ_n})$ are produced by the Full compute rule. They are obtained by combining rules of $\mathcal{A}_T$ and $\mathcal{A}_P$, which are in finite number. In addition, the compute rules impose the states to generalized to be applied ($\langle S, \vec{i} \rangle$, and not $\langle S, u(\vec{i}) \rangle$). $\bullet$ and $\circ$ are thus sub-terms of a clause of $\mathcal{A}_T$ or $\mathcal{A}_P$, which are in finite number. □

Figure 8.5 provides the MSA built from the motivating example. The construction is performed backward from the final state. As stated before, the generalizations over the states with $+$, and $\times$ allow to capture the parametric branches with a finite number of states. $\phi$-functions $\Phi_X$, $\Phi_+$ and $\Phi_\times$ are computed on-demand during the unrolling. Each colored node makes an assumption on the value of $X$, that we will describe accurately in the following sections. For presentation reasons, we just provide the transitions leading to the solution.

## 8.5 Analysis of $\mathcal{A}_T \times \mathcal{A}_P$

Once the MSA of $\mathcal{A}_T \times \mathcal{A}_P$ is built, it remains to analyze it in order to decide whether the program slice is an instance of the template, and to provide the unifiers in case of success. In this section, we describe a method to handle the MSA and to yield the unifiers as SAREs.

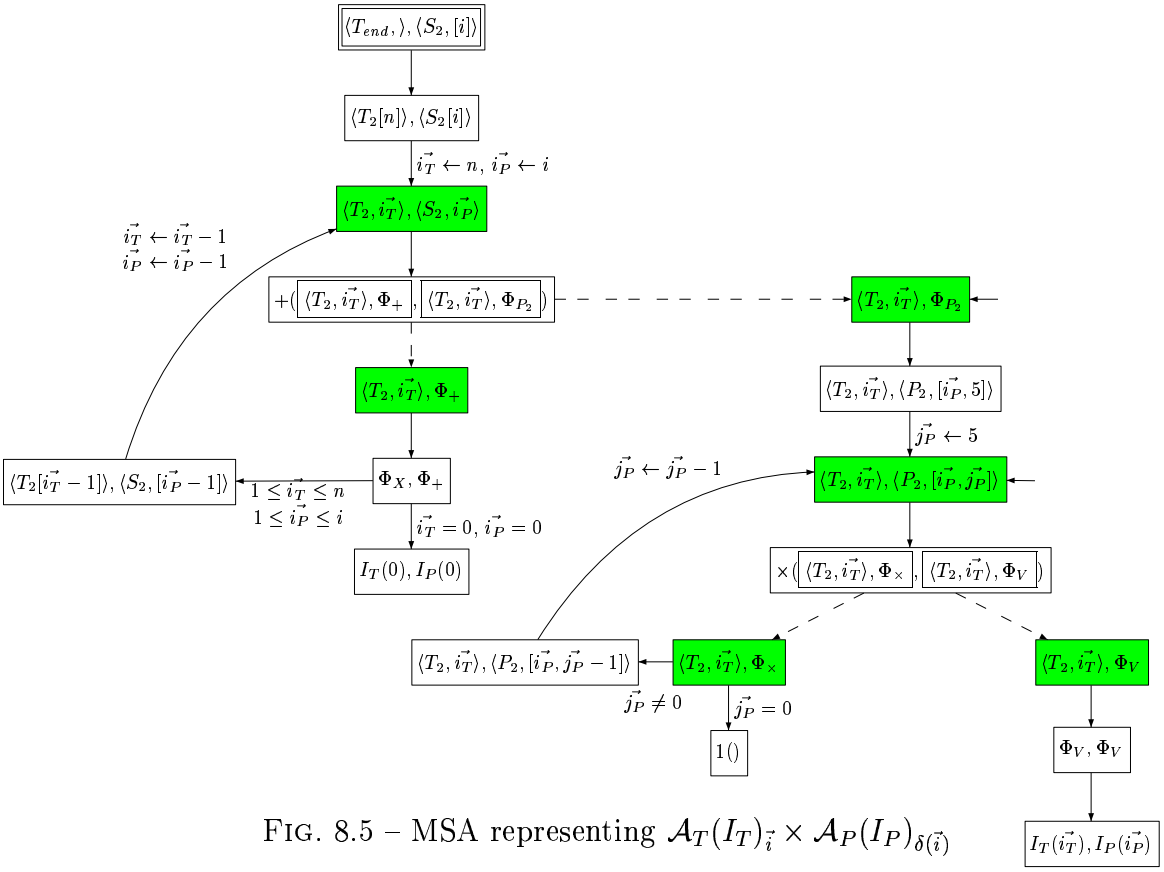



FIG. 8.5 – MSA representing $\mathcal{A}_T(I_T)_{\vec{i}} \times \mathcal{A}_P(I_P)_{\delta(\vec{i})}$

### 8.5.1 Overview of the Analysis

Figure 8.6 sums up the main steps of the analysis. A first step is to generate from the MSA all possible OR-branchings corresponding to different assumptions on template variables. We then check each of these OR-branching, by verifying if they do not leads to an operator conflict. In this situation, we check another OR-branching. Otherwise, we recover the definition of the template from the MSA that we emit as a solution. We describe thereafter each of these steps.

### 8.5.2 OR-branchings Generation

The template tree-automaton contains non-deterministic transitions allowing to recognize each possible instance of the template. During the construction of the MSA, these transitions are fired *on-demand* to imitate the program slice. Several assumptions are always made between (a) imitate the slice (looping transitions) and (b) stop the definition of $X$ and continue to step the remaining of the template. This last choice is achieved by firing *input transitions* allowing to reach an argument of the template variable. Since these transitions are fired from the same node in a non-deterministic manner, we call them *OR-branchings*.

The nodes starting an OR-branching have been colored in figure 8.5. One distinguish *generalized*-OR-branchings starting from nodes $\boxed{\langle S_2, \vec{i_T} \rangle, \langle P, \vec{i_P} \rangle}$ and $\Phi$-OR-branchings

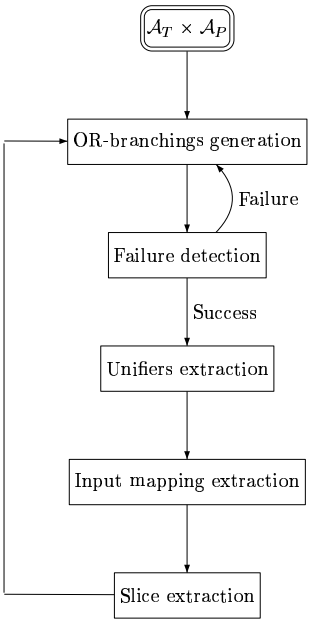



FIG. 8.6 – Overview of the analysis

starting from nodes $\boxed{\langle S_2, \vec{i_T}\rangle, \Phi}$ where $\Phi$ is a program $\phi$-function.

Figure 8.7 provides the detail of a $\Phi$-OR-branching followed by a generalized-OR-branching. The $\Phi$-OR-branching allows to choose between unrolling $X$ on the states resulting from the application of Program $\Phi$-compute, or stopping the unrolling thanks to Template $\varepsilon$-compute transition. In the same manner, the generalized OR-branching is built from rule Full compute, which allow to add a $\times$ to the current definition of $X$ ; and rules Template $\varepsilon$-compute allowing to stop the unrolling.

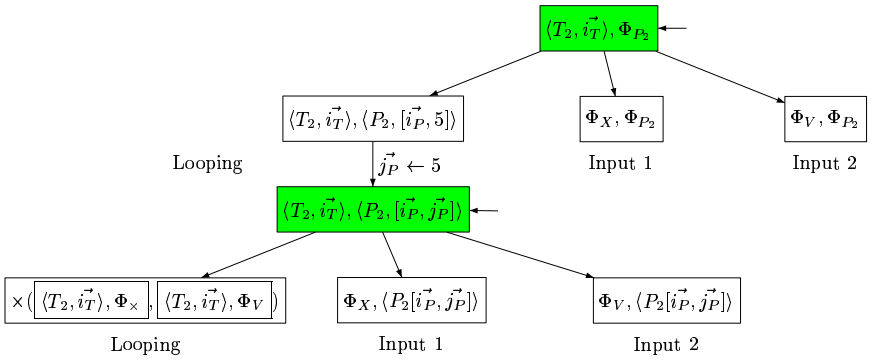



FIG. 8.7 – Detail of two OR-branchings

As stated in proposition 8.1, the program slice matches the template if the final state of $\mathcal{A}_T \times \mathcal{A}_P$ is reachable from the initial state(s) (constants, and inputs $I_T, I_P$). This occurs when there exists an unrolling of the MSA that reachs leave states (1() or inputs) without operator conflicts. Such an unrolling chooses the right transitions in the OR-

branchings, to build the template variable $X$. Given a «choice of OR-branchings» leading to a complete unrolling, one should recover the relevant definition of $X$. To simplify the problem, we assume that the template variable computes the same term at each iteration. This leads to consider that the *branches chosen in the OR-branchings are always the same*. The approach investigated in this thesis is to handle each combination of OR-branchings, and try to recover the corresponding value of $X$. The relevant combination for the motivating example is given in figure 8.5.

Consider again the two OR-branchings given in figure 8.7. Denoting by $\Phi_L$, $\Phi_{I_1}$ and $\Phi_{I_2}$ the looping and the input transitions of the Φ-OR-branching, and by $G_L$, $G_{I_1}$ and $G_{I_2}$ the same branches for the generalized OR-branching, a systematic generation would produce:

$$\Phi_L\ G_L,\ \Phi_L\ G_{I_1},\ \Phi_L\ G_{I_2},\ \Phi_{I_1},\ \Phi_{I_2}$$

One remark that $\Phi_L\ G_{I_1}$ and $\Phi_L\ G_{I_2}$ are redondant with $\Phi_{I_1}$ and $\Phi_{I_2}$ since they lead to the same definition of $X$. As a consequence, we have just to consider the following OR-branchings:

$$\Phi_L G_L, \Phi_{I_1}, \Phi_{I_2}$$

and thus, we discard the branches $G_{I_1}$ and $G_{I_2}$. In a general manner, Φ-OR-branchings starting from a state $\boxed{\langle S_2, \vec{i_T}\rangle, \Phi}$ are immediatly followed by a Program Φ-compute then a Program generalization leading to a state of the form $\boxed{\langle S_2, \vec{i_T}\rangle, \langle P, \vec{i_P}\rangle}$ which starts a generalized OR-branching. Conversely, one can shows that a generalized OR-branching follows a Φ-OR-branching in the same way, except for the first generalized OR-branching starting the unrolling of $X$. This important property lead to vary *only* Φ-*OR-branchings*, while enabling the full compute transition of the generalized OR-branchings.

### 8.5.3 Failure Detection

Once a combination of OR-branchings is chosen, we first check whether no failure state is reachable, then we extract from the MSA the parts unrolling the templates variables, in order to recover these definitions.

As stated in proposition 8.1, the program slice matches the template if the final state of $\mathcal{A}_T \times \mathcal{A}_P$ is reachable from the initial state(s) (constants, and inputs $I_T, I_P$). Since the MSA unrolls the term computed by the template $\mathcal{T}_T(I)$ on the term computed by the program slice $\mathcal{T}_P(I)$, identifying template variables to sub-trees of $\mathcal{T}_P(I)$. When $T$ and $P$ do not match, the unrolling stops before covering all the term $\mathcal{T}_P(I)$, and produces a failure due to operator conflict (see transition Full compute). Hence, we have to check whether an operator conflict is reachable from the final state, or not.

Since the Φ-compute rules choose systematically *all* the definitions of $\phi$-functions without taking into account subscripts values, some transitions constraints may be always false, leading to *unreachable* states. Before analyzing the MSA, the first step is thus to clean it by removing the unreachable states. This can be achieved in the same manner as our first method describe in chapter 7. For each state, we basically compute a regular expression whose letters are affine relations. Applying the algorithm described in [63], we obtain the *reaching set* of the state, which is the set of all possible subscript values while stepping all

possible paths from the initial state. If the reaching set is empty, the state is unreachable, and we remove it.

If there remains an operator conflict, then the OR-branching does make the proper choices for the template variable, and we check another OR-branching. Otherwise, we extract the unifiers from the MSA, by applying the method described thereafter.

### 8.5.4 Unifiers Extraction

In order to recover the value of the template variables from the MSA, we extract each part of the MSA unrolling the program on a template variable $X$. This can be achieved by filtering the MSA states with a template variable state. The motivating example has one template variable state, which is $\langle S_2, . \rangle$. Each connected part is then handled as a sub-automaton, whose initial state is the first state unrolling the template variable. Formally, its predecessors have no template variable states. If several states verify this property, we consider as many different automata as initial states. The final states starts the input transitions, leading to template variable parameters.

Given the OR-branching provided in figure 8.5, we obtain the sub-automaton provided in figure 8.8. We have put an arrow above the initial state $\boxed{\langle S_2, \vec{i_T} \rangle, \langle S_2, \vec{i_P} \rangle}$. In addition, the final states $\boxed{\Phi_X, \Phi_+}$ and $\boxed{\Phi_V, \Phi_V}$ are surrounded.

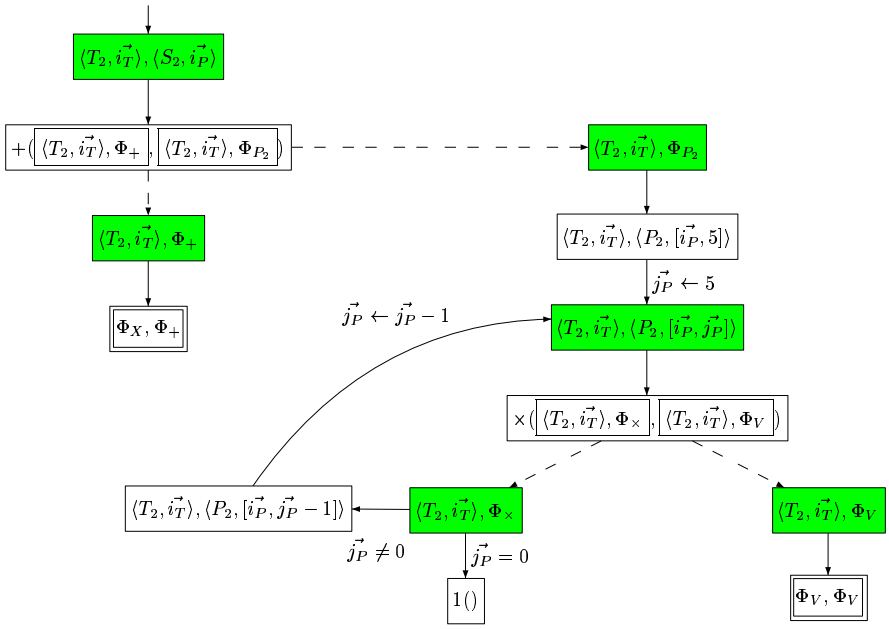



FIG. 8.8 – Sub-automaton unrolling $X$ on the program slice

Once the set of sub-automata $\{\mathcal{A}_1 \dots \mathcal{A}_n\}$ is found, it remains to compute the set of corresponding unifiers. Figure 8.9 provides an algorithm to compute the unifier corresponding to a sub-automaton. The unifier is expressed by a system of affine recurrence equations (SARE), a dataflow representation of a program introduced in chapter 6. The symbolic iteration vector of $Q = (q_T, q_P)$ is the concatenation of the iteration vectors of

the statements corresponding to $q_T$ and $q_P$. For instance, the iteration vector of the first state is $[\vec{i_T}, \vec{i_P}]$. The generation of the SARE clauses is driven by the transitions of the sub-automaton. The generation stops on the leaf states reaching an argument of $X$. For example, $\boxed{\Phi_X, \Phi_+}$ reaches the first argument of $X$ ($\Phi_X$) that we write $x_1$ (step (f)). In the same manner, $\boxed{\Phi_V, \Phi_V}$ reaches the second argument of $X$, denoted $x_2$.

---

**Algorithm** *MSA_To_SARE*

---

**Input:** *$\mathcal{A}_X$, a sub-automaton of the MSA which unrolls the program on $X$.*
**Output:** *$S_X$, a SARE providing the definition of $X$.*

1. If $Q = (q_T, q_P)$ is a state of $\mathcal{A}_X$, we write $Q[\vec{i}]$ to recall that $\vec{i}$ is the (symbolic) iteration vector of $Q$ in $\mathcal{A}_X$.
2. (a) Let $Q[\vec{i}]$ be the initial state of $\mathcal{A}_X$. Emit the clause:

$$\boxed{\forall \vec{i} : X(x_1 \dots x_n)(\vec{i}) = Q(\vec{i})}$$

   (b) For each transition $\varepsilon$-compute:

$$Q[\vec{i}] \longrightarrow Q'[\vec{i}]$$

   Emit the clause:

$$\boxed{\forall \vec{i} : Q(\vec{i}) = Q'(\vec{i})}$$

   (c) For each transition full compute:

$$Q[\vec{i}] \longrightarrow f(Q_1 \dots Q_n)[\vec{i}]$$

   Emit the clause:

$$\boxed{\forall \vec{i} : Q(\vec{i}) = f(Q_1(\vec{i}) \dots Q_n(\vec{i}))}$$

   (d) For each transition generalize:

$$Q[\vec{i}] \xrightarrow{\vec{i} \leftarrow u(\vec{i})} Q'[\vec{i}]$$

   Emit the clause:

$$\boxed{\forall \vec{i} : Q(\vec{i}) = Q'(u(\vec{i}))}$$

   (e) For each transition $\Phi$-compute:

$$Q[\vec{i}] \xrightarrow{\vec{i} \in D} Q'[\vec{i}]$$

   Emit the clause:

$$\boxed{\vec{i} \in D : Q(\vec{i}) = Q'(\vec{i})}$$

   (f) For each final state $Q_f[\vec{i}]$ corresponding to the parameter $x_i$ of $X$, emit the clause:

$$\boxed{\forall \vec{i} : Q_f(\vec{i}) = x_i}$$

---

FIG. 8.9 – *MSA_To_SARE*

Applying the algorithm to the sub-automata given in figure 8.8, we finally obtain the following SARE:

$$
\begin{array}{lll}
\forall \vec{i_T}, \vec{i_P} & : X(x,y)(\vec{i_T}, \vec{i_P}) & = \boxed{\langle T_2, \vec{i_T}\rangle, \langle S_2, \vec{i_P}\rangle}(\vec{i_T}, \vec{i_P}) \\
\forall \vec{i_T}, \vec{i_P} & : \boxed{\langle T_2, \vec{i_T}\rangle, \langle S_2, \vec{i_P}\rangle}(\vec{i_T}, \vec{i_P}) & = +\left(\boxed{\langle T_2, \vec{i_T}\rangle, \Phi_+}(\vec{i_T}, \vec{i_P}), \boxed{\langle T_2, \vec{i_T}\rangle, \Phi_{P_2}}(\vec{i_T}, \vec{i_P})\right) \\
\forall \vec{i_T}, \vec{i_P} & : \boxed{\langle T_2, \vec{i_T}\rangle, \Phi_+}(\vec{i_T}, \vec{i_P}) & = \boxed{\Phi_X, \Phi_+} \\
\forall \vec{i_T}, \vec{i_P} & : \boxed{\Phi_X, \Phi_+} & = x \\
\forall \vec{i_T}, \vec{i_P} & : \boxed{\langle T_2, \vec{i_T}\rangle, \Phi_{P_2}}(\vec{i_T}, \vec{i_P}) & = \boxed{\langle T_2, \vec{i_T}\rangle, \langle P_2, [\vec{i_P}, 5]\rangle}(\vec{i_T}, \vec{i_P}) \\
\forall \vec{i_T}, \vec{i_P} & : \boxed{\langle T_2, \vec{i_T}\rangle, \langle P_2, [\vec{i_P}, 5]\rangle}(\vec{i_T}, \vec{i_P}) & = \boxed{\langle T_2, \vec{i_T}\rangle, \langle P_2, [\vec{i_P}, \vec{j_P}]\rangle}(\vec{i_T}, \vec{i_P}, 5) \\
\forall \vec{i_T}, \vec{i_P}, \vec{j_P} & : \boxed{\langle T_2, \vec{i_T}\rangle, \langle P_2, [\vec{i_P}, \vec{j_P}]\rangle}(\vec{i_T}, \vec{i_P}, \vec{j_P}) & = \times\left(\boxed{\langle T_2, \vec{i_T}\rangle, \Phi_\times}(\vec{i_T}, \vec{i_P}, \vec{j_P}), \boxed{\langle T_2, \vec{i_T}\rangle, \Phi_V}(\vec{i_T}, \vec{i_P}, \vec{j_P})\right) \\
\vec{j_P} = 0 & : \boxed{\langle T_2, \vec{i_T}\rangle, \Phi_\times}(\vec{i_T}, \vec{i_P}, \vec{j_P}) & = 1 \\
1 \leq \vec{j_P} \leq 5 & : \boxed{\langle T_2, \vec{i_T}\rangle, \Phi_\times}(\vec{i_T}, \vec{i_P}, \vec{j_P}) & = \boxed{\langle T_2, \vec{i_T}\rangle, \langle P_2, [\vec{i_P}, \vec{j_P} - 1]\rangle}(\vec{i_T}, \vec{i_P}, \vec{j_P}) \\
\forall \vec{i_T}, \vec{i_P}, \vec{j_P} & : \boxed{\langle T_2, \vec{i_T}\rangle, \langle P_2, [\vec{i_P}, \vec{j_P} - 1]\rangle}(\vec{i_T}, \vec{i_P}, \vec{j_P}) & = \boxed{\langle T_2, \vec{i_T}\rangle, \langle P_2, [\vec{i_P}, \vec{j_P}]\rangle}(\vec{i_T}, \vec{i_P}, \vec{j_P} - 1) \\
\forall \vec{i_T}, \vec{i_P}, \vec{j_P} & : \boxed{\langle T_2, \vec{i_T}\rangle, \Phi_V}(\vec{i_T}, \vec{i_P}, \vec{j_P}) & = \boxed{\Phi_V, \Phi_V}(\vec{i_T}, \vec{i_P}, \vec{j_P}) \\
\forall \vec{i_T}, \vec{i_P}, \vec{j_P} & : \boxed{\Phi_V, \Phi_V}(\vec{i_T}, \vec{i_P}, \vec{j_P}) & = y
\end{array}
$$

Remark that the surrounding subscripts are useless since we assume $X$ has the same definition at each iteration. Thus we can remove them. In a more human-readable way, this SARE can be written:

$$
\begin{array}{lll}
 & X(x,y) & = +(x, P(5)) \\
j = 0 & : P(j) & = 1 \\
1 \leq j \leq 5 & : P(j) & = \times(P(j-1), y)
\end{array}
$$

which corresponds to the expected unifier of the motivating example.

Once a unifier is built from each sub-automaton $\mathcal{A}_i$, we have to check whether two sub-automata computing the same template variable provides the same value. This can be done by applying the SAReQ algorithm [13]. If the SAREs computing the same template variable are equivalent, they are emitted as a unifier of the matching problem $T \stackrel{?}{=} P$. It remains to inspect the other OR-branchings to find the other possible unifiers.

### 8.5.5 Input Mapping Extraction

Once the unifiers are found, it remains to provide the relevant values of template inputs. In the same manner as the instantiation test described in the previous chapter, the cartesian product automaton unrolls simultaneously the template and the program until the

template inputs are reached, linking the template input to a program value, which can be either a program input $a(\vec{i_T})$, or a more complex expression $t(\vec{i_P})$ depending on a counter $\vec{i_P}$ Consequently, the input mapping can be obtained by inspecting the leaves.

Figure 8.10 provides the algorithm to recover the input mapping. A stated above, we just collect the informations provided by reachable leaves. The last item expresses a general failure of the cartesian product automaton analysis, we do not know whether the program matches the template.

---

**Algorithm** *Extract_Inputs*

---

**Input:** $\mathcal{A}_{T\times P}$, *the cartesian product automaton*
**Output:** $\delta$, *a mapping defining template inputs*

For each leaf $\boxed{a_T(\vec{i_T}), S(\vec{i_P})}$:

- Compute the reaching set $R$ of $\boxed{a_T(\vec{i_T}), S(\vec{i_P})}$
- For each $(\vec{i_T}, \vec{i_P}) \in R$,
  emit the mapping $\boxed{a_T(\vec{i_T}) \overset{\delta}{\mapsto} S(\vec{i_P})}$
- In case of functional incoherence (two different definitions for a given $a_T(\vec{i_T})$), then emit $\boxed{?}$

---

FIG. 8.10 – *Extract_Inputs*

In practice, reaching sets are expressed by a finite set of clauses, despite their parametric size. This guarantee that the mapping emitted has a finite representation.

### 8.5.6 Slice Extraction

In the same manner as the instantiation test described in the previous chapter, the cartesian product allows to unroll simultaneously the template and the program, describing exactly the program operations involved in the slice. The algorithm described in figure 8.11 follows the same idea than the algorithm described in the previous chapter. By construction, the MSA exactly describes the program slice. We have thus to collect the program statements reachable from the initial state, and to compute the corresponding iteration vectors by using the reaching sets.

## 8.6 Complexity Issues

Consider the matching problem between a template with $T$ statements and $X$ free variables, and a program slice with $P$ statements. In the worst case, each free variable is unrolled on the whole program, providing $X$ sub-automata $\mathcal{A}_1 \ldots \mathcal{A}_X$ with $P$ states. Since an OR-branching tries to unroll a free variable, and to project on each argument, it has $a + 1$ branches, where $a$ is the arity of the free variable. Denoting by $a_1 \ldots a_X$

**Algorithm** *Extract_Slice*

**Input:** $\mathcal{A}$, *the unification automaton*
**Output:** $\mathcal{S}$, *the corresponding program slice*

For each state $\boxed{\langle T, \vec{i_T}\rangle, \langle P, \vec{i_P}\rangle}$:

- Compute its reaching set $R = \{(\vec{i_{T1}}, \vec{i_{P1}}) \ldots (\vec{i_{Tn}}, \vec{i_{Pn}})\}$
- emit $\boxed{\{(P, \vec{i_{P1}}) \ldots (P, \vec{i_{Pn}})\}}$.

FIG. 8.11 – *Extract_Slice*

the arities of the $X$ free-variables, and bounding the number of OR-branchings in $\mathcal{A}_i$ by $|\mathcal{A}_i| = P$ we finally obtain $\prod_{i=1}^{X}(a_i+1)^P$ combinations at most.

For each OR-branching combination, we have to re-compute the reaching set of each node of the automaton. Hence, we have $|\mathcal{A}_{T\times P}|\times\prod_{i=1}^{X}(a_i+1)^P$ reaching sets computations, where $|\mathcal{A}_{T\times P}|$ denotes the number of states of the MSA. In the worst case, we have $|\mathcal{A}_{T\times P}| = T \times P$. Taking the reaching set computation as the significant operation, the worst-case complexity of the instantiation test can be written:

$$\mathcal{O}\left(T \times P \times \prod_{i=1}^{X}(a_i+1)^P\right)$$

More often, $X = 1$ and $a_1 = 2$, leading to write the complexity $\mathcal{O}(T \times P \times 3^P)$. Although the complexity is exponential in the slice size, we have to keep in mind that it is an upper bound of the average complexity. Experimental results described thereafter show that 25% of the OR branchings combinations can be pruned. Combined with another optimization avoiding the redondances in reaching sets computations, the instantiation test takes a few minutes in practice for slices with less than 10 statements.

## 8.7 Experimental Results

We provide thereafter a description of several important implementation points, including particularly two techniques to reduce the amount of OR-branchings to handle, and the reaching set to compute. Then we give some experimental results on simple matching examples.

### 8.7.1 OR-branchings Generation

The key point to reduce the complexity of the algorithm is to reduce as much as possible the number of OR-branchings to handle. We present thereafter two pruning techniques, validated by experimental results.

| Template | Program | Physical pruning | | RS pruning | |
|---|---|---|---|---|---|
| | | Removed | Total | Removed | Total |
| temp_reduc | reduc1 | 0 | 6 | 0 | 6 |
| temp_reduc | reduc1_peel | 6 | 36 | 2 | 30 |
| temp_reduc | reduc2 | 6 | 36 | 5 | 30 |
| temp_reduc | reduc2_peel | 6 | 36 | 3 | 30 |
| **Removed** | | **15.7 %** | | **8.7 %** | |

TABLE 8.1 – OR-branchings pruning

Figure 8.12.(a) provides a simple example of MSA with two OR-branchings $A$ and $B$. A naive OR-branchings generation will produce:

$$\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}$$

Since the choice of $a_1$ makes $B$ unreachable, $a_1b_2$ can be discarded (*physical pruning*). Remark that the OR-branching nodes do not always form a tree. Figure 8.12.(b) gives a simple example, where none of the generated OR-branchings can be discarded.

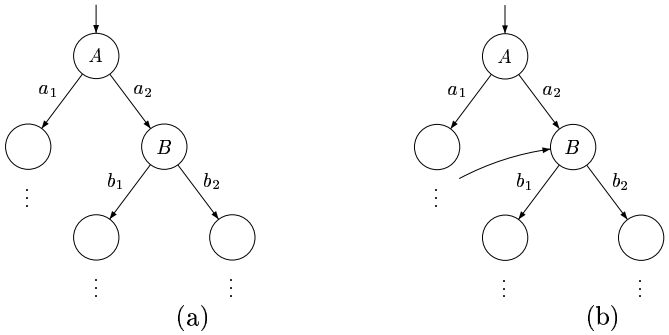



FIG. 8.12 – Two simple examples of OR-branching

Once an OR-branching is chosen, the algorithm computes the reaching sets to remove the unreachable states. Taking the previous example, if the OR-branching $a_1b_1$ makes $B$ unreachable, then all the OR-branching of the form $a_1?$ will lead to the same simplified MSA, and can be discarded (*reaching set pruning*).

Consider the two Φ-OR-branchings at the bottom of figure 8.8. To reduce the amount of OR-branchings, we consider that argument of free variables are different, and we discard the same choices in the OR-branchings coming from a decompose. Such an optimization reduces drastically the cost of the method and allows to handle most matching problems.

Table 8.1 provides experimental results for some matchings on small kernels. reduc1 is a simple reduction computing the sum of an array, while reduc2 is the more complicated reduction handled in this chapter. The _peel are control variations with a peeling on the main loop. Finally, temp_reduc is the reduction template handled in all this thesis. It appears that 24.4 % of the OR-branchings can be removed, leading to reduce significantly the whole execution time.

| Template | Program | Cache RS | | | Cache Closures | | | Execution time |
|---|---|---|---|---|---|---|---|---|
| | | Hits | Misses | Total | Hits | Misses | Total | |
| temp_reduc | reduc1 | 95 | 98 | 193 | 554 | 4 | 558 | 22 s |
| temp_reduc | reduc1_peel | 1098 | 255 | 1353 | 2173 | 4 | 2177 | 2 mn 5 s |
| temp_reduc | reduc2 | 819 | 267 | 1086 | 3189 | 7 | 3196 | 3 mn 38 s |
| temp_reduc | reduc2_peel | 901 | 273 | 1174 | 3251 | 9 | 3260 | 5 mn 2 s |
| **Reuse** | | **76.5 %** | | | **99.7 %** | | | |

TABLE 8.2 – Cache behaviour

### 8.7.2 Presburger Relations Handling

The significant operation of our algorithm is the computation of Presburger relations, and particularly the transitive closures. In order to avoid to re-compute the same regular expressions we maintain a *base* containing the computed Presburger relations with their regular expression (*reaching set cache*). If the regular expression is not in the base, it is computed by traversing the tree in post-fix order. In the same manner, transitive closures are handled with another base (*closures cache*). These two bases are implemented with an AVL indexed by the strings of the regular expressions.

Table 8.2 provides experimental results on the same kernels. The execution times were measured on a Pentium 4, 1.6 GHz with 256 MB RAM. The reuse rate of the cache RS is pretty good. Indeed, from one OR-branching to another, there is often a few change, which affect only a small part of the MSA. As a consequence, most of the reaching sets have already been computed. In another hand, the MSAs have a few cycles (see column *Cache Closure, Misses*), which are involved in most regular expressions. This explains the reuse rate of the cache closures.

Combining the OR-branching pruning with the caches allows to reduce significantly the execution time (about 75 %). Even if the complexity remains exponential, the factor is enough reduced to have a reasonnable execution time on average size patterns and program slices (less than 10 assignements).

## 8.8 Discussion

We have proposed a new instantiation test able to cope with more matching problems than the instantiation test presented in the previous chapter. Particularly, we are able to handle most matching problems with a recursive unifier. In the same manner as the equivalence test and the previous instantiation test, we are able to cope with all possible program variations, except semantics variations.

**Organization & Data structure variations** The cartesian product traverses exactly the program operations concerned with the template, without taking statements interleaved within the program into account. Hence, organization variations are handled. Moreover, our instantiation test compares the operations without taking account of variables, and particularly their type Consequently, data structure variations are also handled.

**Control variations** The template and the program are traversed through the dependences without taking account of control structures. For instance, we do not make difference between $n$ consecutive operations `s = s + a(i)` and the same statement nested in a loop. Since control variations do not hurt the dependences between operations, they are also handled.

However, the definitions obtained are by construction always a subset of the program's SARE clauses, which make impossible to detect unifiers whose definition needs to cut the domain of a clause. For instance, consider the following matching problem:

$$\begin{cases} & O & = S_2(n) \\ i = 0 & : S_2(i) & = X(S_1) \\ 1 \leq i \leq \boxed{n} & : S_2(i) & = X(S_2(i-1)) \\ & S_1 & = 0 \end{cases} \qquad \begin{cases} & O & = S_2(n \times p) \\ i = 0 & : S_2(i) & = S_1 + 1 \\ 1 \leq i \leq \boxed{n \times p} & : S_2(i) & = S_2(i-1) + 1 \\ & S_1 & = 0 \end{cases}$$

The program obviously matches the template with $X$ defined as:

$$\begin{cases} & X(x) & = S(p) \\ i = 0 & : S(i) & = x + 1 \\ 1 \leq i \leq \boxed{p} & : S(i) & = S(i-1) + 1 \\ & S_1 & = 0 \end{cases}$$

But our method leads to a failure. Indeed, this definition of $X$ is not a sub-SARE of the program SARE, and would requires to split the iteration domains, that is nnot allowed by our method.

Despite an exponential complexity, the method can be applied to slices with a reasonnable size ($\approx$ 10 assignments) thanks to several optimization techniques allowing to reduce the amount of OR-branchings, reaching sets and transitive closures to handle. Experimental results on simple matching problems have shown that it decreases significatively the cost of our instantiation test to a few minutes.

# Chapter 9

# Substitution

As suggested by the title of this dissertation, the last step of our method is to substitute the occurences found in the program by the corresponding call to the optimized library, whenever its *possible* and *interesting*. The first difficulty to deal with is to decide whether the substitution is possible, in the meaning that it will preserve the program semantics. There are indeed cases where cyclic flow dependences forbid the removal of the slice, and finally its substitution by a call to the library. The next important issue to tackle is to select the best, or at least, a good substitution set, that lead to a global performance improvement. As we will see, this step raises important issues related to static performance evaluation, a very difficult problem that we try to tackle here. Finally, it remains to generate the code of the program with the calls to the library, in the most efficient way.

Whereas Metzger [76] uses a simple approach assigning a mark, the *saving*, to each function library, Kessler [64] uses a performance prediction approach based on benchmarking of library functions on the target architecture. More sophisticated performance prediction approaches have been proposed, and provide a set of parameters which aims to characterise the performances of the source program. Padua [21] proposes an analytical approach based on the stack distance model to evaluate the number of cache misses, Ghosh *et al.* [51] have introduced the *cache miss equations* as a mathematical framework that precisely represents cache misses in a loop nest. Fahringer [41] proposes a profile-based approach which benchmarks small kernels, and try to recognize them a the program, in order to deduce the whole performance paremeters.

Following Metzger, the approach investigated in this chapter is to associate to each library function a number quantifying its capacity to increase program performances. The optimal set of slices to substitute is then selected by solving a 0-1 program whose constraints express the overlapping between slices. We also present an algorithm to generate a straightforward code with substitution.

This chapter is organized as follows: Section 9.2 presents a method to decide whether a slice can be separated from the program, and thus substituted. Section 9.3.2 presents a simple algorithm to select a set of slices whose substitution will leads to a performance improvement. Finally, Sections 9.4 and 9.5 present the algorithmic needed to instantiate the template, and to generate the code with the substitution. Section 9.6 provides related works on performance prediction, and Section 9.7 concludes, pointing out the limitations of our method, and the possible improvements.

## 9.1 Overview of the Method

Figure 9.1 sums up the main steps of the substitution. As mentioned in introduction, the first step is to select the slices that can be substituted by a call to the library without destroying the program semantics. This step is not so trivial, and requires the accurate formalization and algorithms described in the next section. We then select the slices whose substitution can lead to an improvement of program performances. Finally, we perform the substitutions by generating the code with the relevant calls. The following sections describe each of these steps.

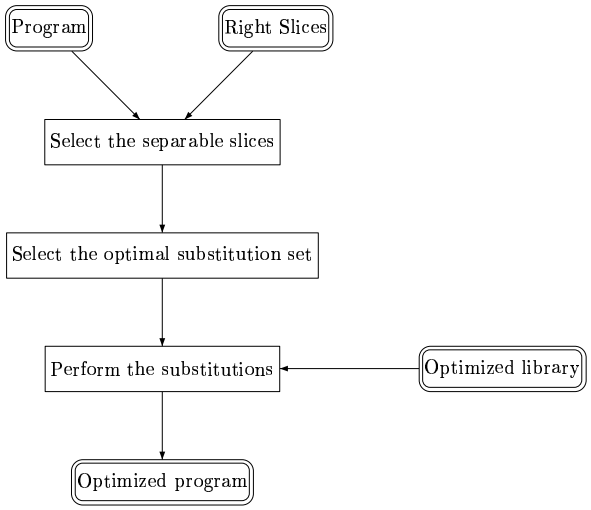



FIG. 9.1 – Main steps of the substitution

## 9.2 Select the Separable Slices

Most often, the slices are interleaved with the rest of the program. Before performing substitution, we have first to separate them from the program. In this section, we present an algorithm to check whether a given slice is separable from the program. We first introduce the notion of complementary slice, then we present our separability test.

### 9.2.1 Complementary Slice

Consider a recognized algorithm $A = \{\langle A_1, I_1\rangle \dots \langle A_n, I_n\rangle\}$, with first operation $\langle A_1, \vec{i}_1\rangle$ and last operation $\langle A_n, \vec{i}_n\rangle$. Its *complementary slice* is the set of program operations executed between the first and the last operations of $A$:

$$\overline{A} = \{\langle S, \vec{i}\rangle \mid \langle A_1, \vec{i}_1\rangle \prec \langle S, \vec{i}\rangle \prec \langle A_n, \vec{i}_n\rangle \wedge \nexists i \; s.t. \; S = A_i\}$$

Consider the example given in figure 9.2, where the recognized algorithm is constituted of operations:

$$A = \{\langle A_1, \{9, 10\}\rangle, \langle A_2, \{\}\rangle, \langle A_3, \{1, 2, 3, 4\}\rangle\}$$

```
P1  s = 0
    do i = 1, 10
P2  | a(i) = a(i-1) + 1
    | if i >= 9 then
A1  | | dot = dot + 2*a(i)
    | endif
    enddo
A2  dot = dot + b*c
P3  s = s + 1
    do i = 1, 4
P4  | s = s + b(i)
A3  | dot = dot + a(i)*b(i)
    enddo
```

FIG. 9.2 – Running example

Its complementary slice is thus:

$$\overline{A} = \{\langle P_2, \{10\}\rangle, \langle P_4, \{1,2,3,4\}\rangle\}$$

$\overline{A}$ can be computed from $A$ using the algorithm described in figure 9.3. The first operation of $A$ is computed by searching the lexicographic minimum of $I_1$. The last operation of $A$ is computed in the same way (step 1). Following the definition of $\overline{A}$, we compute for each statement $P_i$ the set of the corresponding operations between the first and the last operations of $A$. The resulting set is an union of $\mathbb{Z}$-polytopes whose emptiness can be checked by using an appropriate solver [42] (step 2.(b)).

**Algorithm** *Complementary*

**Input:** *The recognized algorithm* $A = \{\langle A_1, I_1\rangle \dots \langle A_a, I_a\rangle\}$
*The program* $P = \{\langle P_1, \mathcal{I}_1\rangle \dots \langle P_p, \mathcal{I}_p\rangle\}$
**Output:** $\overline{A}$, *the complementary of* $A$ *in* $P$

1. Let $\langle A_1, \vec{i}_1\rangle$ be the first operation of $A$, and $\langle A_n, \vec{i}_n\rangle$ be its last operation.
2. For each statement $\langle P_i, \mathcal{I}_i\rangle \in P$, $P_i \notin \{A_1 \dots A_a\}$:
   (a) Form the iteration domain: $I = \{\vec{i} | \vec{i} \in \mathcal{I}_i \wedge \langle A_1, \vec{i}_1\rangle \prec \langle P_i, \vec{i}\rangle \prec \langle A_n, \vec{i}_n\rangle\}$
   (b) If $I \neq \emptyset$, emit $\langle P_i, I\rangle$.

FIG. 9.3 – *Complementary*

### 9.2.2 Separability Test

Once $\overline{A}$ is computed, it remains to check whether it is separable from $A$ in order to replace $A$ by a call to an optimized library. $A$ is said to be *separable* if all flow dependences start *exclusively* from $A$ to $\overline{A}$, or *exclusively* from $\overline{A}$ to $A$.

If the flow dependences start exclusively from $A$ to $\overline{A}$, $A$ is said *top-separable* from $\overline{A}$, and can be substituted by a call to $A$, *before* $\overline{A}$.

Here is an example of top-separable slice (left) and its substitution by a call (right):

```
   do i = 1,n                          call daxpy(a,x,y)
   | if i >= 2 then                    do i = 1,n
A  | | s(i) = y(i-1) + 1               | if i >= 2 then
   | endif                        A    | | s(i) = s(i) + 1
A  | y(i) = y(i) + a*x(i)              | endif
A  | s(i) = 2*y(i)                A    | s(i) = 2*y(i)
   enddo                               enddo
```

In the case where flow dependences starts exclusively from $\overline{A}$ to $A$, $A$ is said *down-separable* from $\overline{A}$, and can be substituted by a call to $A$, *after* $\overline{A}$. We provide thereafter an example of down-separable slice (left) and its substitution by a call (right):

```
   do i = 1,n                          do i = 1,n
   | if x(i) >= 0 then                 | if x(i) >= 0 then
A  | | x(i) = s(i) + 1               A  | | x(i) = s(i) + 1
   | endif                            | endif
A  | y(i) = y(i) + a*x(i)         A    | x(i+1) = 2*x(i)
A  | x(i+1) = 2*x(i)                  enddo
   enddo                               call daxpy(a,x,y)
```

Otherwise, we consider that $A$ and $\overline{A}$ are *not separable*, and we do not perform the substitution. Here is an example of unseparable slice:

```
   do i = 1,n
A  | y(i) = y(i) + a*x(i)
A  | x(i+1) = 2*y(i)
   enddo
```

On the motivating example, the slice is down-separable because of the flow dependence from $\langle P_2, 10\rangle$ to $\langle A_1, 10\rangle$.

Whenever it is possible, we compute the flow dependences by using exact dataflow analysis. Otherwise, we use an approximate reaching definition analysis. In this case, the imprecise flow dependences can lead to reject a separable algorithm. In addition, we have to check whether the side effects of the recognized algorithm are limited to its (official) output. If an intermediate variable is readen out of the slice, we consider the algorithm as inseparable. This can also be checked by using the flow dependences. In the same manner, we compute the exact data-flow whenever it is possible. Otherwise, we use the approximation provided by the reaching definitions analysis.

Remark that our separability test is a conservative approximation since it does not enable inseparable slices. This property is stated in the following proposition, where our separability test is denoted by *Is_Separable*.

**Proposition 9.1 (Correction).** *Consider a program $P$ and a slice $S \subset P$. Then:*

$$\textit{Is\_Separable}(S) \Longrightarrow S \textit{ is separable from } P$$

*Proof.* Suppose *Is_Separable*$(S)$, and denote by $\{S_1 \dots S_n\}$ the smallest partition of $S$ whose elements are contiguous operations of $S$, and follow the execution order. More formally:

- For each operations $\omega_1$ and $\omega_2 \in S_i$ with $\omega_1 \prec \omega \prec \omega_2$, we have $\omega \in S_i$.
- Given $i < j$, for each operations $\omega \in S_i$ and $\omega' \in S_j$, we have $\omega \prec \omega'$.

In the same manner, consider the same partition for the complementary slice $\overline{S}$: $\{\overline{S_1} \ldots \overline{S_p}\}$. By construction the $S_i$ alternates with the $\overline{S_i}$ in the execution order. Indeed, if $S_i$ and $S_j$ alternates, one can fusion them to make a smaller partition, which contradicts the hypothesis. Consequently, the whole sequence of operations can be written:

$$S_1;\ \overline{S_1};\ \ldots;\ S_{n-1};\ \overline{S_{n-1}};\ S_n$$

When the slice is down separable, no dependence exists from $S_i$ to a $\overline{S_{i+k}}$. Obviously no dependences exists from $\overline{S_{i+k}}$ to $S_i$, since the operations of $\overline{S_{i+k}}$ are executed after those of $S_i$. Consequenty, the portions $S_i$ and $\overline{S_{i+k}}$ are independents and we can move the whole slice after its complementary by applying legal commutations. In a symetric manner we can show the correction of the top separability. □

## 9.3 Select the Optimal Substitution Set

Once the separable slices are selected, it remains to provide the slices whose substitution leads to a performance improvement. We present here a preliminary approach to resolve the overlappings between slices, choosing a substitution set leading to a reasonnable performance improvement. We use a simple technique based on a system of marks, and we do *not* involve an accurate performance prediction technique, which is not in the scope of this thesis.

### 9.3.1 Defining the Performance Gain

We associate to each library function a real number called *gain*, which quantify the impact of substitution on the program performances. This section does not aim to present an automatic method to compute the performance gains. We assume they are provided by the library developper, according to the observed speed-ups.

Although gains allow to make a choice between two conflicting slices, it is not clear that the replacement will really improve program performances. A better approach, discussed in conclusion but not investigated in this thesis would be to compare the performances of the source program and the replaced program, for each possible replacement. Such a comparison would be achieved by using benchmarking, or analytical performance prediction techniques, as described in related work [21, 51, 41].

### 9.3.2 Selecting the Optimal Substitution Set

Once the gain of each slice is known, it remains to select the substitution set with the maximum gain. Since two slices can overlap, we cannot select all the slices. In this section, we present the method proposed by Metzger [76] to get an optimal substitution set, given a gain function over slices.

Given a set of separable slices $\mathcal{S} = \{S_1 \dots S_n\}$, a *valid substitution set* is a part $R \subset \mathcal{S}$ whose slices do not share operations (overlap). We define the *gain* of $R$ as the sum of its slices gain: $g(R) = \sum_{s \in R} g(s)$. With these definitions, an *optimal substitution set* is a valid substitution set with the maximum gain: $\max_{R \text{ is valid}} g(R)$.

Following Metzger [76], an optimal substitution set can be found by solving the following integer program with 0-1 variables:

$$\begin{array}{ll} \underline{\max} & s_1 g(s_1) + \ldots + s_n g(s_n) \\ \underline{\text{s.c.}} & s_i + s_j \leq 1 \text{ for each overlapping slices } S_i \text{ and } S_j \end{array}$$

Where $s_i$ is a 0-1 variable taking the value 1 when the slice $S_i$ is selected, and 0 otherwise. The integer program just maximizes the gain of the substitution set under the constraint that the slices must not overlap.

## 9.4 Instantiate the Template

Before applying the substitution, an important step is to set the template variables according to the unifier provided by the instantiation test. The inputs and the output have also to be set according to program variables. Consider indeed the example given in figure 9.4.(a) where a BLAS daxpy $y \leftarrow ax + y$ has been recognized. Before applying the daxpy, one must setting up its inputs according to program variables (see (b), part $I$). In a symetric manner, one must also put the results within program variables (part $O$).

```
do i = 1,n
| do j = 1,p
| | y(i,j) = 2*x(i) + y(i,j)
| enddo
enddo
```

(a)

```
  do i = 1,n
I | y_input(1:p) = y(i,1:p)
  | call daxpy(2 ... x ... y_input)
O | y(i,1:p) = y_input(1:p)
  enddo
```

(b)

FIG. 9.4 – Input and output code

We provide thereafter a method to generate the code for free-variables, followed by precise description of the input and output code generation.

### 9.4.1 Template Variables

Since the template variables are expressed are SAREs, a first task is to compute the corresponding imperative program, and to put it in a function. Each SARE array $S$ represents a statement, whose iteration domain can be computed by union of the domains $D$ involved in its defining clauses:

$$\bigcup_{\vec{i} \in D:\ S[\vec{i}]=t} D$$

And whose scheduling function $\theta(S, \vec{i})$ is exactly the SARE schedule. Union of $\mathbb{Z}$-polyhedra can be achieve by using the Polylib [69], and affine schedules can be computed with the

algorithm described in [30]. It remains to give the schedule and the iteration domain of each statement to an efficient code generator [14], to obtain the relevant definition of the template variable.

### 9.4.2 Input and Output Mapping

Another important task is to generate the code to express the template inputs and output with program variables. The instantiation test provides a mapping betwen a template input $I$, and program expressions as follows:

$$\begin{array}{l} \vec{i} \in D_1 : \quad I(\vec{i}) = t_1(\vec{i}) \\ \ldots \\ \vec{i} \in D_n : \quad I(\vec{i}) = t_n(\vec{i}) \end{array}$$

Where the $D_k$ are $\mathbb{Z}$-polyhedra and the $t_k(\vec{i})$ are terms built from program basic operators ($+$, $\times$, etc), and program SARE's arrays $S[u(\vec{i})]$. A first task is to substitute the $S[u(\vec{i})]$ by program variables in order to obtain a program-related definition of $I$. Recall that the SAREs are produced by the algorithm described in chapter 6, which associates exactly one array $S[\vec{i}]$ to a program statement with iteration vector $\vec{i}$:

$$S : \quad \texttt{s} = ...$$

and ensures that $S[u(\vec{i})]$ contains the value computed by the operation $\langle S, u(\vec{i}) \rangle$ *i.e.* the value of `s` at the iteration $u(\vec{i})$. In order to recover this value, a simple solution is to insert a statement keeping the values taken by `s` on the different iterations:

$$\begin{array}{ll} S : & \texttt{s} = ... \\ & \texttt{s\_expanded}[\vec{i}] = \texttt{s} \end{array}$$

It remains to substitute $S[u(\vec{i})]$ by $\texttt{s\_expanded}[u(\vec{i})]$ in $t_k(\vec{i})$ to obtain a program-related definition of the template input $I$. Finally, for each definition $\vec{i} \in D_k : \quad I(\vec{i}) = t_k(\vec{i})$, it remains to generates the code scanning the polyhedron $D_k$ by using an efficient code generator [14], and putting $I(\vec{i}) = t_k(\vec{i})$ as the executed statement.

Since we detect slices with one output statement, output mapping generation is straighforward. Recall that the instantiation test constructs a mapping from template output subscripts to slice output subscripts. On the above example, we have the mapping:

$$\delta : \left| \begin{array}{lcl} D_S & \rightarrow & D_T \\ (i,j) & \mapsto & j \\ y(i,j) & \mapsto & y\_input(j) \end{array} \right.$$

It remains to generate the code to scan the iteration domain of the template $D_T$, putting the assignment $y(\delta^{-1}(i)) = y\_input(i)$ in the loop nest. Such a code can be generated by using an heuristic [14] minimizing the control. Experimental results will show that in most cases, input and output mappings are straightforward, and can be achieved with a fortran intrinsic `a( ... binf:bsup:step ... )`.

## 9.5 Perform the Substitutions

Once the optimal substitution set is found, it remains to perform the substitution. In this section, we present an algorithm able to remove a slice from the program, and replace it by a call to the optimized library function.

Basically, the substitution deletes the operations of $A$, and puts the relevant call before, or after $\overline{A}$. It can be achieved by using the algorithm given in figure 9.5. If $A$ is top-separable, the call is inserted before the first operation of $\overline{A}$ (step 1). All relevant operations of $A$ are discarded by using a condition. If $A_i$ is not in a loop, it is removed from the text (step 2). If $A$ is down-separable, the call is inserted after the last operation of $\overline{A}$ (step 3). Note that the call will also contains the input and output mappings whose generation is detailed in the previous section.

---

**Algorithm** *Substitute*

---

**Input:** $A = \{(A_1, I_1) \dots (A_a, I_a)\}$, *the recognized algorithm*
$(F, \vec{i}_F)$ *the first operation of* $\overline{A}$
$(L, \vec{i}_L)$ *the last operation of* $\overline{A}$
**Output:** $P$, *the modified program*

1. if $A$ is top-separable, insert:
   **if** $\vec{i} = i_F$ **then** call *Optimized_A*
   Before $F$.
2. Replace each statement $A_i$ by:
   **if** $\vec{i} \notin I_i$ **then** $A_i$
3. if $A$ is down-separable, insert:
   **if** $\vec{i} = i_L$ **then** call *Optimized_A*
   After $L$.

---

FIG. 9.5 – *Substitute*

The application of *Substitute* to the motivating example provides the program given in figure 9.6. The generated code could be improved by deleting the conditional nestings corresponding to an empty iteration domain. A more efficient code can be generated by using a recent code generation technique [14].

## 9.6 Related Work

The key point of this chapter is the choice of the relevant substitution which should leads to the best performance improvement. In this section, we present several approaches which addresse the problem of *performance prediction*.

Padua *et al.* [21] propose an approach to compute a symbolic expression estimating the the computation time $T_{CPU}$ spent by the processor itself, and the time $T_{MEM}$ spent while accessing the memory hierarchy. Their approach aims to handle general programs with complex control flow such as loops with early exits and while loops. The sub-model

```
P1    s = 0
      do i = 1, 10
P2    | a(i) = a(i-1) + 1
      | if i >= 9 then
      | | if i ∉ {9,10} then
A1    | | | dot = dot + 2*a(i)
      | | endif
      | endif
      enddo
A2    removed
P3    s = s + 1
      do i = 1, 4
P4    | s = s + b(i)
      | if i ∉ {1...4} then
A3    | | dot = dot + a(i)*b(i)
      | endif
 call | if i = 4 then call ...
      enddo
```

FIG. 9.6 – The substituted program

for $T_{CPU}$ estimates the time spent by the processor doing computation by counting the number of operations executing in the CPU's functional units. Given an instruction, they compute a symbolic expression providing an upper bound of its number of instances. The global expression is obtained by grouping together the different instuctions expressions, weighted by the corresponding execution cycles on the target architecture. $T_{MEM}$ estimates the time spent while acessing the memory, by taking into account cache misses. An upper bound on cache misses is estimated by using the *stack distances model*. Their stack is basically an historic of memory accesses built from a program trace, which allows to evaluate the number of distinct references between two accesses to the same memory cell. They use this information to compute a symbolic expression providing an upper bound of the number of cache misses. Their approach has been validated on SpecFP kernels with target processors MIPS R10000 and UltraSparc IIi. Their experiments reveal an error of 20% in average, which is enough for our purpose.

Ghosh *et al.* [51] propose an analytical approach to evaluate the number of cache misses in static control programs, without simulation. For each reference, they built a system of linear diophantine equations, called *cache miss equations* whose solutions are exactly its miss instances. They claim that mathematical techniques can be used to compute the number of solutions, providing the exact number of cache misses.

Fahringer [41] proposes a profile-based approach to estimate the performances of parallel programs running on distributed memory parallel architectures. His tool, $P^3T$, is based on the underlying compilation and programming model of the Vienna Fortran Compiler System, which is a High-performance Fortran style compiler using the Single-Program-Multiple-Data (SPMD) programming model. The computation time estimation is based on pre-measuring the performances of a large set of kernels from primitive operations *e.g.* basic arithmetic operations to entire code patterns *e.g.* matrix multiply. The kernels performances are pre-measured on different target architectures, on a significative set of input instances. The computation time of the program is then estimated by detecting

occurences of these kernels, and multiplying their pre-measured runtime by the statement execution count, obtained by profiling. The results are finally accumulated together to obtain the overall computation time. The quality of the prediction depends on the completeness of the kernel library. Moreover the technique assumes that the kernel performance remains the same when incorporating within a program, which is not sure. For instance, the cache state just before the execution of the kernel may affects strongly its performances. This last point is the major drawback of the gain system described in this chapter.

## 9.7 Discussion

In this chapter, we have presented a method to select a subset of slices whose substitution is possible, and should improve the performances of the source program. We have also proposed an algorithm to generate the code with the substitutions.

The selection method depends on a total order between library functions, allowing to make a selection whenever two slices overlaps. However, our approach does not take account of the effective performance improvement of the program. We assume that a substitution will always improve performance, and that the performance improvement just depends on the algorithmics used in the library function, independenly of the program. Such an hypothesis is too strong. Indeed, a library function behaviour may strongly depends on the cache state before the call.

In a future work, we would like to investigate more accurate performance prediction models, such as Fahringer's [40], Ghosh's [51] or Padua's approches [21]. Such prediction models associates to a function $f$ a vector of parameters $\overrightarrow{\Pi(f)}$ which quantifies its performances. For instance, it would contain a symbolic expression giving the execution time and the number of cache misses. Given a slice $\mathcal{S}$ implementing a library function $f$, a more accurate gain would be $\overrightarrow{\Pi(\mathcal{S})}/\overrightarrow{\Pi(f)}$, where / denotes the division dimension by dimension. Following the algorithm described in section 9.3.2, this would leads to solve a 0-1 program with parameters. Such a program could be solved by using the PIP tool [42], which would provides a selection tree expressing the better set of substitution with respect to the parameters values. The generated code would follows the selection tree, providing the corresponding substituted program for each leaf. The main drawback of analytical-based performance prediction methods is that they works with an idealized model of the target machine, which can leads to predictions that significantly differ from the run times.

Another interesting approach would be to profile all possible substitutions over a representative subset of input instances, and to choose the best [83]. Such an approach has the advantages and the drawbacks of profile-based performance predictions approaches. It will exactly take account of target architecture, however we are not sure that the selected input instances are really representative. This last approach is addressed in the experimental results, presented in the next chapter.

# Chapter 10

# Experimental Results

This chapter completes the theoretical study described in the previous chapters by providing experimental results on real-life examples. Our method is applied to the detection of the BLAS functions (Basic Linear Algebra Subroutines) within the kernels of the SpecFP 2000 and the Perfect Club benchmarks. Experimental results report a substantial acceleration factor for the kernels swim, mgrid and mdg, that confirms the efficiency and the scalability of our method. Important details of our implementation, `TeMa` (Template Matcher), are also presented and discussed.

Section 10.1 presents our tool, `TeMa`, whereas Section 10.2 provides and discusses the experimental results obtained on the SpecFP and the Perfect Club benchmarks. We finally conclude in Section 10.3.

## 10.1 Implementation Issues

The optimization framework presented in this thesis has been implemented in a tool called (Template Matcher). `TeMa` is constituted of several seperate command line batch tools, which works together. `TeMa` has been implemented in Objective Caml, and represents more than 17000 lines of code. The different tools involved in `TeMa` and their relationships are depicted in figure 10.1.

The *normalization* module allows to rewrite a program to have one operator by statement. We have implemented our own Fortan 90 front-end. Most Fortran 90 programs are correctly handled, but some syntactic constructions are not yet accepted, and need to be modified by hand. Our front-end has handled with success seven kernels of the SpecFP 2000 [54] and Perfect Club benchmarks [39]. The experimental results show that the normalization increases significantly the program size (see table 10.1). The normalization of programs and patterns is always very fast, and will be used in the following treatments. Before applying the slicing method, one may need to compute the approximate data-flow information. This step is performed by the *SSA-graph generator* module, which computes the SSA form of the program and yield the result in the form of particular kind of graph called *SSA-graph*. Such a structure is useful to generate the tree automata during the *Slicing* step. The slicing provides a set of output assignement numbers to the instantiation test *Matching*.

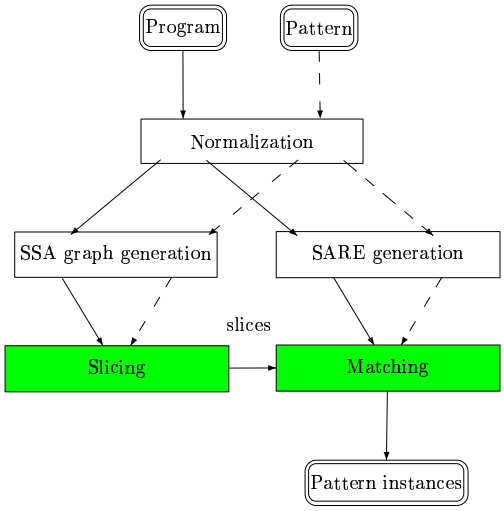



FIG. 10.1 – Overview of `TeMa`

In a symetric manner, the instantiation test needs an exact data-flow information to work. This data-flow information is stored in a SARE (System of Affine Recurrence Equations), which is generated from the normalized program by the *SARE generator*. The *Matching* module implements the instantiation test with second-order matching described in chapter 7. It involves several GPL libraries including Omega [62] and Polylib [69], interfaced by SPPoC [19].

### 10.1.1 Slicing

#### Approximated reaching definitions

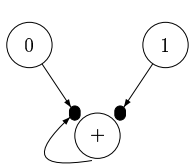
The approximate data-flow analysis build a graph representation of the program called *SSA-graph*. As stated in figure 10.2, the program is first put in SSA-form, then the SSA-graph is generated, assigning a node to each assignment, and labelling the node by the assignment operator. Remark that constants 0 and 1 are particular cases of operators, with 0 arguments. Each node has one *input port* for each argument (black points). The transitions are finally fired according to the data-flow information provided by the $\phi$-functions. Our data-flow analyzer implements the algorithm described in [1] page 683.

```
s = 0
do i = 1,n
| s = s + 1
enddo
```

(a) Original program

```
s1 = 0
do i = 1,n
| s2 = φ(s1,s2) + 1
enddo
```

(b) SSA-form




(c) SSA-graph

FIG. 10.2 – Construction of the SSA-graph

### Construction of the cartesian product

The slicing module itself inputs the SSA-graphs of the pattern and the program and executes the algorithm described in chapter 5. An OCaml library has been implemented to handle tree-automata, and particuliarly to compute the cartesian product of two tree-automata. The cartesian product is first built by applying directly the construction rules given in definition 5.2, page 73. The rules of the two tree-automata are hashed with respect to their head symbol in order to improve the construction. Unfortunately, the obtained automaton contains unreachable states. They are eliminated by applying a flooding algorithm starting from the leaves. The algorithm works on a graph representation of the cartesian product which ensures its linearity in the number of states.

## 10.1.2 Template Matching

### Construction of the unification automaton

In order to construct and manipulate simply the word automata, a generic OCaml library has been implemented. Particularly, it uses a constructor:

```
val make_from_rules :
  ('state -> ('transition * 'state) list) ->
  ('state -> bool) ->
  'state
  -> ('state,'transition) automaton = <fun>
```

The first parameter is a function implementing the rule system defining the automaton, as described in figure 7.5 page 117, or page 132. The second argument is a function deciding whether a state is final, and the last one is the initial state of the automaton. The library also includes a function of visualisation, using the `dot` tool [36].

Such a library allows to implement simply the contruction of the MSA for the two instantiation tests. It then remains to provide an implementation of the rule system by using OCaml filtering.

### Analysis issues

Presburger relations are handled using the Omega library [62] *via* the OCaml interface provided by the SPPoC library [19]. A naive and inefficient implementation would be to compute for each state its regular expression, and to give it to Omega in order to decide whether the state is reachable. In order to avoid to compute the same regular expression several times, we maintain a *base* containing the computed Presburger relations with their regular expression (*reaching set cache*). If the regular expression is not in the base, it is computed by traversing the tree in post-fix order. In the same manner, transitive closures are handled with another base (*closures cache*).

These two bases are implemented with an AVL indexed by the strings of the regular expressions, and allow to divide by two the execution time of the instantiation test. Indeed, experimental results given in section 10.2.3 show a good reuse of regular expressions: about 10 % for the reaching sets, and 96 % for the transitive closures.

## 10.2 Experimental Results

We have apply our method to detect all the functions of BLAS level 1 and 2 [67] within the kernels of the SpecFP 2000 [54] and the PerfectClub [39] benchmarks. The performance library chosen is the Intel MKL (Math Kernel Library), which provides highly optimized implementations of BLAS functions, among routines for solving problems of computational linear algebra, and some others computation intensive problems. The recognition was performed on a Pentium 4, 1.6 Ghz with 256 MB RAM, and the speed-ups have been measured on the target machine of MKL, an Itanium 2, 897 Mhz bi-processor machine with 2 GB RAM.

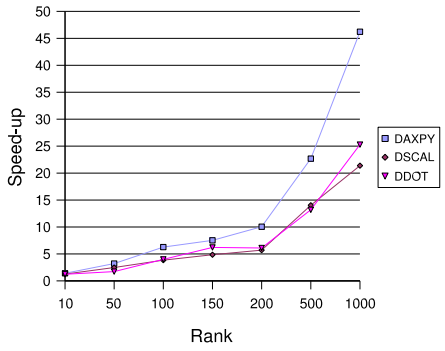



FIG. 10.3 – Speed-ups of three frequently used MKL functions

Figure 10.3 provides the speed-ups of three usual BLAS level 1 functions. Basically, daxpy computes the linear combination $\vec{y} \leftarrow a\vec{x} + \vec{y}$ given a scalar $a$ and two vectors $\vec{x}$ and $\vec{y}$, dscal computes the dilatation $\vec{x} \leftarrow a\vec{x}$ and ddot computes the dot product $(\vec{x}|\vec{y})$. These curves have been obtained by comparing the execution time of the MKL function to a straight implementation, while varying the vector dimension (rank).

We have restricted our analysis to several functions representing an important part of the execution time. Table 10.1 provides the functions analyzed in each program. Column *Normalized* gives the number of lines after normalizing the functions with one operation by assignment. The normalized forms of applu and mdg functions have important sizes since they use assignements with large formula.

Our pattern base is constituted of direct implementations of BLAS functions from the mathematical description. Several usual semantics variations were also added to increase the number of detections. For instance, for the dot product we will look for implementations starting with $dot \leftarrow 0$ or $dot \leftarrow x(0) * y(0)$, and with left or right folding: $dot \leftarrow dot + x(i) * y(i)$ or $dot \leftarrow x(i) * y(i) + dot$ ; which leads to look for four different dot products. The following sections detail the experimental results obtained for each module of `TeMa`.

| Kernel | Functions analyzed | Coverage | Lines of code | |
|---|---|---|---|---|
| | | | Original | Normalized |
| 171.swim | 5: (All) | 100 % | 351 | 414 |
| 172.mgrid | 3: `interp psinv resid` | 25 % | 120 | 243 |
| 173.applu | 3: `blts buts rhs` | 14 % | 446 | 1415 |
| 301.apsi | 14: `advt wcont smth smthf csmth horsmt horbc dctdx dctdxd dpdx dftdx ccrank dudz dvdz` | 6 % | 464 | 807 |
| qcd | 2: `project syslop` | 5 % | 115 | 355 |
| mdg | 3: `interf poteng intraf` | 52 % | 646 | 1496 |
| tis | 2: `trfa olda` | 26 % | 127 | 171 |

TABLE 10.1 – Program parts analyzed within SpecFP and Perfect Club benchmarks

### 10.2.1 Slicing

Our slicing method aims to provide all pattern occurences in the program. Since it relies on an approximative, but conservative dataflow information, several non-equivalent slices can be yielded. In order to reduce the recognition cost, we expect the slicing method to be as accurate as possible. We provide thereafter an experimental study of the accuracy, followed by an experimental verification of the theoretical complexity.

#### Accuracy

Figure 10.4 provides the amount of candidate slices detected for each kernel, and highlights the non-equivalent slices (black), and the equivalent ones (grey and white). The slicing method also yield trivial instances of patterns. For instance, $y = \alpha x + y$ could be yielded as an instance of daxpy over one-dimension vectors. Consequently, we distinguish trivial slices (grey), to non-trivial slices, whose substitution could lead to a performance improvement (white). In addition, figure 10.5 provides the corresponding rates between non-equivalent, irrelevant and interesting slices for each kernel.

It appears that 35.9% of candidates do not match, 56.4% are correct instances, but over vectors with a too small rank to obtain speed-ups, and 7.7% of candidates are correct and can be replaced by a call to BLAS. Experimental results has pointed out two causes of bad detections.

**Lack of types** Approximate reaching definitions lead to handle any variable as a scalar variable, and consequently to create wrong dependences between statements. This will leads for example to confuse a daxpy `y(i) = y(i) + a*x(i)` with a dot product `dot = dot + a(i)*b(i)`.

**Lack of $\phi$-functions semantics** Because of the approximation made by the scalar reaching definitions used within the slicing method, the control structures are not handled correctly. Consider indeed the following program (left) and its scalar SSA-form (right).

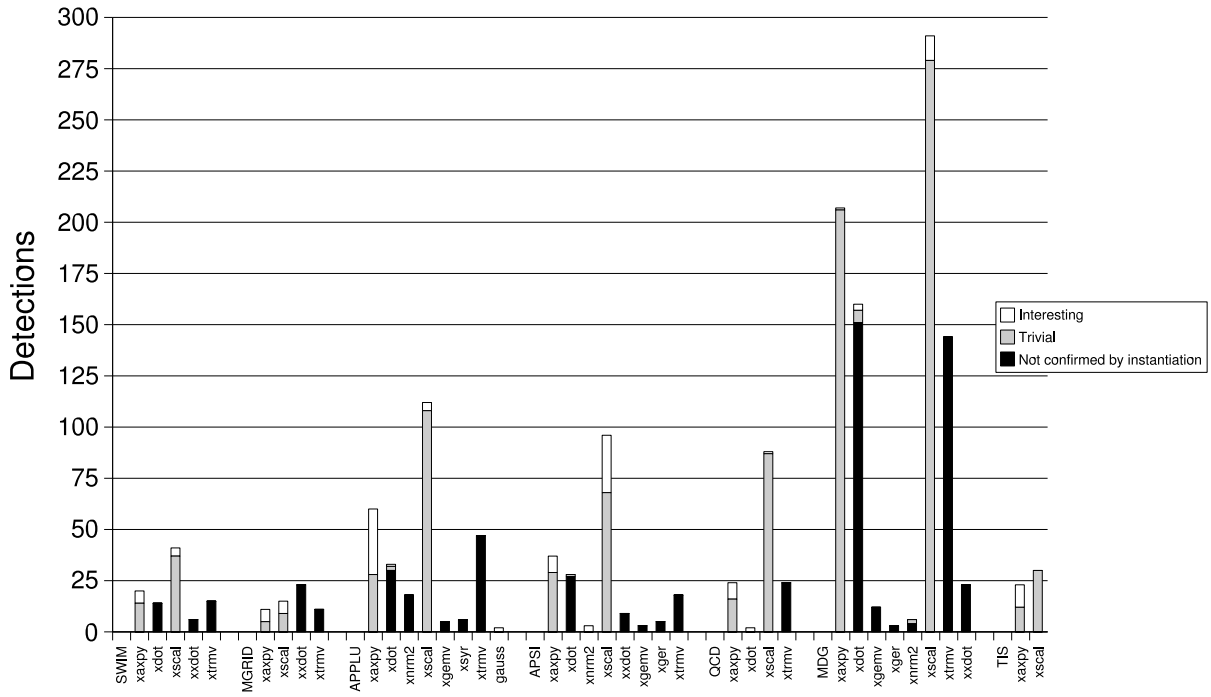



FIG. 10.4 – BLAS functions detected

```
s = a(0)
do i = 1,n
| if i mod 2 = 0 then
| | s = s + 1
| endif
| s = s + a(i)
enddo
r = s
```

```
s1 = a(0)
do i = 1,n
| if i mod 2 = 0 then
| | s2 = φ1(s1,s3) + 1
| endif
| s3 = φ2(s1,s2,s3) + a(i)
enddo
r = φ3(s1,s3)
```

$\phi$-function $\phi_2$ allows to control the execution of the loop and the conditional. Due to the approximation, we can choose either `s1`, `s2` or `s3`. As a consequence, our slicing does not take into account the number of iterations of `do` loops and the branches executed in the conditionals.

In addition, 56.4% of slices found are constituted of correct instances of patterns, but whose substitution does not lead to a performance improvement, since they work on too small input instances. They are often simple arithmetic expressions, such as $y = \alpha x + y$ for a `xaxpy`. Finally, 7.7% of the slices is constituted by interesting candidates whose substitution can potentially increase the program performance. Our algorithm seems to have discovered all of them, and particularly hidden candidates. Indeed, most slices found are interleaved with the source code, and deeply destructured. Most variations found are data-structure variations *e.g.* daxpy on a particular dimension of an array, and organization variations *e.g.* garbage code and temporaries. Moreover, semantics variations introduced in the pattern base allow to detect more than 50 % of the candidates.

| Kernel | % Not confirmed | % Trivial | % Interesting |
|---|---|---|---|
| 171.swim | 35.6 | 53.1 | 10.4 |
| 172.mgrid | 56.7 | 23.3 | 20.0 |
| 173.applu | 37.7 | 49.1 | 13.2 |
| 301.apsi | 31.2 | 49.2 | 19.6 |
| qcd | 17.4 | 74.6 | 8.0 |
| mdg | 39.9 | 58.3 | 1.8 |
| tis | 21.4 | 75.0 | 3.6 |
| SpecFP 2000 | 35.3 | 48.5 | 16.1 |
| Perfect Club | 35.9 | 61.4 | 2.7 |
| **Total** | **35.9** | **56.4** | **7.7** |

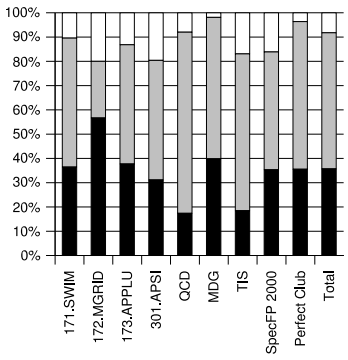



FIG. 10.5 – Repartition of detections

### Slicing time

Table 10.2 provides the execution time of the slicing method on a Pentium 4, 1.8 GHz with 256 MB RAM. Column *Analyzed lines of code* gives the size of the analyzed functions for each kernel, column *SSA* indicates the time spent in the SSA-graph generation (approximate data-flow analysis) needed by the slicing method *Slicing*. The execution time of the SSA-graph generation and the slicing method remains reasonnable, except for applu and mdg, since their SSA-graph have a more important average input degree $d_P \approx 5$.

| Kernel | Analyzed lines of code | SSA | Slicing | Total |
|---|---|---|---|---|
| 171.swim | 414 | 19 s | 31 s | 50 s |
| 172.mgrid | 243 | 3 s | 5 s | 8 s |
| 173.applu | 558 | 1 mn 50 | 1 mn 5 | 2 mn 55 |
| 301.apsi | 807 | 51 s | 1 mn 16 | 2 mn 7 |
| qcd | 355 | 40 s | 36 s | 1 mn 16 |
| mdg | 1264 | 8 mn 59 | 5 mn 53 | 14 mn 52 |
| tis | 171 | 0.5 s | 2 s | 2.5 s |
| **Total** | **3812** | **12 mn 42** | **8 mn 14** | **22 mn 10** |

TABLE 10.2 – Execution time of the slicing method

According to the complexity study given in chapter 5 section 5.7, the execution time of the slicing method can be written $\alpha \times {d_T}^2 {d_P}^2 TP$ where $\alpha$ is a constant, $d_T$ and $d_P$ denote the average input degree of the SSA-graph of the template and the program, and $T$ and $P$ denote the number of assignements within the template and the program. Table 10.3 provides the average ratio $\alpha$ = Effective_Time / ${d_T}^2 {d_P}^2 TP$. The ratio is constant, which confirms the theoretical result, and lead to write the execution time of the slicing method in the following manner:

$$\tau_{\text{slicing}}(T, P) \approx 13.52 \times 10^{-6} \times {d_T}^2 {d_P}^2 TP \text{ seconds}$$

Obviously, $\alpha$ depends on the machine used, and needs to be recomputed for another target machine. Combined with an evaluation $\tau_{\text{matching}}(T, P)$ of the execution time of the instantiation test, such a formula can be useful to predict the execution time of the whole recognition process.

| Kernel | Average $\alpha$ ($\times 10^{-6}$) |
|---|---|
| 171.swim | 13.97 |
| 172.mgrid | 14.57 |
| 173.applu | 15.65 |
| 301.apsi | 13.21 |
| qcd | 10.92 |
| mdg | 11.36 |
| tis | 14.94 |
| **Average** | **13.52** |

TABLE 10.3 – Experimental verification of the slicing complexity

### 10.2.2 Template Matching with Tree-Automata

For complexity reasons, this accurate (but expensive) test was not used in our experiments on the SpecFP 2000 and the Perfect Club benchmarks. Instead, we better use our matching method based on Huet and Lang's procedure, whose preformance is detailed in the next section. Nevertheless, reader interested by the performance of our tree-automata-based algorithm can have a look at experimental results given in chapter 8, page 143.

### 10.2.3 Template Matching with Semi-Unification

We provide thereafter some experimental results allowing to characterize the statistical behaviour of the instantiation test. As for the slicing method, we present an experimental verification of the complexity, then we provide some statistical results about the reaching set and closure caches used in our implementation.

#### Matching time

Table 10.4 provides the times spent in the exact data-flow computation of the program (*SARE*), and the instantiation test itself (*Matching*). As for the slicing method, the time spent in data-flow analysis remains reasonnable except for applu. Indeed, within the functions analyzed, most of assignments read and write the same array `rsd`, which leads to compute an important number of direct dependences. The important time spend in the matching step for mdg is due to the important amount of candidates provided by the slicing method. Most of them are trivial instances of BLAS level 1 functions. Additionally, about 5 minutes are spent in the computation of transitive closures while matching the dot product xdot and the matrix-vector product xtrmv.

Following the slicing method, we present now an experimental verification of the theoretical complexity. In chapter 7, section 7.6, we have shown that the theoretical complexity can be written $T \times P$, where $T$ denotes the number of assignements in the pattern, and $P$ denotes the number of assignements in the program.

Table 10.5 gives the average ratio $\alpha = \text{Effective\_Time} \ / \ (T \times P)$. The ratio is nearly constant, which confirms the theoretical result, and lead to write the execution time of the instantiation test as $\tau_{\text{matching}}(T, P) \approx 0.12 \times T \times P$ seconds.

| Kernel | Analyzed lines of code | SARE | Matching |
|---|---|---|---|
| 171.swim | 414 | 2 mn | 15 mn |
| 172.mgrid | 243 | 13 s | 35 s |
| 173.applu | 558 | 14 mn | 6 mn |
| 301.apsi | 807 | 2 mn | 5 mn |
| qcd | 355 | 33 s | 8 mn |
| mdg | 1264 | 4 mn | 45 mn |
| tis | 171 | 41 s | 40 s |
| **Total** | 3812 | **24 mn** | **1 h 22 mn** |

TABLE 10.4 – Execution time of the instantiation test

| Kernel | Average $\alpha$ |
|---|---|
| 171.swim | 0.23 |
| 172.mgrid | 0.07 |
| 173.applu | 0.10 |
| 301.apsi | 0.15 |
| qcd | 0.08 |
| mdg | 0.12 |
| tis | 0.10 |
| **Average** | **0.12** |

TABLE 10.5 – Experimental verification of the matching complexity

This relation complete $\tau_{\text{slicing}}(T, P)$ obtained previously, and could be used to predict the execution time of the whole recognition process. A possible application would be to provide an estimation of the optimization time to the compiler, which could cancel the optimization process whenever the estimated time is too important.

### Cache statistics

During the computation of reaching sets, the instantiation test stores the reaching sets, and the transitives closures computed with their corresponding regular expression in two different caches, in order to avoid to compute two times the same Presburger relation. We provide thereafter some statistics on the cache usage, showing a good reuse rate.

Table 10.6 provides the amount of misses and hits in the reaching set cache (*cache RS*), and the closure cache (*Cache Closures*). The hits in the reaching set cache often come from the states resulting of a Decompose rule. Indeed, the resulting states $t_i \stackrel{?}{=} t'_i$ have often the same reaching set than the source state $f(\vec{t}) = f(\vec{t'})$ since Decompose produces $\varepsilon$-transitions. Remark that it is not true in a general manner, since a resulting state can have several incomming transitions. The reaching set cache allows to take profit of the reuse rate of 10.2%, leading to a substantial performance improvement. All the transitive closures comes from the matching of xdot and xtrmv. Most of the matches have MSA with one cycle, leading to compute the transitive closure one time (miss), and to reuse it to compute the reaching sets of the following states (hits). This explains the important reuse rate of transitive closures (96.5 %). These two caches improve the performances of the instantiation test by 50 %, leading to a reasonnable execution time in practice.

| Kernel | Cache RS | | | Cache Closures | | |
|---|---|---|---|---|---|---|
| | Hits | Misses | Total | Hits | Misses | Total |
| 171.swim | 415 | 2420 | 2835 | 0 | 0 | 0 |
| 172.mgrid | 71 | 394 | 465 | 0 | 0 | 0 |
| 173.applu | 239 | 2781 | 3020 | 42 | 2 | 44 |
| 301.apsi | 382 | 2229 | 2611 | 124 | 6 | 130 |
| qcd | 45 | 1109 | 1154 | 39 | 2 | 41 |
| mdg | 1036 | 10971 | 12007 | 650 | 21 | 671 |
| tis | 118 | 375 | 493 | 0 | 0 | 0 |
| Total | 2306 | 20279 | 22585 | 855 | 31 | 886 |
| **Reuse rate** | **10.2 %** | | | **96.5 %** | | |

TABLE 10.6 – Cache statistics

### 10.2.4 Substitution

Once the BLAS functions discovered, it remains to perform the substitutions. This section points out the limitations of the substitution algorithm and presents the speed-ups resulting from substitution. We first recall some basic definitions on speed-up and performance gain.

**Definition 10.1 (Speed-up, gain).** *Consider a program $P$ with an optimized version $P_{opt}$ executing on an input $I$ during $T$ resp. $T_{opt}$ time units. The* speed-up *is defined by:*

$$Speed\text{-}up = \frac{T}{T_{opt}}$$

*While the* gain *is defined by:*

$$Gain = 1 - \frac{T_{opt}}{T}$$

The gains have the good property to be cumulated by a simple addition. Consider indeed two local optimizations leading separately to gains $G_1$ and $G_2$. The whole gain is then simply $G = G_1 + G_2$.

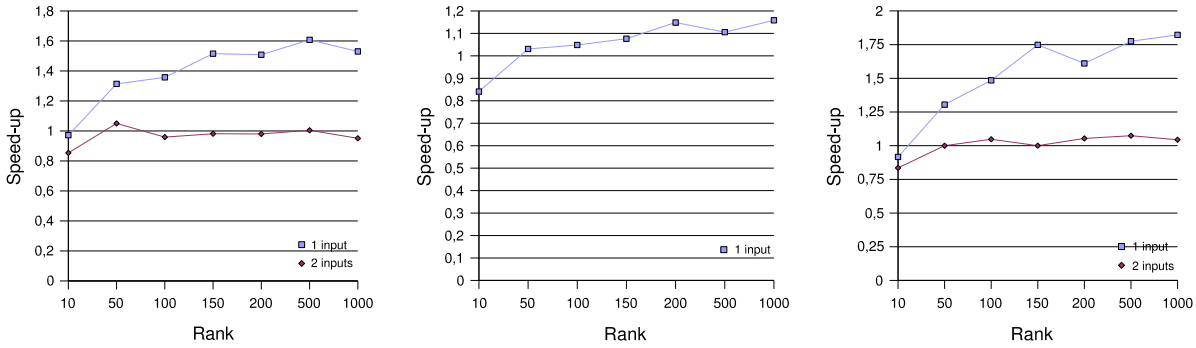
Since the substitution pass is not yet implemented, the substitution were applied by hand to a subset of slices representing a significant part of the execution time. According to the algorithm presented in chapter 9, section 9.4, we often need to set the function inputs before applying the substitution. Consider indeed the following program, where a scal has been recognized:

```
do i = 1,n
| x(i,2) = α * x(i,2)
| ...
enddo
```

Since xscal function cannot be applied directly to x, we need to add a memcopy to set the inputs:

```
x_dscal(1:n) = x(1:n,2)
call dscal(n,α,x_dscal,1)
x(1:n,2) = x_dscal(1:n)
```

Unfortunately, such a memcopy reduces the performance improvement of dscal, and may lead to a slow-down. Figure 10.6 shows the slow-downs caused by the input construction on three frequently used BLAS 1 functions. The speedups are obtained by comparing the performances of a direct implementation to the corresponding MKL function with one, then two memcopys to its the inputs. When one input at most needs to be constructed, we can already obtain speedups. Otherwise, the substitution leads to a slow-down of the program.




(a). xaxpy (b). xscal (c). xdot

FIG. 10.6 – Slow-down resulting from input construction

Most of the inputs to construct are constituted of a dimension of a bigger array, fixing the others dimensions ; for instance `x_input(k) = a(i,j,k)`. Assuming `a` is an array of dimension $n \times p \times q$, a solution to improve the performances would be to put directly `a` in the call, adding a stride equal to the product of the first dimensions: $n \times p$.

| Kernel | daxpy | | scal | | dot | | gauss | | Gain (%) | Speed-up |
|---|---|---|---|---|---|---|---|---|---|---|
| | Tried | Subst. | Tried | Subst. | Tried | Subst. | Tried | Subst. | | |
| 171.swim | 6 | 6 | 4 | 0 | 0 | 0 | 0 | 0 | **66.8** | **3.0** |
| 172.mgrid | 7 | 7 | 6 | 0 | 0 | 0 | 0 | 0 | 3.6 | 1.04 |
| 173.applu | 32 | 0 | 4 | 0 | 1 | 0 | 2 | 0 | 0 | 1 |
| 301.apsi | 8 | 2 | 28 | 0 | 0 | 0 | 0 | 0 | 0.7 | 1.0 |
| qcd | 8 | 0 | 1 | 0 | 2 | 0 | 0 | 0 | 0 | 1 |
| mdg | 0 | 0 | 10 | 3 | 2 | 0 | 0 | 0 | **16.5** | **1.19** |
| tis | 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| **Average** | | | | | | | | | **12.5** | **1.14** |

TABLE 10.7 – Speed-ups resulting from the substitution

Table 10.7 provides the speed-ups obtained after performing the substitution to the different kernels. For each kernel, we detail the gain resulting from the substitution of each function detected, and we provide the global speed-up. The experiments were carried out on an Itanium 2 897 Mhz with 2 GB RAM, bi-processor. Fortran programs were compiled using the g95 compiler. The selection of the functions to substitute were achieved incrementally, starting from the first substitution, and checking at each step whether a substitution increases the performance on a reference input.

| Kernel | Analyzed lines of code | SSA | Slicing | SARE | Matching | Total |
|---|---|---|---|---|---|---|
| 171.swim | 414 | 19 s | 31 s | 2 mn | 15 mn | 17 mn 50 s |
| 172.mgrid | 243 | 3 s | 5 s | 13 s | 35 s | 56 s |
| 173.applu | 558 | 1 mn 50 | 1 mn 5 s | 14 mn | 6 mn | 22 mn 55 s |
| 301.apsi | 807 | 51 s | 1 mn 16 s | 2 mn | 5 mn | 9 mn 7 s |
| qcd | 355 | 40 s | 36 s | 33 s | 8 mn | 9 mn 49 s |
| mdg | 1264 | 8 mn 59 | 5 mn 53 s | 4 mn | 45 mn | 1 h 3 mn |
| tis | 171 | 0.5 s | 2 s | 41 s | 40 s | 1 mn 24 s |
| Total | | | | | | **1 h 44 mn** |

TABLE 10.8 – Execution time of the recognition process

Despite the small number of functions found, the substitution leads to a global average gain of 12.5%, corresponding to a speed-up of 1.14. The important speedup of swim is due to six daxpy dealing with large arrays (with a size of 1334), and representing an important part of the execution time. The substitutions in mgrid deals with small arrays leading to small speedups. Several substitutions in apsi could not be achieved because of data dependences with the complementary slice. As well as swim, the speedups obtained in mdg are due to the important size of the arrays handled, and the important part taken in the execution time. No good substitution was found in applu, qcd and tis. Indeed, most of them needs two memcopy, and the remaining occurences work on too small input instances to obtain speed-ups.

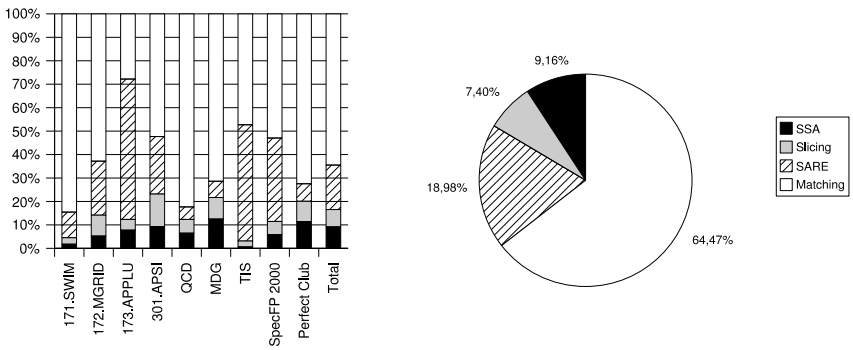



FIG. 10.7 – Repartition of the execution time

## 10.3 Conclusion

Table 10.8 provides the execution time of the whole recognition process, detailed module by module. We provide the execution time of the approximate data-flow analysis (*SSA*) followed by the slicing method *Slicing*, and the exact data-flow analysis *SARE* needed by the instantiation test *Matching*. In addition, figure 10.7 depicts the repartitition of the execution time for each kernel analyzed.

Except for applu, the execution times remains reasonnable. Since none of the works presented in chapter 3 present experimental results with precise execution time, we cannot compare their performances to ours. Concerning the repartition of the execution time, more than 60 % of the execution time is taken by the exact method whereas the slicing method takes only 7.4 %. This confirms that the slicing is fast method compared to the instantiation test. The same remark applies on the approximate and the exact data-flow analysis.

The experimentals results have shown that the optimization framework allows to obtain speed-ups for 3 kernels out of the 7 kernels analyzed. The two causes of slow-downs highlighted are (1) slices with a too small input size, and (2) slices needing several memcopy to build the inputs. Whereas it is generally impossible to predict the input size, we can check easily the number of input to copy. In addition, we have shown that the memcopys can be avoided in most cases by adding strides to the BLAS call, leading to good speed-ups.

# Chapter 11

# Conclusion

Assigning meanings to programs is a key-point to achieve high-level optimizations. This thesis presents a new and efficient approach to increase abstraction level of programs, by recognizing and abstracting known library functions. We can cope with a wide spectrum of algorithmic variations, from variable renaming to complex control transformations (*e.g. splitting, skewing or tiling*). A complete algorithmic framework is presented, formally proven, and validated on the real life programs of the SpecFP 2000 [54] and the Perfect Club [39] benchmarks. This chapter summarizes the contributions of this thesis, and details the perpectives opened in program analysis. Particularly, several improvements and applications in program analysis are presented.

## 11.1 Contributions

### A Complete Framework for Template Recognition and Replacement

In this thesis, we have proposed a new approach for program optimization, based on automatic rewriting of a program to use an optimized library. We provide a complete algorithmic framework covering all aspects of the problem from the recognition of the library functions within the program, to the substitution. Previous works attempts to solve this problem for general libraries [86, 71, 64, 17, 76], but most of them describe only the recognition process, keeping the substitution process for future work. To our knowledge, this thesis presents the first approach to handle generic libraries, whose functions are expressed by templates. Our method has been implemented, and validated by recognizing BLAS functions within the kernels of the SpecFP 2000 and the Perfect Club benchmarks. We obtain an average speedup of 1.14, with a peak at 3 for swim, a kernel of the SpecFP 2000.

### Two Strong Methods for Template Recognition

In addition, we provide the first approach to recognize *template* instances in an imperative program. Our recognition method relies on a *slicing method*, which detects the program parts which *possibly* matches the template. We demonstate that our slicing method does not miss any proper instance (conservativity, see proposition 5.2, page 81). Moreover, the

amount of wrong candidates slices remains low (under 50 %). We then check whether the slices are effectively instances of the template by using an *instantiation test*. We provide *two* instantiations tests based on different theoretical frameworks. The first one works in the context of *unification theory*, and adapts Huet and Lang's matching procedure [56]. The other one represents the template and the program with *tree-automata*, and achieves the matching by using a cartesian product. We show that the two algorithms differs by their complexity and their detection capabilities. The first instantiation test has a low complexity, but it can only handle template variables $X$ defined by a finite arithmetic expression. Whereas the second instantiation test is able to detect template variables involving a `do` loop despite a greater complexity. Another important contribution is the *demand-driven* data-flow analysis performed by the second instantiation test. Such a method allows to compute only the $\phi$-functions needed to achieve the matching. In addition to reduce the cost of the method, it breaks the static control restriction, allowing to handle general programs, including for instance `while` loops, and non-affine conditionals. The slicing method has been published in [5] and the instantiation test based on second-order matching in [4]. In [3] we investigate the demand-driven computation of $\phi$-functions. The `TeMa` tool implementing our recognition framework has been published in [2]. Finally, [6] addresses the application of our framework to recover program slices which can be expressed in SPL, a domain-specific language involved in SPIRAL [107].

#### A New Substitution Method

Once the proper instances are found within the program, it remains to select the slices whose replacement by a library call is possible, and interesting. We present an algorithm based on a *separability test* to decide whether a substitution is possible. Since several program slices can conflict (overlapping), we have to select a subset of slices whose substitution is possible, and will lead to the best performances improvement. Our selection algorithm is inspired of Metzger's approach [76], which provides a partial solution. The selection of the best substitution set is still an open problem, which we would like to address in a future work. We have also presented the algorithmic content needed to generate the code with the substitution.

## 11.2 Perspectives

### 11.2.1 Improvements

#### Agregation

Our method is able to detect template's occurrences with only one output statement. Such a limitation leads to miss several interesting slices. Indeed, given an unrolled daxpy:

```
y(0) = a*x(0) + y(0)
y(1) = a*x(1) + y(1)
y(2) = a*x(2) + y(2)
```

our method would detects individually the statements as an instance of daxpy, but not the whole daxpy since its outputs are shared by several statements.

One can remark that a daxpy is constituted of 1-dimension daxpy instances. A solution investigated in a recent paper [6] is to detect "atomic" daxpys, then to *agregate* them to make a larger daxpy. In a more general manner, consider an algorithm $A$ which produces an array, and a family of algorithms $(A_{\vec{i}})_{\vec{i}}$, where $A_{\vec{i}}$ outputs the $\vec{i}$-th array cell of $A$ for each possible input:

$$A_{\vec{i}}(I) \equiv_{\mathcal{H}} A(I)[\vec{i}]$$

For each relevant input $I$ and array index $\vec{i}$. Then $A$ is said to be an *agregation* of the $A_{\vec{i}}$. Agregation induces a hierarchy between algorithms, and particularly between templates. Typically, a daxpy is an agregation of several scalar daxpy, and a matrix-vector product is an agregation of dot products. Figure 11.1 provides an agregation hierarchy between some BLAS level 1 and 2 functions. $A \to B$ means "$B$ is an agregation of $A$ instances".

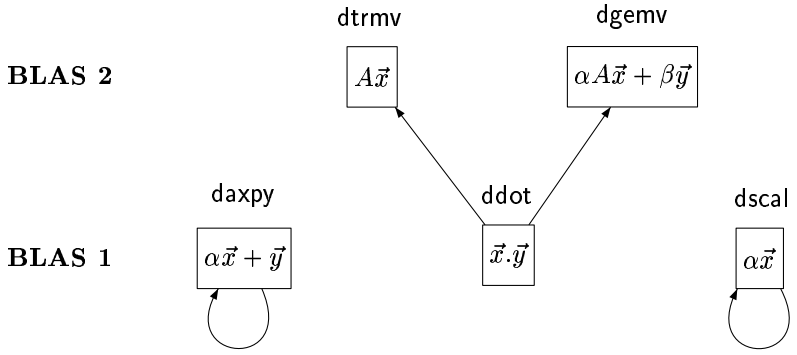



FIG. 11.1 – A possible agregation hierarchy for some BLAS level 1 and 2 functions

A way to explore would be to detect the templates of the hierarchy by using our recognition method. Then to agregate them in a bottom-up manner, from the leaves functions to the top functions. If $A$ is an agregation of $(A_{\vec{i}})_{\vec{i}}$, all combinations of $A_{\vec{i}}$ instances would be agregated, to yield $A$ instances.

### Breaking the Static Control Constraint

We have proposed an instantiation test which computes *on-demand* the program $\phi$-functions needed for the matching. This allows to handle general programs since unreached $\phi$-functions can be uncomputable. However, the templates instances *must* be static control programs. An usefull extension would be to allow the free variables to absorb program parts with uncomputable $\phi$-functions. For instance, we could handle gaussian eliminations where the choice of the pivot is performed by a general function `j = pivot(a,i)`. Two ways could be explored:

- A first approach is to handle $\phi$-functions as *functional symbols* ($+$, $\times$, etc). Given a program clause $S(\vec{i}) = \phi(S_1(\vec{i}), S_2(\vec{i}))$ unrolled with a template variable $X$, we could generate the transition $X, S(\vec{i}) \xrightarrow{\text{Full compute}} \phi(X, S_1(\vec{i})\ ,\ X, S_2(\vec{i}))$ which puts $\phi$ in the definition of $X$. This approach already works on simple examples where $\phi$ is not nested in a cycle. The others cases needs to be studied.

- One can also view $\phi$-functions as *SARE arrays* with a general definition such as $\phi(x_1 \ \ldots \ x_n)[\vec{i}] = x_i \quad \forall \vec{i} \in D_i$, where $\vec{i}$ denotes the iteration vector of $\phi$. Although the $D_i$ cannot be computed in a general manner, one can nevertheless built the unification automaton using the construction rules given chapter 8. The difficulty is then reported in the analysis pass, which have to deal with Presburger relations depending on unknown domains.

### Semantic Variations

Our recognition method is able to detect the program slices Herbrand-equivalent to an instance of a given template. Unfortunately, we are not able to handle program variations involving semantic properties of operators such as associativity or commutativity.

- An ambitious solution would be to extend the first instantiation test with the rules Mutate and Splice of equational unification (see [33], page 28). A direct application would introduce one OR-branching by state involving an equation starting with a functional symbol: $f(\vec{t}) \stackrel{?}{=} t'$ or $t \stackrel{?}{=} f(\vec{t'})$. This would lead to handle an unreasonnable amount of OR-branching combinations. To keep a reasonnable execution time, a more accurate application of equational rules has to be investigated.

- Using the agregation hierarchy, another solution would be to generate for each basic function a limited amount of semantic variations. This would allow to recognize semantic variations of higher-level functions *e.g.* BLAS 2 on the example. This empiric method, already applied with simple variations in our approach, seems promising.

### Transitive Closures

Our exact instantiation test rely on a symbolic execution of a counter automaton, which leads to compute the set of all possible counters values on a statement. As always, the problem comes from cycles which leads to compute transitive closures of Presburger relations. We currently use a semi-decision procedure due to Pugh [63], and implemented in the Omega calculator [62]. However, it is expensive and unfit for the intensive usage performed by our instantiation test. Since most of the transitive closures are simple cases *e.g.* $[i = i - 1, i > 0]^*$, we believe that a simple normalization technique should be relevant. In addition, we would like to address this problem in a more general way by studying Presburger automata [105].

### Third-order Matching

We provide an instantiation test able to semi-decide whether a program is an instance of a template, giving the corresponding free-variables values (unifiers). For instance, our instantiation test is limited to second-order unifiers (simple functions). We believe it would be interesting to find third-order unifiers (such as Caml functionals, functions taking functions in parameter). In a recent paper, De Moor et Sittampalam [31] propose a partial solution for functional programs, which seems interesting to investigate.

### 11.2.2 Applications

#### Model Checking

Model checking [96] aims to verify whether a program satisfy the temporal constraints specified by a linear temporal logic formula (LTL). This problem can solved in a general manner by associating to the program a Büchi automaton $\mathcal{A}_P$ recognizing a representative sub-set of its execution traces, and an automaton $\mathcal{A}_F$ which recognize all the traces satisfying the formula. It remains then to check whether $\mathcal{L}(\mathcal{A}_P) - \mathcal{L}(\mathcal{A}_F)$ is empty. We believe that templates can represent a sub-set of LTL formula, which could makes of our approach an interesting alternative to Büchi automata.

#### Translation Validation and Program Verification

Due to increasing complexity of optimizations, it becomes more and more difficult to certify a compiler. A solution is to check at compile time whether the optimizations do not hurt the program semantics. Our instantiation test is already able to check the equivalence before and after a source-to-source transformation on program parts with static control, and we believe that it could be extended to simple cases of programs with `while` loops and non-affine conditionals. The work of Denis Barthou on dataflow analysis with non-affine constraints [12] provides insights in this direction.

#### Parallel Skeletons

A *parallel skeleton* [29] is a generic pattern of parallel computation which can be parametrized by a small number of sequential functions. A simple example of parallel skeleton is PIPE, a template taking a list of jobs $f_1 \dots f_n$ with a flow dependence from $f_i$ to $f_{i+1}$, and executing them as different stages of a pipeline. Within this skeleton, the parallelism is achieved by allocating each job to a different processor. Using the OCaml style, its signature can be written:

```
PIPE: ('data→'data) list → ('data→'data)
```

Another example is FARM, a skeleton which inputs a job and executes it in parallel on a set of input data. The parallelism is achieved by using several processors to execute the job. Assuming that the different instances of the job shared a context, its signature can be written:

```
FARM: ('context→'input→'output)→'context → ('input list→'output list)
```

In a future work, we would like to address the application of *template* recognition to detect parallel skeletons. For the moment, we believe that our method could be applied to detect simple skeleton with a *fixed* number of jobs. For instance, the templates to detect PIPE and FARM would be:

```
do i = 1,n
| output(i) = X1(X2(X3(input(i))))
enddo
```

(a) PIPE with 3 stages

```
do i = 1,n
| output(i) = X(input(i))
enddo
```

(b) FARM

where the PIPE template is restricted to 3 stages. While writing the templates we have to check carefully that the data-flow contraints are respected. For instance, in PIPE the flow-dependence $X_3 \rightarrow X_2 \rightarrow X_1$ is given by the expression. Moreover, our matching procedure will define the $X_i$ by *pure functions*, ensuring the absence of other dependence between the $X_i$. Unfortunately, this definition of the PIPE template is too restrictive since it can just detects pipelines with three stages. In a future work, we would like the extend our recognition framework to handle templates with an arbitrary number of free-variables.

### Automatic Translation to a Domain-Specific Language

Domain-Specific Languages (DSL) allow to express programs in a more abstract and compact way than traditional imperative languages. In addition, DSL compilers are able to achieve more aggressive optimizations than traditional compilers. In a recent paper [6], we have investigated the re-engineering of Fortran programs to the SPL language [107], a DSL used in signal processing applications. We propose a preliminary approach to recover the program parts which can be expressed in SPL. We believe that our method can be applied to other DSLs. In addition to increasing the readability of the program, it would allow to take advantage of the optimizations performed by the DSL compiler.

# Personal bibliography

## Refereed International Conferences

- Christophe Alias and Denis Barthou. On Domain Specific Languages Reengineering. In *Proceedings of the 4th ACM International Conference on Generative Programming and Component Engineering (GPCE)*, Tallinn, Estonia, September 2005. LNCS 3676, pp. 63-77, Springer-Verlag.

- Christophe Alias and Denis Barthou. Deciding Where to Call Performance Libraries. In *Proceedings of the 11th IEEE/ACM International Euro-Par Conference*, Lisbon, Portugal. LNCS 3648, pp. 336-345, Springer-Verlag.

- Christophe Alias and Denis Barthou. Algorithm Recognition based on Demand-Driven Dataflow Analysis. In *Proceedings of the 10th IEEE Working Conference on Reverse Engineering (WCRE)*, Victoria, Canada, November 2003. IEEE Computer Society.

## Refereed International Workshops

- Christophe Alias. TeMa: an Efficient Tool to find High-Performance Library Patterns in Source Code. In *Proceedings of the First International Workshop on Patterns in High-Performance Computing (PatHPC)*, Urbana-Champaign, USA, May 2005.

- Christophe Alias and Denis Barthou. On the Recognition of Algorithm Templates. In *Proceedings of the 2nd International Workshop on Compiler Optimization meets Compiler Verification (COCV)*, Warsaw, Poland, April 2003. Published in Electronic Notes in Theoretical Computer Science (ENTCS) Vol. 82 No. 2.

# Bibliography


[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[2] C. Alias. TeMa: an efficient tool to find high-performance library patterns in source code. In *Proceedings of the First International Workshop on Patterns in High-Performance Computing (PatHPC)*, University of Illinois at Urbana-Champaign, USA, May 2005.

[3] C. Alias and D. Barthou. Algorithm recognition based on demand-driven data-flow analysis. In *Proceedings of the 10th IEEE Working Conference on Reverse Engineering (WCRE)*, pages 296–305, Victoria, Canada, November 2003. IEEE Computer Society.

[4] C. Alias and D. Barthou. On the recognition of algorithm templates. In *Proceedings of the 2nd International Workshop on Compiler Optimization meets Compiler Verification (COCV), 6th European Conferences on Theory and Practice of Software (ETAPS 2003)*, Warsaw, Poland, April 2003. Electronic Notes in Theoretical Computer Science (ENTCS) Vol. 82 No. 2.

[5] C. Alias and D. Barthou. Deciding where to call performance libraries. In *Proceedings of the 11th ACM International Euro-Par Conference*, pages 336–345, Lisbon, Portugal, August 2005. LNCS 3648, Springer-Verlag.

[6] C. Alias and D. Barthou. On domain specific languages reengineering. In *Proceedings of the 4th ACM International Conference on Generative Programming and Component Engineering (GPCE'05)*, pages 63–77, Tallinn, Estonia, September 2005. LNCS 3676, Springer-Verlag.

[7] V.H. Allan, R. . Jones, R.M. Lee, and S.J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.

[8] B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equalities of variables in programs. In *POPL'88*.

[9] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel C++ library. In *Wkshp. on Lang. and Comp. for Par. Comp. (LCPC)*, pages 193–208, August 2001.

[10] W. E. Arnoldi. The principle of minimize iterations in the solutions of matrix eigenvalues problems. In *Quart. Appl. Math.*, volume 9, pages 17–29, 1951.

[11] M. Balaeinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *METRICS'99*, pages 292–303, 1999.

[12] D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *J. of Parallel and Distributed Computing*, 40(2):210–226, February 1997.

[13] D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affine recurrence equations. In *8th International Euro-Par Conference*, page 309. Springer, LNCS 2400, 2002.

[14] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'2 IEEE International Symposium on Parallel and Distributed Computing*, pages 23–30, Ljubjana, october 2003.

[15] M. Berkelaar. *LP_SOLVE 5.1.1.3 Reference Manual.* Eindhoven University of Technology.

[16] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, october 1966.

[17] S. Bhansali and J.R. Hagemeister. A pattern-matching approach for reusing software libraries in parallel systems. In *Proc. of the Workshop on Knowledge-based Systems for the Reuse of Program Libraries.*

[18] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proceedings of the 6th International Conference on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67. Springer-Verlag, 1994.

[19] P. Boulet and X. Redon. SPPoC : manipulation automatique de polyèdres pour la compilation. *TSI*, 8:1–31, 2001.

[20] P. Briggs. Register allocation via graph coloring. Technical Report TR92-183, 24, 1998.

[21] C. Cascaval, L. DeRose, D. A. Padua, and D. A. Reed. Compile-time based performance prediction. In *LCPC*, 1999.

[22] C.-W. Chin, J. Bilmes, J. Demmel, and Krste Asanovic. The PHiPAC v1.0 matrix-multiply distribution. Technical report, October 23 1998.

[23] A. Cimitile, A. De Lucia, and M. Munro. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance: Research and Practice*, 8(3):145–178, 1996.

[24] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 1997. release October, 1rst 2002.

[25] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, and C. Pu. Spidle: A DSL approach to specifying streaming applications. In *GPCE*, 2003.

[26] P. Cousot and R.Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252, 1977.

[27] D. W. Currie, A. J. Hu, and S. Rajan. Automatic formal verification of DSP software. In *Design Automation Conference*, pages 130–135, 2000.

[28] K. Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* PhD thesis, Technische Universität Ilmenau, Germany, 1998.

[29] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE '93: Parallel Architectures and Languages Europe*, pages 146–160. Springer-Verlag, Berlin, DE, 1993.

[30] A. Darte, Y. Robert, and F. Vivien. *Scheduling and automatic Parallelization.* Birkhäuser, 2000.

[31] O. de Moor and G. Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 269(1–2):135–162, 2001.

[32] S. Demri, F. Laroussinie, and Ph. Schnoebelen. A parametric analysis of the state explosion problem in model checking. In volume 2285 of Lecture Notes in Computer Science, editor, *Proc. 19th Ann. Symp. Theoretical Aspects of Computer Science (STACS'2002)*, pages 620–631, March 2002.

[33] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. 1990.

[34] G. Dowek. Third-order matching is decidable. *Annals of Pure and Applied Logic*, 69:135–155, 1994.

[35] G. Dowek. *High-Order Unification and Matching.* Handbook of Automated Reasonning, Chapter 16. Elsevier Science, 2001.

[36] S. C. North K. P. Vo E. R. Gansner, E. Koutsofios. A technique for drawing directed graphs. *IEEE Trans. on Soft. Eng.*, 19(3), 1993.

[37] E. A. Emerson and C. L. Lei. Modalities for model-checking: Branching time strikes back. In *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, 1985.

[38] M. A. Ertl. Optimal code selection in DAGs. In *Principles of Programming Languages (POPL '99)*, 1999.

[39] M. Berry et al. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. *Int. J. of Supercomputer Applications*, 3:5–40, March 1989.

[40] T. Fahringer. *Automatic Preformance Prediction for Parallel Programs on Massively Parallel Computers.* PhD thesis, University of Vienna, 1993.

[41] T. Fahringer. Estimating and optimizing performance for parallel programs. *IEEE Comp.*, 28(11):47–56, November 1995.

[42] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[43] P. Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, February 1991.

[44] B. Fischer. Specification-based browsing of software component libraries. In *Automated Software Engineering*, pages 74–83, 1998.

[45] M. J. Fischer and M. O.Rabin. Super-exponential complexity of presburger arithmetic. In *Proceedings of the SIAM-AMS Symposium in Applied Mathematics*, volume 7, pages 27–41, 1974.

[46] G. Fowler. Cql - a flat file database query language. In *Proceedings of the USENIX Winter 1994 Technical Conference*, San Francisco, California, January 1994.

[47] M. Frigo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[48] K. Gallagher. *Using Program Slicing in Software Maintenance.* PhD thesis, University of Maryland, 1989.

[49] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In Lecture Notes in Computer Science 2102, editor, *Proceedings of CAV'01*, pages 53–65, 2001.

[50] R. Gerth, D. Peled, M. Vardi, , and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. PSTV 1995 Conference*, Warsaw, 1995.

[51] S. Ghosh, M. Martonosi, and S. Malik. Cahe miss equations: An analytical representation opf cache misses. In *ICS*, 1997.

[52] W.D. Goldfarb. Note on the undecidability of the second-order unification problem. *Theretical Comp. Sci.*, 13:225–230, 1981.

[53] M. Gupta. *Automatic Data Partitionning on Distributed Memory Multicomputers.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[54] J. Henning. SPEC CPU 2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[55] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[56] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[57] ILOG. *CPLEX 8.1 Reference Manual*, 2002.

[58] J.-J. Jeng and B. H. C. Cheng. Using formal methods to construct a software component library. In I. Sommerville and M. Paul, editors, *Proceedings of the Fourth European Software Engineering Confe rence*, pages 397–417. Springer-Verlag, 1993.

[59] J.-J. Jeng and B.H.C. Cheng. Using formal methods to construct a software component library. In *Proc. of 4th Eur. Soft. Eng. Conf.*

[60] W. L. Johnson and E. Soloway. PROUST: Knowledge-based program understanding. *IEEE Trans. on Software Engineering*, 11(3):267–275, March 1985.

[61] S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. Humphrey, J. Reynders, S. Smith, and T. Williams. Array design and expression evaluation in POOMA II. In *ISCOPE*, volume LNCS 1505, 1998.

[62] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Interface Guide. Technical report, Dept. of Computer Science, Univ. of Maryland, College Park, 1996. URL: `http://www.cs.umd.edu/projects/omega/`.

[63] W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its applications. *Int. J. of Parallel Programming*, 24(6):579–598, 1996.

[64] C. W. Kessler. Pattern-driven automatic parallelization. *Scientific Programming*, 5(3):251–274, 1996.

[65] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS'2001*.

[66] F. Lanubile and G. Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Trans. on S.E.*, 23(4):246–259, 1997.

[67] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[68] S. Letovski. Cognitive process in program comprehension. In E.Soloway and S. Iyengar, editors, *Empirical Studies of Programmers*, Norwood, 1986.

[69] V. Loechner and D. Wilde. Parametrized polyhedra and their vertices. *Int. J. of Parallel Programming*, 25(6), December 1977.

[70] J. Lyle and M. Weiser. Automatic bug location by program slicing. In *Proceedings of the second International Conference on Computers and Applications*, pages 877–883, 1987.

[71] B. Di Martino and G. Iannello. PAP recognizer: A tool for automatic recognition of parallelizable patterns. In *IWPC'04*, pages 164–174. IEEE Computer Society Press, 1996.

[72] V. Maslov. Lazy array data-flow dependence analysis. In *POPL'94*, pages 311–325, Portland, OR, 1994.

[73] Y. V. Matiyasevich. *Hilbert's Tenth Problem.* MIT Press, Cambridge, Massachusetts, 1993.

[74] D. Maydan, S. Amarasinghe, and M. Lam. Array dataflow analysis and its use in array privatization. In *POPL'93*, pages 2–15, Charleston, SC, January 1993.

[75] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[76] R. Metzger and Z. Wen. *Automatic Algorithm Recognition: A New Approach to Program Optimization.* MIT Press, 2000.

[77] M. Minoux. *Mathematical Programming, Theory and Algorithms.* Wiley, 1986.

[78] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler for prefetching. In *ASPL*, pages 62–73, 1992.

[79] S. S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann, 1997.

[80] G. Necula. Translation validation. In *PLDI'2000*, pages 83–95, 2000.

[81] G. C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.

[82] V. Padovani. Decidability of fourth-order matching. *Mathematical structures in computer science*, 10(3).

[83] D. Padua. Personnal communication, May 2005.

[84] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Trans. on S.E.*, 20(6):463–475, June 1994.

[85] J. Penix, P. Baraona, and P. Alexander. Classification and retrieval of reusable components using semantic features. In *Proceedings of the 10th Knowledge-Based Software Engineering Co nference*, pages 131–138, 1995.

[86] S. S. Pinter and R. S. Pinter. Program optimization and parallelization using idioms. *ACM Trans. on Programming Languages and Systems*, 1994.

[87] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *LCPC'93*, pages 546–566, Porland, OR, August 1993.

[88] M. Püschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, and R. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *J. of High Perf. Computing and Applications*, 1(18):21–45, 2004.

[89] X. Redon and P. Feautrier. Detection of scans in the polytope model. *Parallel Algorithms and Applications*, 15:229–263, 2000.

[90] T. Reps, S. Horwitz, and D. Binkley. Interprocedural slicing using dependence graphs. In *ACM Trans. of Programming Languages*, volume 12, pages 26–61, 1990.

[91] O. Ruthing, J. Knoop, and B. Steffen. Detecting equalities of variables: Combining efficiency with precision. In *Proceedings of the 6th International Static Analysis Symposium (SAS'99)*, pages 232 – 247. Springer-Verlag, Heidelberg, LNCS 1694, 1999.

[92] L. Réveillère, F. Mérillon, C. Consel, R. Marlet, , and G. Muller. A DSL approach to improve productivity and safety in device drivers development. In *15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, Grenoble, France, September 2000.

[93] J. G. Siek and A. Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *ISCOPE*, pages 59–70, 1998.

[94] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. on Software Engineering*, 10:595–609, 1984.

[95] A. Stepanov and M. Lee. The standard template library. ANSI X3J16-94-0095/ISO WG21-NO482.

[96] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First IEEE Symp. on Logic in Computer Science*, pages 322–331, 1986.

[97] T.L. Veldhuizen. Arrays in blitz++. In *ISCOPE*, volume LNCS 1505, 1998.

[98] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[99] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998.

[100] W. Wiedenbeck and V. Fix. Characteristics of the representation olf novice and experts programmers: An empirical study. *Int. J. Man-Machine Studies*, 1993.

[101] N. Wilde and M.C. Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[102] L. M. Wills. *Automated Program Recognition by Graph Parsing.* PhD thesis, MIT, July 1992.

[103] G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.

[104] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* 1989.

[105] P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *TACAS*, 2000.

[106] P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasonning about infinite computation paths. In *Proc. 24th IEEE Symp.on Foundations of Computer Science*, pages 185–194, Tucson, 1983.

[107] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A language and compiler for DSP algorithms. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, pages 298–308, 2001.

[108] W. Yang, S. Horwitz, and T. Reps. Detecting program components with equivalent behaviors. Technical report, University of Wisconsin, Madison, 1989.

[109] T. Yokoyama, Z. Hu, and M. Takeichi. Deterministic second-order patterns in program transformation. In *LOPSRT'2003.*

[110] H. Zima and B. Chapman. *Supercompilers for parallel and vector computers.* 1991.

[111] L. Zuck, A. Pnueli, Y. Fang, , and B. Goldberg. Voc: A methodology for the translation validation for optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.

# Index

## Résumé

La plupart des optimisations appliquent des transformations locales bas-niveau, sans se soucier du calcul exprimé par le programme. Bien que ces optimisations produisent des résultats satisfaisants, elles ne sont pas encore suffisantes, et amènent bien souvent le programmeur à utiliser des bibliothèques optimisées. Pour le moment, les bibliothèques optimisées doivent être appelées à la main. Apprendre et utiliser une nouvelle bibliothèque est malheureusement très fastidieux, et il est surprenant de voir le peu d'aide apporté par le compilateur. Une solution naturelle serait de chercher les occurrences immédiates des fonctions d'une bibliothèque dans le programme, et de les remplacer par l'appel correspondant.

Dans cette thèse, nous proposons une approche totalement automatique pour reconnaître dans un programme les occurrences des fonctions d'une bibliothèque optimisée, et les substituer par l'appel de fonction lorsque c'est possible et intéressant. Notre méthode est capable de détecter toutes les tranches de programme équivalentes aux fonctions recherchées au sens de l'équivalence de Herbrand ; un sous-ensemble de l'équivalence sémantique qui ne tient pas compte de la sémantique des opérations atomiques. En plus des fonctions, nous sommes également capables de trouver des instances de templates dans un programme. Une telle caractéristique rend possible la reconnaissance de bibliothèques de templates, et la réécriture d'un programme pour utiliser des templates. Une fois les instances trouvées dans le programmes, il reste à sélectionner les candidats dont le remplacement par un appel de fonction est possible et améliore effectivement les performances du programme. Nous proposons également un algorithme pour sélectionner les substitutions valides, et pour générer le code avec la substitution. La sélection du bon ensemble de substitution se fait par un système de notes. Deux autres solutions expérimentales basées sur du benchmarking sont proposées, l'une décrivant exhaustivement l'espace des substitutions, et l'autre testant les substitutions en suivant une approche gloutonne.

Notre approche a été implémentée dans l'outil `TeMa` (`Template Matcher`). `TeMa` représente plus de 17000 lignes de code C++ et OCaml, et a été appliqué à la détection des fonctions de la bibliothèque BLAS (Basic Linear Algebra Subroutines) dans les noyaux des benchmarks SpecFP 2000 et Perfect Club. Les résultats expérimentaux montrent un facteur d'accélération substantiel sur les noyaux swim, mgrid et mdg.



## Abstract

Most of compiler optimization techniques apply local transformations on the code, replacing sub-optimal fragments with better ones. They are often low-level, and applied without knowing what the code is supposed to do. Unfortunately, these optimizations are not enough to produce an optimal code, and leads the programmer to use performance libraries. For the moment, the library functions must be called by hand. However, learning and using a new library remains fastidious, and it is surprising how little the compiler helps the programmer in this task. A natural solution would be to search naive occurrences of library functions through the program, and to replace them by the corresponding call.

In this thesis, we propose a fully automatic approach to recognize occurrences of library functions in a program, and to substitute them, whenever it is possible and interesting, by a call to the library. Our approach is able to recognize all the program slices computing the same mathematical formula than the searched function. This allow to cope with organization, data-structure and control variations, and more generally with any program transformation which does not take operators properties (associativity, commutativity, etc.) into account. In addition to functions descriptions, we are also able to find template instances in the source code. Such a characteristic enable the recognition of template libraries, and the rewriting of a program to use templates. Once the proper instances are found within the program, it remains to select the slices whose replacement by a library call is possible, and interesting. We propose a complete algorithmic framework to select all valid substitutions, and to generate the corresponding code. To select a good substitution set, we propose a preliminary solution based on a system of marks. Two other experimental approaches based on benchmarking are also investiguated.

Our approach has been implemented in the `TeMa` tool (`Template Matcher`). `TeMa` represents more than 17000 lines of code, and was applied to the detection of the BLAS functions (Basic Linear Algebra Subroutines) within the kernels of the SpecFP 2000 and the Perfect Club benchmarks. Experimental results report a substantial acceleration factor for the kernels swim, mgrid and mdg.