THÈSE de DOCTORAT de l'UNIVERSITÉ de VERSAILLES

Spécialité : Informatique

présentée par

Denis BARTHOU

pour obtenir le titre de DOCTEUR de l'UNIVERSITÉ de VERSAILLES

Sujet de la thèse :

Analyse du Flot de Données pour Tableaux en Présence de Contraintes Non-affines

Array Dataflow Analysis in Presence of Non-affine Constraints

Soutenue le *23 février 1998*

Devant le jury composé de :

Claude	TIMSIT	Président
Michel	COSNARD	Rapporteur
François	IRIGOIN	Rapporteur
Paul	FEAUTRIER	Directeur
Corinne	ANCOURT	Examinatrice
Jean-François	COLLARD	Examinateur

Thèse préparée à l'Université de Versailles - St Quentin
au sein du laboratoire Parallélisme Réseaux Systèmes Modélisation
(PR*i*SM)

Remerciements

Je tiens ici à remercier sincèrement,

Paul Feautrier, pour sa disponibilité, ses judicieux conseils et ses reformulations synthétiques et claires de mes propos, parfois assez obscurs, qui m'ont beaucoup apporté, ainsi que pour tout ce qu'il m'a appris durant cette thèse,

Claude Timsit, qui a bien voulu être le président de mon jury, pour ses commentaires pertinents et pour l'éclairage nouveau qu'il a porté sur mes travaux,

Michel Cosnard et François Irigoin pour avoir accepté d'être mes rapporteurs, pour le soin avec lequel ils ont relu cette thèse et pour leurs remarques et suggestions intéressantes,

Corinne Ancourt d'avoir eu la gentillesse de participer à ce jury et Jean-François Collard, pour son soutien quotidien, son dynamisme de tous les instants et ses commentaires précieux.

Je suis également très reconnaissant à

William Pugh et David Wonnacott, avec qui j'ai eu de longues discussions qui m'ont beaucoup aidé dans la compréhension de leurs analyses et des miennes,

Tous les membres du CRI de l'École des Mines de Paris, et notamment Béatrice Creusillet, pour le temps qu'elle a passé à m'expliquer les analyses par régions,

Tous les membres du PR*i*SM, et plus particulièrement tous les membres de l'équipe, anciens ou actuels,

Ma famille, pour son soutien constant et inconditionnel.

Contents

Présentation **11**
I Introduction 11
I.1 Contexte 11
I.2 Analyses de dépendances 13
I.3 Plan 15
II Analyses de dépendances et parallélisation 15
II.1 Notations et définitions 15
II.2 Abstractions 16
II.3 Mise en œuvre du parallélisme 18
II.4 Conclusion 20
III Analyse exacte du flot des données 20
III.1 Modèle de programmes 20
III.2 Méthode polyédrique 21
III.3 Algorithme polynômial simplifié 22
III.4 Autres travaux 25
III.5 Conclusion 27
IV Analyse approchée de flot 27
IV.1 Modèle de programmes 27
IV.2 Quelle approximation ? 28
IV.3 Présentation du formalisme 30
IV.4 Amélioration de l'approximation 33
IV.5 Traduction des propriétés 36
IV.6 Construction d'ensembles de sources 38
IV.7 Implémentation 38
IV.8 Autres travaux 40
IV.9 Conclusion 43
V Applications de l'analyse de flot 44
V.1 Vérification de programmes 44
V.2 Détection des récurrences 46
V.3 Expansion statique maximale 46
V.4 Ordonnancement 50
V.5 Conclusion 51
VI Conclusion 52

VI.1 Contributions . . . 52
VI.2 Perspectives . . . 53

1 Introduction 55
1.1 Context . . . 55
1.2 Dependence Analysis . . . 57
1.3 Overview . . . 59

2 Dependence Analysis in Parallelization 61
2.1 Notations and Definitions . . . 61
2.2 Dependences Abstractions . . . 62
2.2.1 Definition of dependences . . . 62
2.2.2 Traditional dependence representations . . . 63
2.3 Exposing parallelism . . . 65
2.3.1 Code transformations/ Scheduling . . . 65
2.3.2 Avoiding memory reuse . . . 67
2.3.3 Detection of recurrences . . . 69
2.4 Conclusion . . . 69

3 Exact Array Dataflow Analysis 71
3.1 Program Model . . . 71
3.2 Polyhedral Method . . . 72
3.2.1 Formal Solution . . . 72
3.2.2 Evaluation Techniques . . . 73
3.2.3 Optimizations . . . 76
3.3 Simplified Polynomial Algorithm . . . 79
3.3.1 Another adaptation of Fourier-Motzkin elimination . . . 79
3.3.2 Principle of the algorithm . . . 80
3.3.3 Handling the subscript equation . . . 80
3.3.4 Elimination step . . . 81
3.3.5 Complexity of the algorithm . . . 83
3.3.6 Domain of the algorithm . . . 84
3.3.7 Extension of the domain to parametric coefficients . . . 84
3.3.8 Performances . . . 87
3.3.9 Alternative technique for combining quasts . . . 88
3.4 Related Work . . . 90
3.4.1 Pugh and Wonnacott's approach . . . 90
3.4.2 Maslov's approach . . . 93
3.4.3 Maydan et al's approach . . . 93
3.4.4 Heckler and Thiele's approach . . . 94
3.4.5 Going beyond static control programs . . . 94
3.5 Conclusion . . . 95

4 Approximate Array Dataflow Analysis 97
4.1 Program Model . . . 97
4.2 Which Approximation? . . . 98
4.2.1 The need for an approximation . . . 98
4.2.2 Motivating examples . . . 100
4.2.3 From exact to approximate analysis . . . 101
4.3 Description of the Formalism . . . 103
4.3.1 Non-affine constraints . . . 104
4.3.2 Parameterization . . . 106
4.3.3 Fuzziness . . . 108
4.3.4 Example E1: `while` loop . . . 110
4.3.5 Example E2: `if..then..else` construct . . . 112
4.3.6 Example E3: Non-affine array subscript . . . 113
4.3.7 Example E4: Non-affine loop bounds . . . 113
4.4 Reducing Fuzziness . . . 114
4.4.1 General properties . . . 114
4.4.2 Characterization of the parameters . . . 116
4.4.3 Structural analysis . . . 117
4.4.4 Iterative analysis . . . 118
4.5 Translating Properties . . . 122
4.5.1 The resolution method . . . 123
4.5.2 Boolean constraints . . . 125
4.5.3 Constraints as inequalities on non-affine functions . . . 127
4.5.4 Simplifications . . . 131
4.5.5 A particular case: Exact results . . . 131
4.5.6 Conclusion . . . 132
4.6 Building Source Sets . . . 132
4.6.1 Context quasts . . . 132
4.6.2 Removing parameters . . . 133
4.6.3 Example E1: `while` loop . . . 134
4.6.4 Example E2: `if..then..else` construct . . . 135
4.6.5 Example E3: Non-affine subscript . . . 137
4.6.6 Example E4: Non-affine loop bounds . . . 138
4.7 Implementation . . . 139
4.7.1 Overview of Caravan . . . 139
4.7.2 Translation module . . . 140
4.7.3 Example . . . 141
4.8 Related Work . . . 142
4.8.1 Analyzing individual array references . . . 143
4.8.2 Analyses using summaries . . . 145
4.8.3 Detailed comparison with Wonnacott's work . . . 146
4.8.4 Detailed comparison with Creusillet's work . . . 150
4.9 Conclusion . . . 158

5 Applications **161**
5.1 Program Checking 162
5.2 Traditional Code Transformations 163
5.3 Detection of Recurrences 164
5.4 Maximal Static Expansion 166
5.4.1 Motivating examples 167
5.4.2 Related work 171
5.4.3 Static expansion 172
5.4.4 Expansion scheme 173
5.4.5 Algorithm 175
5.4.6 Examples 178
5.4.7 Conclusion 182
5.5 Scheduling 183
5.6 Conclusion 185

6 Conclusion **187**
6.1 Contributions 187
6.2 Future Work 188

Personnal Bibliography **191**

References **193**

Index **204**

Sommaire

Présentation **11**
Résumé en français.
1 **Introduction** **55**
Contexte. Analyse de dépendances.
2 **Analyses de dépendance et parallélisation** **61**
Abstractions des dépendances. Expression du parallélisme
3 **Analyse exacte du flot des données** **71**
Méthode polyédrique. Algorithme polynomial. État de l'art et comparaisons.
4 **Analyse approchée de flot de données** **97**
Formalisme. Approximation. Implémentation. État de l'art et comparaisons.
5 **Applications** **161**
Vérification. Récurrences. Expansion statique.
6 **Conclusion** **187**
Contributions. Perspectives.
Bibliographie personnelle **191**
Références bibliographiques **193**
Index **204**

Présentation

Ce chapitre offre un résumé en français des chapitres suivants écrits en anglais. Son organisation est le reflet de la structure de la thèse et les sections et sous-sections correspondent respectivement aux chapitres et à leurs sections. Le lecteur désirant approfondir un des sujets présentés ci-dessous pourra donc se reporter à la partie correspondante en anglais pour y trouver le détail des algorithmes ainsi que des exemples.

I Introduction

I.1 Contexte

Les progrès accomplis en termes de technologie du processeur, et notamment l'augmentation de la fréquence, n'ont pas été suivis par une amélioration similaire de la capacité et des performances mémoire. La lenteur relative des accès mémoire, même avec une hiérarchie de caches, empêche les processeurs d'atteindre leurs performances optimales. Les machines séquentielles n'offrent désormais plus une puissance suffisante pour traiter de nombreuses applications, parmi lesquelles, outre les « Grands Challenges », on trouve des simulations ou des applications multimédia. Les simulations qui prennent une place de plus en plus importante dans de nombreux domaines, soit pour des raisons économiques (en aérodynamique par exemple), soit parce que l'expérimentation n'est pas possible (en sismologie, cosmologie,...), requièrent de très grandes quantités de mémoire et les applications multimédia quant à elles imposent de lourdes contraintes sur les temps d'exécution.

L'idée est alors de diviser le programme ou les données initiales en différentes parties qui sont traitées par autant de processeurs. Cette idée d'exécution parallèle a conduit dans un premier temps aux architectures vectorielles, comme celle du Cray I. Les mêmes instructions s'exécutent sur un vecteur de données. Toutefois, l'efficacité de telles architectures est très liée à la régularité du code parallèle. Une façon plus générale d'améliorer les performances au niveau du processeur est obtenue grâce aux architectures super-scalaires ou VLIW. Différentes instructions sont exécutées en parallèle. Enfin, une autre approche consiste à multiplier le nombre de processeurs dans une machine dans le but de multiplier de manière similaire les performances. La mémoire est alors soit

distribuée entre les processeurs (comme pour l'IBM SP2) soit partagée (comme pour le CRAY YMP).

Afin d'atteindre des performances élevées sur un ordinateur parallèle, un programme, ou tout au moins l'algorithme qu'il implémente, doit évidemment contenir un degré significatif de parallélisme. Exhiber ce parallélisme et tenir compte des spécificités de la machine cible est une tâche qui revient au programmeur et/ou au compilateur. De plus, le programme parallèle doit être portable afin de pallier l'obsolescence rapide des machines parallèles. Deux possibilités sont offertes au programmeur :

- Les langages explicitement parallèles. La plupart des langages proposés sont des extensions de langages séquentiels, comme par exemple OCCAM, FORTRAN D [72], VIENNA FORTRAN [132] ou HPF [57]. Certaines extensions sont des bibliothèques : PVM [119] et MPI [63] par exemple. Cette approche rend possible la programmation d'algorithmes parallèles performants. Néanmoins, outre l'algorithmique, le programmeur doit également s'occuper d'opérations plus techniques et dépendant de la machine, comme la répartition des données entre les processeurs suivant leurs capacités mémoire, les communications, les synchronisations,... Cela exige une connaissance approfondie du fonctionnement de la machine cible. Des efforts notables ont été faits avec HPF pour qu'une partie de ce travail soit effectué par le compilateur, mais alors le programmeur doit avoir une bonne connaissance de ce que le compilateur est capable de faire et de la manière dont il le fait... Quant à la portabilité, cela a été l'un des objectifs principaux d'HPF ainsi que des bibliothèques PVM et MPI. Toutefois, portabilité ne signifie pas efficacité, et certains choix faits sur le modèle de machine parallèle affectent inévitablement les performances.

- La parallélisation automatique de langages séquentiels de haut niveau. Les avantages évidents de cette approche sont la portabilité, la simplicité de programmation et le fait que même les vieux codes séquentiels non documentés sont en théorie parallélisables. Cependant, la tâche qui revient au compilateur est énorme. En effet, le programme doit dans un premier temps être analysé afin de comprendre au moins partiellement ce qu'il fait et quelles sont les parties qui pourront être parallélisées. Ensuite, ce parallélisme doit être exhibé tout en tenant compte des spécificités de la machine. Même pour des programmes très simples et pour un modèle de machine simplifié, l'«optimalité» de l'une ou l'autre de ces étapes ne peut être atteinte, pour des raisons de décidabilité. En fait, il existe un large éventail de techniques de parallélisation, de complexités et d'efficacités variées. La difficulté vient du choix de la méthode à appliquer, choix qui est jusqu'à présent guidé par l'expérience du programmeur. Ainsi, de la même façon que pour HPF, le processus de parallélisation demande un effort de la part du programmeur ainsi que du compilateur.

> Le langage le plus couramment utilisé pour la parallélisation «automatique» est FORTRAN 77. En effet, de nombreuses applications scientifiques sont écrites dans ce langage et FORTRAN n'autorise que des structures de données (scalaires et tableaux) et de contrôle relativement simples. Des études ont toutefois été menées pour la parallélisation du C ou de langages fonctionnels tel que le PROLOG ou le LISP. Il existe déjà de nombreux paralléliseurs automatiques et outils de parallélisation : BOUCLETTE du LIP à Lyon [21], LOOPO du FMI à Passau [62], PARAFRASE-2 et POLARIS du CSRD de l'université de l'Illinois [100, 20], PAF de l'université de Versailles, PIPS de l'école des Mines de Paris [73] et SUIF de l'université de Stanford [3] entre autres, ainsi qu'un certain nombre d'outils commerciaux d'aide à la parallélisation, comme CFT, FORGE, FORESYS ou KAP.

On s'intéresse plus particulièrement dans cette thèse aux techniques de parallélisation automatique et plus spécifiquement au type d'analyses détectant les parties d'un programme qui sont parallèles.

I.2 Analyses de dépendances

La parallélisation automatique requiert une compréhension approfondie du comportement du programme à paralléliser. Cette connaissance permet ensuite au compilateur d'exhiber le parallélisme, d'améliorer la localité des données ou l'utilisation de la mémoire en transformant le code, ainsi que de détecter par exemple les récurrences, dans le but de produire un programme parallèle efficace.

Le comportement du programme est complètement déterminé par la suite des valeurs prises par les variables durant l'exécution. L'analyse des valeurs des variables fait partie des outils classiques des compilateurs actuels et est appelée *analyse du flot des données* [1]. L'interprétation abstraite [32] offre un cadre formel pour ce type d'analyse, l'information concernant les valeurs des variables étant trouvée en exécutant le programme dans une nouvelle sémantique approchant la sémantique d'origine. Les premières analyses se sont attachées à trouver des propriétés sur les valeurs des variables scalaires. Une équation de flot de données est associée à chaque instruction du programme et ce système d'équations est ensuite résolu soit itérativement soit directement [115]. Ces méthodes ont depuis lors été étendues aux tableaux, mais elles considèrent les tableaux comme des éléments indivisibles ou ne considèrent que des ensembles d'éléments de tableaux.

Les techniques de vectorisation et de parallélisation sont basées principalement sur la détection du parallélisme caché par des références indépendantes à des parties distinctes de tableaux. Deux opérations sont dites en dépendance si l'une d'elles écrit dans la cellule mémoire accédée par l'autre. Comme pour de nombreuses propriétés du programme, les dépendances peuvent être trouvées par une analyse traditionnelle du flot des données. Cependant, des techniques

plus spécifiques ont été mises au point pour traiter les dépendances entre tableaux. Il existe ainsi de nombreux tests de dépendances, pour vérifier la légalité d'une transformation de code [7]. Toutefois, ces tests ne sont pas exacts en général et ne permettent pas de distinguer une vraie dépendance qui décrit un vrai flot d'information entre opérations, d'une dépendance où la valeur est détruite entre les deux opérations en dépendance. Des méthodes plus précises ont vu le jour pour calculer, pour chaque élément de tableau lu par une opération (appelée le *puits*), l'opération (la *source*) qui a produit la valeur lue. Ces méthodes sont appelées *analyses de dépendances de flot de données pour tableaux* [52], ou analyses de dépendances basées sur les valeurs [103]. Ce type d'information permet d'améliorer significativement les transformations de code et les optimisations mémoire, et par conséquent permet d'améliorer les performances du code parallèle. Les analyses exactes de dépendance de flot reposent sur deux choses :

- chaque opération, c'est-à-dire, chaque instance d'instruction du programme, est définie par son instruction et une représentation finie de la date logique d'exécution. Cette abstraction de la date d'exécution doit être liée aux structures de contrôle afin de pouvoir par la suite utiliser le résultat de l'analyse pour transformer le code. En général, les programmes contenant des appels récursifs de procédure ou des `goto` non structurés ne peuvent pas être analysés de façon exacte. La plupart des formalismes proposés à ce jour se limitent aux structures de contrôle que sont la séquence, les boucles et les conditionnelles. La date logique d'exécution est alors représentée par le vecteur des compte-tours des boucles englobantes.

- Une fois le problème de dépendance formulé, des outils de programmation linéaire entière sont utilisés afin de calculer la solution exacte vérifiant toutes les contraintes de dépendances. Il faut alors que ces contraintes soient affines en les variables et paramètres du programme linéaire. Généralement, les variables choisies sont les compte-tours des boucles. Ainsi, tous les indices de tableaux, toutes les bornes de boucles et tous les prédicats des conditionnelles doivent être affines en les compte-tours. Les programmes à une seule procédure dont les seules structures de contrôle sont la séquence, les boucles et les conditionnelles, vérifiant ces contraintes de linéarité sont appelés *programmes à contrôle statique*.

L'analyse exacte de dépendance de flot pour tableaux fait par conséquent des hypothèses très fortes sur les programmes analysés et l'utilisation des outils de programmation linéaire montre que ce type d'analyse est coûteux en général. De nombreux efforts ont été faits afin de réduire ce coût et nous proposons également dans cette thèse une méthode efficace pour la plupart des programmes à contrôle statique permettant de plus de traiter certains cas de non-linéarité. Mais le principal sujet de cette thèse est de proposer un cadre général pour l'analyse du flot des données pour tableaux en présence de contraintes non-affines.

Le domaine d'une telle analyse de dépendances inclus les programmes utilisant des boucles `while`, des conditionnelles aux prédicats non-affines et des indices de tableaux non-affines. Le principe est de calculer pour chaque puits non pas une mais un ensemble de sources possibles. L'analyse, bien qu'approchée, opère au niveau des opérations et des éléments de tableaux. On ne s'intéressera pas aux problèmes des graphes de flot de contrôle complexes introduits par des `goto` ou des appels de procédures. Afin d'affiner l'approximation, on présente deux méthodes, appelées analyses structurelle et itérative, qui trouvent des relations sur les contraintes non-affines. Étant donné n'importe quel ensemble de relations sur ces contraintes, obtenu par les méthodes proposées ici ou par d'autres analyses, on décrit une technique calculant un ensemble de sources possibles et on donne un critère pour que cet ensemble soit le plus petit possible. Ce dernier résultat est très important car il montre que l'analyse de flot proposée, bien que plus coûteuse, donne des résultats plus précis que d'autres méthodes approchées. Cela justifie son utilisation pour certaines applications. Enfin, comme notre analyse est une extension de l'analyse exacte du flot de données pour tableaux, on présentera les adaptations des applications usuelles de l'analyse exacte à l'analyse approchée et les améliorations apportées. On étudiera plus particulièrement le cas de l'expansion de tableaux.

I.3 Plan

L'organisation de ce résumé est la suivante : la section II décrit le rôle de l'analyse de dépendances dans le processus de parallélisation. La section III présente l'analyse exacte de flot pour tableaux formalisée par Feautrier ainsi qu'une méthode plus simple en section III.3. Le cadre général pour une analyse approchée est proposé en section IV. Finalement, les applications qui en découlent sont décrites en section V.

II Analyses de dépendances et parallélisation

Cette section, correspondant au chapitre 2, décrit le type de dépendances utilisées ainsi que les techniques permettant d'exhiber du parallélisme.

II.1 Notations et définitions

Tout au long de cette thèse, nous utiliserons les notations suivantes : la k$^{\text{ième}}$ composante d'un vecteur x est noté $x[k]$. Le sous-vecteur construit à partir des composantes k à l sera noté $x[k..l]$. Si $k > l$, alors ce vecteur sera par convention le vecteur de dimension 0, qui est noté $[]$. De plus, $\ll$ est l'ordre lexicographique strict sur les vecteurs entiers et sera considéré comme la relation d'ordre par défaut sur l'ensemble de ces vecteurs. Les maxima et minima d'ensembles seront pris par rapport aux ordres implicites sur ces ensembles (par exemple, max sur un ensemble de vecteurs signifie $\max_{\ll}$).

Une instance d'une instruction `S` est notée $\langle \mathtt{S}, x\rangle$, où x, le vecteur d'itération de `S`, est le vecteur construit à partir des compteurs des boucles entourant `S` (y compris les boucles **`while`**), de la plus externe à la plus interne.

Par convention, les instructions d'un programme sont notées en lettres majuscules et en style machine à écrire. Les ensembles sont en lettres majuscules et les opérations (instances d'instructions) sont notées par des lettres de l'alphabet grec. Enfin, les fonctions utilisent les mêmes conventions typographiques que les objets qu'elles retournent.

Les *paramètres de structures* sont des constantes symboliques et les *contraintes non-affines* sont des équations ou inéquations qui dépendent de variables autres que des compte-tours et des paramètres de structures, et/ou dépendent de façon non-linéaire de ces compte-tours et paramètres.

II.2 Abstractions

Dans un programme séquentiel, l'ordre d'exécution entre opérations, noté $\prec$, est total. En fait, cet ordre total est imposé par la sémantique des structures de contrôle du langage séquentiel (boucles, séquence,...). Le but de l'analyse des dépendances est d'extraire de l'ordre séquentiel $\prec$ le plus petit ordre partiel $\prec_{//}$ préservant le comportement du programme, autrement dit, préservant les valeurs prises par les variables durant l'exécution. Les opérations qui ne sont pas comparables par $\prec_{//}$ peuvent être exécutées dans n'importe quel ordre ou en parallèle. Deux opérations sont en dépendance de données lorsqu'elles accèdent à la même cellule mémoire et que l'une d'elles est une écriture. Bernstein [13] a distingué trois types de dépendances de données, suivant l'ordre dans lequel se font les accès : les dépendances de flot (écriture puis lecture), de sortie (écriture puis écriture) et anti-dépendance (lecture suivie d'une écriture). Les dépendances de sortie et anti-dépendances ne sont que le résultat d'une réutilisation d'une cellule mémoire. Elles sont introduites par les techniques de programmation et n'ont pas de signification algorithmique. D'autre part, il est important de noter que de nombreuses dépendances sont obtenues par transitivité : une dépendance de sortie suivie d'une dépendance de flot donne une dépendance de flot. La valeur produite par l'opération d'écriture de cette dépendance est en fait *recouverte* ou *tuée* par la valeur donnée par la seconde écriture de la dépendance de sortie. Les dépendances de flot qui ne peuvent pas être obtenues par transitivité sont appelées dépendances *directes*. Le sous-graphe du graphe de dépendances composé uniquement des dépendances directes est appelé *graphe du flot des données* (GFD).

En général, le nombre de dépendances dans un programme dépend du nombre d'itérations des boucles, qui peut ne pas être connu à la compilation. De plus, les conditions de dépendance doivent être en général approchées. Ainsi on doit choisir une abstraction finie pour représenter les dépendances. De nombreuses représentations ont étés proposées et on décrit dans la suite les plus

usuelles, illustrées sur le programme tiré de [43].

```
  program sample
   do i=1, n
    do j=1, n
S    a(i,j) = a(j,i) + a(i,j-1)
    enddo
   enddo
```

Par ordre décroissant de précision, les dépendances peuvent être représentées par :

- des relations de dépendances/quasts [102, 52] : les dépendances de $\langle \texttt{S}_1, i_1 \rangle$ vers $\langle \texttt{S}_2, i_2 \rangle$ sont représentées par $\{(i_1, i_2) \,|P(i_1, i_2)\}$ avec P une formule de Presburger, c'est-à-dire une formule de logique du premier ordre dont les prédicats sont des inégalités affines sur des variables entières; les quasts sont des arbres de sélection pouvant décrire les dépendances avec la même précision que les formules de Presburger. On donne une description plus détaillée des quasts dans la section III.

  Dans le programme `sample`, il y a trois catégories de dépendances :
  - des dépendances de flot de $\langle \texttt{S}, i, j \rangle$ vers $\langle \texttt{S}, j, i \rangle$, avec $1 \leq i < j \leq n$;
  - des anti-dépendances de $\langle \texttt{S}, j, i \rangle$ vers $\langle \texttt{S}, i, j \rangle$, avec $1 \leq i < j \leq n$;
  - des dépendances de flot de $\langle \texttt{S}, i, j \rangle$ vers $\langle \texttt{S}, i, j+1 \rangle$, avec $1 \leq i \leq n, 1 \leq j < n$.

  Toutes ces dépendances sont représentées exactement par des relations de dépendance. Elles sont définies respectivement par :

$$\begin{gathered}\{(i,j),(i',j') \,|i' = j \wedge j' = i \wedge 1 \leq i < j \leq n\}\,,\\ \{(i,j),(i',j') \,|i' = j \wedge j' = i \wedge 1 \leq i < j \leq n\}\,,\\ \{(i,j),(i',j') \,|i' = i \wedge j' = j+1 \wedge 1 \leq i \leq n \wedge 1 \leq j < n\}\,.\end{gathered}$$

- des polyèdres ou des cônes de dépendance [74] : les dépendances de $\langle \texttt{S}_1, i_1 \rangle$ vers $\langle \texttt{S}_2, i_2 \rangle$ sont approchées par l'ensemble $\{i_2 - i_1 \,|P(i_1, i_2)\}$ avec P un polyèdre ou un cône;

  Les dépendances du programme `sample` peuvent toutes être représentées exactement par un polyèdre.

- des vecteurs de distance/direction [127] : les dépendances de $\langle \texttt{S}_1, i_1 \rangle$ vers $\langle \texttt{S}_2, i_2 \rangle$ sont représentées par une approximation de $i_2 - i_1$. Pour chacune de ses composantes, cette différence est soit constante, soit approchée par son signe $(+, -)$ ou par $*$ si le signe est inconnu;

  Les dépendances du programme `sample` sont approchées respectivement par les vecteurs $(+, -)$, $(+, -)$ et $(0, 1)$.

– ou des niveaux de dépendances [2] : le niveau d'une boucle est le nombre de boucles l'englobant, plus un (la boucle la plus externe a pour niveau 1). Étant donné une dépendance entre deux opérations, si ces opérations ne s'exécutent pas à la même itération d'une même boucle, alors cette boucle *porte* la dépendance. Une dépendance est représentée par le niveau de la boucle la plus externe qui la porte.

  Les niveaux de dépendances pour le programme `sample` sont respectivement $1, 1$ et 2.

II.3 Mise en œuvre du parallélisme

Transformations de code et ordonnancement

Les compilateurs vectoriseurs ou paralléliseurs recourent souvent à des transformations de code afin de faire apparaître du parallélisme, améliorer la localité mémoire ou tirer parti de la mémoire cache. Il est important de souligner qu'une transformation ne peut être appliquée que si les relations de dépendance sont conservées. Il existe de nombreuses transformations possibles [127, 133] ainsi que plusieurs tests permettant de décider de la légalité d'une modification du code (voir [7, 85, 52, 73, 102] entre autres). Toutefois, le choix des transformations et l'ordre dans lequel les appliquer afin de maximiser, par exemple, la localité mémoire, est inconnu.

L'algorithme de Kennedy et Allen [2] est un exemple de méthode se limitant à l'application systématique de seulement deux transformations. L'idée est de couper les nids de boucles en autant de nids que de composantes strictement connexes dans le graphe de dépendances et ensuite de rendre chacune des boucles parallèles si possible. Lamport [78] a proposé une autre approche, la méthode hyperplane, adaptée aux nids à dépendances uniformes. Les opérations sont regroupées en front parallèles, définis par des hyperplans. Le code transformé est alors de la forme :

```
do t = 0 to L
 doall p in H(t)
  ...
 endoall
 synchronize
enddo
```

où la boucle la plus externe est séquentielle et les autres sont parallèles. L'expression des hyperplans `H(t)` est obtenue par transformation matricielle de l'espace d'itération initial et des vecteurs de dépendances. Cette méthode a été généralisée de deux manières aux dépendances affines en fonction des compte-tours : l'une consiste à transformer chaque nid étape par étape. L'algorithme proposé par Wolf et Lam [126] utilise pour cela une représentation des dépendances par distance et direction. La deuxième façon consiste à trouver une fonction affine d'ordonnancement, affectant à chaque opération une date symbolique d'exécution calculée en résolvant un système de contraintes affines (parmi lesquelles

figurent les dépendances). L'algorithme décrit par Feautrier [53, 54] requiert le calcul du GFD et fournit un résultat plus précis que les autres méthodes.

Les techniques présentées se ramènent à un calcul d'ordonnancement (linéaire dans le cas de Lamport, affine pour l'algorithme de Wolf et Lam ainsi que pour Feautrier). Nous n'aborderons pas dans cette thèse les techniques utilisées pour faire apparaître une meilleure localité des données, et le lecteur se référera à la vaste littérature sur le sujet ([75, 126, 53, 38, 41, 22] entre autres).

Expansion mémoire

Les dépendances de sortie et anti-dépendances ne caractérisent pas un algorithme mais seulement l'une de ses implémentations. Plusieurs expériences [83, 17, 99] sur les programmes du Perfect Club [14] ont montré que l'élimination de ces dépendances permettait d'obtenir du parallélisme. La privatisation des tableaux et leur expansion sont deux techniques utilisées à cette fin.

La privatisation réplique sur chacun des processeurs les variables utilisées pour stocker des valeurs temporaires. Cela permet de réduire le nombre de synchronisations et de communications tout en faisant apparaître du parallélisme. Un élément de tableaux est dit privatisable dans une boucle si aucune lecture de cet élément ne lit une valeur définie par une itération précédente. Il est cependant parfois nécessaire de recopier ou de ne pas privatiser certains éléments de tableaux lorsque leurs valeurs sont visibles à l'extérieur de la boucle.

L'expansion de tableau élimine les dépendances qui ne sont pas directes en augmentant l'espace alloué pour le tableau, soit par le biais d'une réindexation soit en ajoutant des dimensions. La conservation du flot des données exige en général la connaissance du GFD exact. L'expansion totale consiste à étendre toutes les structures afin que chaque élément ne soit défini au plus qu'une fois (affectation unique, ou AU). Cette méthode est coûteuse en termes de mémoire et ne s'applique, à la compilation, que sur des programmes à contrôle statique. Toutefois l'expansion permet de trouver sur ces programmes plus de parallélisme que la privatisation. Enfin, la réduction de l'espace alloué par un programme mis en AU a été traité dans le domaine du systolique [26, 125, 60, 25], et récemment, Lefebvre et Feautrier [80, 81] ont proposé une méthode permettant de reconstruire des structures de données compatibles avec un ordonnancement, basée uniquement sur le GFD et l'ordonnancement.

Détection des récurrences

La détection des récurrences rend possible des optimisations poussées du code [17]. La plupart de ces analyses sont basées soit sur un modèle d'interprétation abstraite [33, 66, 67], soit sur une reconnaissance de motifs [129, 124]. Redon [111, 110] a décrit une méthode utilisant le GFD pour détecter des récurrences mettant en jeu n'importe quel opérateur associatif et capable d'intégrer certaines informations trouvées par les précédentes techniques. Barreteau [9] a

traité le problème du placement de ces récurrences sur les machines parallèles.

II.4 Conclusion

L'analyse des dépendances ne se résume pas à tester si deux opérations accèdent à la même cellule mémoire. Pertersen et Padua [98] ont montré la limite de ces tests et de nombreuses études révèlent qu'une connaissance plus approfondie du flot des données est nécessaire pour faire apparaître un plus grand degré de parallélisme. Dans cette optique et en dépit de leurs coûts et de l'ensemble réduit de programmes sur lesquels elles s'appliquent, les analyses de dépendances directes basées sur une abstraction polyédrique permettent l'application de techniques de parallélisation qui sont parmi les plus précises.

III Analyse exacte du flot des données

Cette section présente les techniques de calcul exact des dépendances du flot de données. Pour chaque opération lisant une cellule mémoire, une telle analyse trouve la dernière écriture précédant l'écriture qui a accédé à cette cellule. Le formalisme et l'approche utilisés par la méthode polyédrique sont au coeur de la plupart des analyses proposées dans cette thèse. Cette section résume le chapitre 3.

III.1 Modèle de programmes

L'analyse exacte du flot est limitée aux programmes vérifiant les conditions suivantes, appelés *programmes à contrôle statique* :

- les seules structures de données sont des tableaux ou scalaires de type de base (entiers, réels, etc.);
- les seules structures de contrôle sont la séquence, le `if..then..else` et la boucle `do`;
- les bornes des boucles, les prédicats des conditionnelles et les indices des tableaux sont des fonctions quasi-affines des compte-tours et des paramètres de structure (c'est-à-dire des fonctions affines de termes pouvant utiliser des modulos ou des divisions entières sur les compte-tours et les paramètres de structure);
- les instructions de base sont les affectations de variables;
- l'utilisation de pointeurs ou d'alias est proscrite.

III.2 Méthode polyédrique

La méthode polyédrique présentée par Feautrier [49] consiste à déterminer pour chaque opération $\langle \mathtt{R}, y\rangle$ lisant un élément $\mathtt{A}(g(y))$ du tableau A l'instance d'une des instructions $\mathtt{S}_1$,...,$\mathtt{S}_n$ écrivant A, qui a écrit la valeur lue par $\langle \mathtt{R}, y\rangle$. Cette source est calculée en fonction de y.

Supposons que $\mathtt{S}_i$ soit de la forme $\mathtt{A}(f_i(x_i)) = \ldots$ avec x_i le vecteur d'itération de $\mathtt{S}_i$. La première étape recense les vecteurs x_i correspondant à des opérations $\langle \mathtt{S}_i, x_i\rangle$ qui écrivent $\mathtt{A}(g(y))$. Les candidats sources $\langle \mathtt{S}_i, x\rangle$ doivent vérifier plusieurs contraintes sur x, et les vecteurs d'itération x qui conviennent forment un ensemble $\mathsf{Q}_i(y)$ défini par les conditions suivantes :

Prédicat d'existence $\langle \mathtt{S}_i, x\rangle$ est une opération valide : $x \in \mathsf{I}(\mathtt{S}_i)$;

Équation d'indices $\langle \mathtt{S}_i, x\rangle$ et $\langle \mathtt{R}, y\rangle$ accèdent à la même cellule mémoire : $f_i(x) = g(y)$;

Prédicat de séquencement $\langle \mathtt{S}_i, x\rangle$ est exécutée avant $\langle \mathtt{R}, y\rangle$: $\langle \mathtt{S}_i, x\rangle \prec \langle \mathtt{R}, y\rangle$;

Environnement la source est calculée sous l'hypothèse que $\langle \mathtt{R}, y\rangle$ est une opération valide : $y \in \mathsf{I}(\mathtt{R})$.

Pour résumer, $\mathsf{Q}_i(y)$ est défini par :

$$\mathsf{Q}_i(y) = \{x \,|x \in \mathsf{I}(\mathtt{S}_i), f_i(x) = g(y), \langle \mathtt{S}_i, x\rangle \prec \langle \mathtt{R}, y\rangle\}.$$

L'opération produisant la valeur $\mathtt{A}(g(y))$ lue par $\langle \mathtt{R}, y\rangle$ est la dernière opération écrivant dans cette cellule mémoire. Par conséquent, la fonction source est :

$$\sigma(y) = \max \bigcup_{i=1}^{n} \{\langle \mathtt{S}_i, x\rangle \,|x \in \mathsf{Q}_i(y)\}.$$

Lorsque le programme est à contrôle statique, l'ensemble $\mathsf{Q}_i(y)$ est une union de polyèdres $\mathsf{Q}_i^p(y)$. En effet, l'ordre séquentiel $\prec$ est défini par la disjonction :

$$\langle \mathtt{S}_1, x_1\rangle \prec \langle \mathtt{S}_2, x_2\rangle \iff \bigvee_{p=0}^{d_{\mathtt{S}_1\mathtt{S}_2}} \langle \mathtt{S}_1, x_1\rangle \prec_p \langle \mathtt{S}_2, x_2\rangle$$

où

$$0 \le p < d_{\mathtt{S}_1\mathtt{S}_2} : \langle \mathtt{S}_1, x_1\rangle \prec_p \langle \mathtt{S}_2, x_2\rangle \iff (x_1[1..p] = x_2[1..p]) \wedge (x_2[p+1] < x_2[p+1]),$$

$$p = d_{\mathtt{S}_1\mathtt{S}_2} : \langle \mathtt{S}_1, x_1\rangle \prec_p \langle \mathtt{S}_2, x_2\rangle \iff x_1[1..d_{\mathtt{S}_1\mathtt{S}_2}] = x_2[1..d_{\mathtt{S}_1\mathtt{S}_2}] \wedge \mathtt{S}_1 \lhd \mathtt{S}_2.$$

Les ensembles $\mathsf{Q}_i^p(y)$ sont définis par :

$$\mathsf{Q}_i^p(y) = \{x \,|x \in \mathsf{I}(\mathtt{S}_i), f_i(x) = g(y), \langle \mathtt{S}_i, x\rangle \prec_p \langle \mathtt{R}, y\rangle\}.$$

L'expression de la source peut se ramener alors à l'expression :

$$\sigma(y) = \max_{1 \le i \le n} \max_{0 \le p \le d_{\mathtt{S}_i \mathtt{R}}} \left\langle \mathtt{S}_i, \max \mathsf{Q}_i^p(y) \right\rangle .$$

Le maximum de $\mathsf{Q}_i^p(y)$ est calculable en utilisant des outils de programmation linéaire en nombres entiers et peut s'exprimer à l'aide d'un *quast* en fonction de y.

Définition 1 (Quast) *Un* quast *est un arbre de sélection dont les prédicats sont des inégalités quasi-affines en fonction de compteurs de boucles et des paramètres de structures et dont les feuilles sont les opérations sources ou* $-$*, correspondant à une variable non initialisée.*

L'expression de la source est ensuite obtenue à partir des quasts pour chacun des $\mathtt{S}_i$ par des règles de combinaison de quasts. Le lecteur se référera à la partie 3.2.2, page 73, de cette thèse pour les détails de ce calcul ainsi que pour les optimisations possibles.

III.3 Algorithme polynômial simplifié

La méthode polyédrique traite les programmes à contrôle statique mais ne tient pas compte des spécificités des programmes réels. De plus, elle est d'un coût élevé en raison de l'algorithme de calcul du maximum entier d'un ensemble et de la combinaison finale entre les quasts. Des techniques efficaces s'appliquant à un sous-ensemble des programmes à contrôle statique ont été proposées [93, 70], et nous présentons ci-dessous une nouvelle méthode d'une meilleure complexité.

Principe de l'algorithme

Les algorithmes utilisés pour calculer le maximum entier d'un ensemble sont en général dérivés de l'algorithme du simplexe ou de la méthode d'élimination de Fourier-Motzkin [116]. Pugh et Wonnacott [104] ont montré que cette dernière était particulièrement bien adaptée au type de contraintes apparaissant dans la définition des dépendances de flot de données et ne conduit que rarement, sur des exemples réels, à une explosion de la taille du résultat.

La méthode décrite dans cette section est basée sur l'élimination de Fourier-Motzkin pour calculer le maximum de $\mathsf{Q}_i^p(y)$. Le but est d'obtenir une complexité très faible pour le calcul de la source, même dans le pire des cas. Elle se décompose en deux étapes :

1. élimination des équations affines par élimination de Gauss, en introduisant éventuellement des équations modulos pour tenir compte des contraintes de divisibilité;

2. élimination des compte-tours définissant $\mathsf{Q}_i^p(y)$ par ordre inverse de profondeur. De façon générale, un compteur de boucle x est contraint par un

système d'inégalités du type $l \leq dx \leq u$ ainsi qu'éventuellement par une équation modulo $x = c \bmod a$, avec d, a des entiers, l, u et c fonctions des compteurs des boucles englobantes. On peut alors montrer qu'une valeur entière de x existe ssi :

$$l - dc \leq da \left\lfloor \frac{u - dc}{da} \right\rfloor, \text{ ou : } da \left\lceil \frac{l - dc}{da} \right\rceil \leq u - dc.$$

L'élimination peut ensuite se poursuivre sur les compte-tours des boucles extérieures si des restrictions suffisantes sont faites sur les formes des deux inégalités précédentes.

Complexité et domaine

La complexité de l'algorithme pour calculer les maxima de tous les $\mathsf{Q}_i^p(y)$, pour tout p est en $O(l^3)$ si l est le nombre de boucles englobant l'instruction $\mathtt{S}_i$ et la taille de ces maxima est en $O(l^3)$.

La méthode s'applique sur des programmes à contrôle statique tels que :

1. L'équation d'indices est équivalente à un système sur les compte-tours composé d'équations de la forme :

$$\begin{array}{rrcl} & x[i_k] & = & \sum_{j<i_k} b_{kj} x[j] + c_{i_k}, \\ \text{ou} & a_k x[i_k] & = & b_{kj} x[j] + c_{i_k}, \text{ pour } j < i_k, \\ \text{ou} & 0 & = & c_k, \end{array}$$

avec b_{kj}, a_k et c_k des entiers.

2. Lors de l'élimination d'un compteur, celui-ci n'a qu'une borne inférieure ou qu'une borne supérieure. Si cette borne n'est pas constante alors il n'y a pas de contrainte de divisibilité portant sur ce compteur.

3. Les bornes supérieures d'un compteur ne diffèrent que d'une constante.

Ces conditions sont vérifiées en cours de calcul. On revient à la méthode polyédrique dès que l'on sort du domaine d'application de l'algorithme.

Enfin, il faut noter que cet algorithme est capable de traiter certains cas où les coefficients des compteurs de boucles ne sont pas des constantes entières. En effet, toutes les calculs peuvent être menés de façon symbolique si l'on connaît le signe des coefficients des compteurs. Des coefficients paramétriques apparaissent dans les situations suivantes :

- pour des indices de tableaux linéarisés de façon rectangulaire;
- pour des calculs par blocs, comme par exemple le produit matrice vecteur par blocs;

Programme	Utilisant PIP		Méthode simplifié
	Nb. d'appels	Nb. de pivots	Nb. de pivots
`ffc`	4	7170	513
`matmul`	4	4222	229
`chales`	18	10882	717
`choles`	12	9217	614
`burg2`	26	20332	2698
`mod1`	2	1826	382
`mod3`	2	520	13
`van4`	2	5376	588
`yacobi`	105	594897	79649
`gosser`	13	10932	713
`lczos`	60	32773	2554
`relax`	6	25766	2931
`across`	5	468	64

TAB. 1 - *Comparaison du nombre de pivots*

- lors du remplacement d'une variable d'induction couvrant plusieurs boucles.

Toutefois, la méthode que nous venons de présenter doit s'appliquer avec précaution pour l'analyse de programmes avec des coefficients paramétriques. En effet, la méthode polyédrique ne peut plus servir de technique de secours lorsque la méthode simplifiée échoue.

Performances

L'algorithme polynômial a été implémenté en `Le_Lisp` dans le cadre du projet PAF [97]. Lorsque la méthode échoue, le calcul du maximum est confié à PIP [50], écrit en C et qui utilise le simplexe.

La méthode est efficace sur de nombreux programmes. Tous les programmes de la table 1 sont ainsi traités sans faire appel à PIP. La comparaison par rapport à la méthode polyédrique ne se fait pas en termes de temps de calcul car les performances des langages sont trop différentes. On a choisi plutôt de comparer le nombre total de pivots pour les deux techniques. La table 1 donne pour chaque programme le nombre d'appels à PIP pour la méthode polyédrique ainsi que le nombre de pivots utilisés par les deux techniques. La méthode polynômiale permet de gagner un facteur 5 à 40 en termes de nombre de pivots par rapport à la méthode polyédrique.

III.4 Autres travaux

La plupart des analyses du flot des données sont approchées. Elles seront détaillés en section IV.8. Les premières analyses exactes sont dues à Brandes [23] et Feautrier [49]. Les principales limitations de ces analyses sont leurs coûts et leurs domaines d'application restreints. On s'intéresse aux techniques proposées depuis pour pallier l'une ou l'autre de ces limitations.

Pugh et Wonnacott [103, 129] Leur approche pour une analyse exacte du flot des données est basée sur un formalisme similaire à celui de Brandes. Leur méthode s'applique à n'importe quel programme à contrôle statique et construit une relation entre les vecteurs d'itérations d'opérations en dépendance. Ces relations sont définies par des formules de Presburger, c'est-à-dire par des formules logiques dont les prédicats sont des inégalités sur des variables entières. Ainsi, en utilisant les mêmes notations que dans les sections précédentes, la relation de dépendance entre les instructions $\mathtt{S}_i$ et $\mathtt{R}$ est :

$$\left\{ x_i \to y \,\middle|\, \begin{array}{l} x_i \in \mathsf{I}(\mathtt{S}_i) \wedge y \in \mathsf{I}(\mathtt{R}) \wedge \langle \mathtt{S}_i, x_i \rangle \prec \langle \mathtt{R}, y \rangle \wedge f_i(x_i) = g(y) \\ \bigwedge_j \neg(\exists x_j, x_j \in \mathsf{I}(\mathtt{S}_j) \wedge \langle \mathtt{S}_i, x_i \rangle \prec \langle \mathtt{S}_j, x_j \rangle \prec \langle \mathtt{R}, y \rangle \wedge f_j(x_j) = g(y)). \end{array} \right\}.$$

Le pouvoir d'expression de ces relations est identique à celui des quasts. En effet, étant donné un ensemble A:

$$\{\max \mathsf{A}\} = \left\{ x \,\middle|\, x \in \mathsf{A} \wedge \neg(\exists x', x' \in \mathsf{A} \wedge x \ll x') \right\}.$$

En revanche, les deux méthodes diffèrent par la complexité de leur représentation : une source représentée par un quast regroupe en une seule expression l'effet de toutes les écritures et les conditions pour lesquelles elles sont sources, alors que les relations de dépendances définissent pour chaque écriture, séparément les unes des autres les conditions pour lesquelles elles sont sources. Ainsi, la relation de dépendance entre $\mathtt{S}_i$ et $\mathtt{R}$ est définie par les prédicats du quast qui sont sur les chemins qui vont de la racine aux instances de $\mathtt{S}_i$. Lorsque de nombreuses conditions pour être source sont communes à des instances de différentes instructions, alors les quasts offrent une représentation plus concise de la source. À l'inverse, lorsque les conditions sont totalement distinctes d'une instruction à l'autre, les relations de dépendances sont plus concises que les quasts.

Enfin, on peut montrer que la méthode simplifiée présentée en III.3 a une meilleure complexité ($O(l^3)$) que la technique proposée par Pugh et Wonnacott ($O(l^4)$), sur le domaine d'application de la méthode simplifiée, avec l le nombre de boucles entourant la source.

Maslov [90] L'approche proposée par Maslov est similaire à la méthode polyédrique, mais il utilise une représentation de relation de dépendance au lieu de quasts. Il effectue un calcul de maximum sur les relations à l'aide de l'élimination de Fourier-Motzkin. Les contraintes de divisibilité introduisent de nouvelles

variables qui restent dans le résultat final en tant que paramètres. Cette méthode permet de traiter plus de cas que la méthode simplifiée proposée en III.3 mais donne un résultat moins précis que la méthode polyédrique (les paramètres ne sont pas éliminés). La complexité de l'algorithme de Maslov dépasse celle de la méthode simplifiée ($O(l^4)$), avec l le nombre de boucles entourant la source.

Maydan *et al.* [93] Maydan, Amarasinghe et Lam ont proposé une méthode ne s'appliquant qu'à un sous-ensemble de programmes à contrôle statique. Les programmes traités doivent vérifier les conditions suivantes :

- une fonction d'indice d'un tableau écrit doit être une bijection en les compte-tours qu'elle utilise.
- la borne supérieure d'une boucle doit soit être toujours plus grande que la borne inférieure, soit toujours plus petite.

Ce domaine est strictement inclus dans le domaine de notre méthode simplifiée. Comme les conditions d'existence des opérations sont triviales sur ce domaine, leur algorithme consiste principalement à calculer la dernière opération d'écriture vérifiant l'équation d'indices. Le résultat est un quast, et est appelé *Last Write Tree* (LWT). En termes de complexité, la construction d'un LWT demande $O(l^3)$ opérations alors que sur ce domaine, la méthode simplifiée prend $O(l^2)$ opérations, où l est le nombre de boucles entourant la source.

Heckler et Thiele [70] La méthode proposée par Heckler et Thiele ne s'applique également qu'à un sous-ensemble des programmes à contrôle statique. Le principe de leur algorithme est de calculer les solutions du système d'équations affines issu de l'équation d'indices. L'ensemble des solutions est décrit par des paramètres qui sont ensuite calculés afin de vérifier les autres contraintes de dépendance. Des conditions fortes sont imposées sur la forme de la solution paramétrique afin que les valeurs des paramètres puissent être calculées directement, sans passer par une élimination de Fourier-Motzkin. Lorsque ces conditions ne sont pas vérifiées, ils recourent à la méthode polyédrique. La complexité de leur algorithme est celle du calcul d'une forme normale de l'Hermite.

Au delà des programmes à contrôle statique

Plusieurs techniques, dont la propagation des constantes, le remplacement des variables d'induction ou la restructuration des `while`, permettent d'étendre le domaine de l'analyse exacte de dépendances de flot de données pour tableaux à des programmes qui ne sont pas à contrôle statique. Berthou [15] a en outre suggéré de limiter les fragments de code à analyser dans des programmes réels, sans perdre de précision.

En section III.3, nous avons proposé une extension permettant de traiter quelques cas d'indices de tableaux non-affines. Maslov [89] a suggéré d'utiliser un simple test pour déterminer si un indice provenait d'une linéarisation,

lorsque le domaine des boucles est un pavé. Le problème peut dans ce cas être traité comme si le programme était à contrôle statique. Maslov et Pugh [91] ont décrit une méthode plus générale pour traiter certains indices polynômiaux. L'idée consiste à remplacer la fonction polynômiale dans une inégalité par la conjonction des fonctions tangentes, en tous les points entiers du domaine (fini) d'itération. Maslov [90] a également proposé une technique permettant de traiter les variables dont la valeur ne change pas entre les opérations en dépendance comme des paramètres du problème. Enfin Leservot [82] a étendu l'analyse exacte à des *instructions généralisées*, qui peuvent être des fragments de code ou des appels de procédures.

III.5 Conclusion

L'analyse exacte de dépendance de flot de donnée pour tableaux est une technique puissante qui calcule de façon exacte les dépendances directes des programmes à contrôle statique. Les principaux défauts de cette analyse sont son coût et son domaine d'application réduit. Plusieurs solutions ont été proposées pour diminuer le coût en réduisant encore le domaine d'application aux programmes les plus courants. Quant aux limitations sur les programmes à analyser, l'hypothèse intraprocédurale a été partiellement levée et certaines contraintes non-affines sont traitées lorsqu'elles peuvent être soit ignorées soit transformées de façon équivalente en contraintes affines.

IV Analyse approchée de flot

Quand des contraintes non-affines apparaissent dans l'expression de la source, la méthode présentée dans la section précédente ne marche plus. Nous présentons dans cette section un résumé du chapitre 4 sur une analyse de flot des données approchée.

IV.1 Modèle de programmes

On étudiera les programmes vérifiant les contraintes suivantes :

1. les seules structures de données sont des tableaux ou scalaires de type de base (entiers, réels, etc.);
2. les seules structures de contrôle sont la séquence, la boucle `do`, la boucle `while`[1] et le `if..then..else`. Les `goto` et les appels de procédures sont interdits;
3. les instructions de base sont des affectations de variables;
4. l'utilisation de pointeurs ou d'alias est proscrite.

1. De façon similaire aux boucles `do`, les boucles `while` sont supposée posséder une variable d'induction.

IV.2 Quelle approximation ?

Nous avons vu en section III.3 que des cas particuliers de programmes non statiques pouvaient être analysés exactement. Toutefois, dès que l'on considère des programmes avec des prédicats quelconques de conditionnelles, des bornes de boucles quelconques et des indices de tableaux non-affines, même l'existence d'une dépendance devient un problème indécidable.

Deux causes expliquent l'échec de l'analyse exacte :

Un contrôle dynamique La plupart des programmes réels mettent en jeu en fait des conditionnelles et des `while` dont les prédicats sont non-affines en les compte-tours. Le programme de l'exemple E1 illustre le cas d'une

```
    do x1 = 1 to n
S1   s = ..
     do x2 = 1 while (P(x1,x2))
S2    s = .. s ..
     enddo
R    .. = .. s ..
    enddo
```

Exemple E1

```
    do x = 1, n
     if (P(x)) then
S1    s = ..
     else
S2    s = ..
     endif
    enddo
R   if (n>0) then .. = .. s ..
```

Exemple E2

boucle `while` : la valeur lue par `s` à l'instruction S_2 vient soit de l'écriture faite l'itération précédente du `while`, soit de l'instance de l'instruction S_1 pour la même itération de la boucle externe. L'exemple E2 illustre le cas d'une conditionnelle avec un prédicat inconnu. Toutefois, la sémantique du `if..then..else` garantit que la valeur de `s` vient de la dernière instance de l'instruction S_1 ou S_2.

Des indices non-affines Selon Shen *et al.* [117], 34% des programmes réels comportent des indices de tableaux non-affines en les compte-tours. La plupart de ces indices sont affines en des variables autres que des compteurs de boucles. Ainsi, dans l'exemple E3, les deux écritures utilisent des indices de tableaux non-affines. De même, les indices du tableau `A` dans l'exemple E4 qui est tiré de [124] sont non-affines. Toutefois, dans ce cas, cela est équivalent à une boucle `do` dont les bornes seraient non-affines.

Une façon d'analyser un programme comportant des termes non-affines est de l'instrumenter pour en déduire le GFD exact [79, 94, 98]. Toutefois, le graphe ainsi construit dépend des données initiales du programme et ne peut être généralisé. Une partie de l'analyse peut être également reportée à l'exécution, soit en produisant une version séquentielle et une version parallèle du code et en choisissant laquelle exécuter en testant l'existence de dépendances [98], soit en spéculant sur les dépendances et en exécutant une version parallèle, quitte à revenir ultérieurement à une version séquentielle [108, 109]. On adoptera *a priori*

```
S1  ii = ..
    do x = 1, n
S2   A(ii) = ..
S3  ii = ..
S4   A(ii) = ..
    end do
    do x = 1, n
R    .. = .. A(x) ..
    enddo
```

Exemple E3

```
    if (P) then
S1    jlow = 2, jup = jmax -1
    else
S2    jlow = 1, jup = jmax
    endif
    do x = 1, n
S3   A(jlow : jup) = ..
     if (P) then
R     .. = A(2 : jmax-1)
     endif
    enddo
```

Exemple E4

une autre approche, où toute l'analyse est faite statiquement. On montrera toutefois que cela n'est pas en contradiction avec les précédentes techniques.

Les analyses traditionnelles de flot de données recourent à des analyses complémentaires sur les contraintes non-affines afin d'affiner leurs résultats. Différentes études [48, 18] ont montré l'importance de telles analyses dans la détection du parallélisme. Ainsi, dans l'exemple E3, on remarque que la valeur de `ii` en S_4 est la même que la valeur de `ii` en S_2 l'itération suivante. Trouver par une analyse complémentaire cette information nous permettrait de conclure que toutes les écritures de S_4 sont recouvertes par celles de S_2, sauf la dernière. En ce qui concerne l'exemple E4, l'analyse de Tu et Padua trouve les valeurs de `jlow` et `jup` en fonction de `jmax`, ce qui leur permet de conclure que le tableau `A` est privatisable. Nous montrerons que cette information nous permet également de calculer un GFD exact.

L'analyse approchée que l'on cherche vérifiera les critères suivants :

1. elle est conservative : l'approximation des dépendances de flot est une sur-approximation;

2. elle est exacte sur les fragments à contrôle statique;

3. elle peut utiliser l'information trouvée par des analyses complémentaires sur les contraintes non-affines. Intuitivement, la quantité d'information sur les contraintes non-affines donne une estimation de la qualité du résultat de l'analyse. On verra en section IV.5 que cela est malheureusement faux en général : certaines informations peuvent ne servir à rien pour l'analyse; d'autre part, il n'est pas toujours possible de tenir compte de toute l'information utile.

Collard [29, 30] a proposé une analyse approchée vérifiant les critères 1 et 2 pour les `while` et les conditionnelles, et nous proposons une généralisation de son approche, introduisant par là même le critère 3.

IV.3 Présentation du formalisme

Contraintes non-affines

Numérotons chaque contrainte non-affine du programme. Étant donné une contrainte non-affine c_h, on note $\mathtt{T}_h$ l'instruction dans laquelle elle apparaît. Cette instruction est soit une conditionnelle (branche «vrai» ou branche «faux»), soit une boucle si ces bornes sont non-affines, soit une affectation à un tableau avec des indices non-affines. La valeur de c_h dépend des compte-tours englobant $\mathtt{T}_h$, éventuellement du vecteur d'itération de la lecture dont on cherche la source si c_h est l'équation d'indices, et si $\mathtt{T}_h$ est un `while`, c_h dépend alors aussi du compte-tours du `while`. Par conséquent, si $x \in \mathsf{Q}_k(y)$ où $\mathsf{Q}_k(y)$ est l'ensemble des vecteurs d'itération des candidats sources de l'instruction $\mathtt{S}_k$, alors $c_h(x[1..e_h], y) = \text{vrai}$, avec $e_h = d_{\mathtt{T}_h}$ ou $d_{\mathtt{T}_h} + 1$ si $\mathtt{T}_h$ est un `while`. Suivant la précision que l'on cherche à atteindre, la contrainte non-affine c_h peut s'écrire soit comme un prédicat, soit comme une inégalité :

$$c_h(x, y) = \left(a_h(x, y) + \sum_k n_k(x, y) \geq 0 \right),$$

avec a_h une forme affine et n_k des fonctions non-affines. Il faut noter que les n_k peuvent être des variables qui ne sont pas des compte-tours.

Définition 2 (Domaine de paramètres) *Soient C_k l'ensemble des indices des contraintes non-affines apparaissant dans la définition de $\mathsf{Q}_k(y)$ et $m_k = \max_{h \in \mathsf{C}_k} e_h$. L'ensemble :*

$$\mathsf{D}_k(y) = \left\{ x \,\middle|\, x \in \mathbb{Z}^{m_k}, \bigwedge_{h \in \mathsf{C}_k} c_h(x[1..e_h], y) \right\},$$

est l'ensemble de vecteurs d'itération de dimension m_k pour lesquels toutes les contraintes indexées par C_k sont vraies. Cet ensemble est appelé domaine de paramètres *associé à la dépendance entre une instance de $\mathtt{S}_k$ et $\langle \mathtt{R}, y \rangle$.*

Paramétrage de la source

On rappelle la définition de la source :

$$\sigma(y) = \max_{1 \leq i \leq n} \max_{0 \leq p \leq d_{\mathtt{S}_k \mathtt{R}}} \left\langle \mathtt{S}_k, \max \mathsf{Q}_k^p(y) \right\rangle.$$

L'idée du paramétrage de la source est simple : intuitivement, plus on retardera le moment de l'approximation dans le calcul, plus on a de chances qu'elle soit meilleure. On va donc introduire des paramètres représentant les contraintes non-affines dans l'expression de la source, afin de pouvoir calculer l'expression de la source en fonction de ces paramètres puis faire l'approximation. On verra par la suite que l'introduction de paramètres a également d'autres avantages.

L'ensemble $\mathsf{Q}_k^p(y)$ peut clairement s'écrire comme l'intersection de l'ensemble des vecteurs d'itération vérifiant les contraintes affines de $\mathsf{Q}_k^p(y)$, $\mathsf{L}_k^p(y)$, et de l'ensemble des vecteurs d'itération vérifiant les contraintes non-affines de $\mathsf{Q}_k^p(y)$:

$$\mathsf{Q}_k^p(y) = \mathsf{L}_k^p(y) \cap \{z \,|z[1..m_k] \in \mathsf{D}_k(y)\}\,.$$

Posons maintenant $\hat{\mathsf{Q}}_k^p(x,y) = \mathsf{L}_k^p(y) \cap \{z \,|z[1..m_k] = x\}$. On a partitionné $\mathsf{Q}_k^p(y)$ en autant de sous-ensembles que d'éléments de $\mathsf{D}_k(y)$: $\mathsf{Q}_k^p(y) = \bigcup_{x \in \mathsf{D}_k(y)} \hat{\mathsf{Q}}_k^p(x,y)$.

Par conséquent,

$$\max \mathsf{Q}_k^p(y) = \max_{x \in \mathsf{D}_k(y)} \max \hat{\mathsf{Q}}_k^p(x,y). \tag{1}$$

La valeur du maximum est atteinte pour un x de $\mathsf{D}_k(y)$.

Définition 3 (Paramètre du maximum) *Le vecteur pour lequel le maximum de* (1) *est atteint est appelé* paramètre du maximum de $\mathsf{D}_k(y)$ pour S_k à la profondeur p *et est noté* $\alpha_k^p(y)$. *Si le maximum n'existe pas, on donne à* $\alpha_k^p(y)$ *la valeur* $-$. *De plus, par définition de* $\hat{\mathsf{Q}}_k^p(x,y)$,

$$\alpha_k^p(y) = \max \mathsf{L}_k^p(y)_{|m_k} \cap \mathsf{D}_k(y).$$

L'ensemble $\mathsf{L}_k^p(y)_{|m_k}$ est la projection de $\mathsf{L}_k^p(y)$ sur $\mathbb{Z}^{m_k}$. L'expression paramétrée de la source est alors :

$$\sigma(y) = \max_{1 \le k \le n} \max_{0 \le p \le d_{\mathsf{S}_k \mathsf{R}}} \left\langle \mathsf{S}_k, \max \hat{\mathsf{Q}}_k^p(\alpha_k^p(y), y) \right\rangle. \tag{2}$$

Remarquons que cette expression est calculable en fonction de $\alpha_k^p(y)$ avec les mêmes techniques que celles utilisées dans le cas exact. De plus, lorsqu'il n'y a pas de contrainte non-affine dans la définition de la source, alors $\alpha_k^p(y) = \max \mathsf{L}_k^p(y)_{|m_k}$ et la source est exacte. Ainsi, l'analyse exacte de la section III est bien un cas particulier de la FADA.

Flou du résultat

Aucune approximation n'a été faite lors du paramétrage de la source. Les paramètres du maximum ne peuvent pas être calculés mais l'expression de la source (2) peut être utilisée telle quelle à l'exécution pour déterminer dynamiquement la source exacte d'une variable, si les paramètres du maximum sont calculés et mis à jour dynamiquement également. On n'envisagera pas cette utilisation possible de la source par la suite.

Une approximation très simple consiste à donner aux paramètres toutes les valeurs possibles et de considérer l'ensemble des sources décrit de cette façon comme l'*ensemble des sources possibles*. Une telle approximation est clairement conservative, mais le résultat de l'analyse est alors très *flou*.

Pour améliorer cette approximation, on peut chercher des propriétés P sur les contraintes non-affines. De ces propriétés, on va déduire des propriétés $\hat{\mathsf{P}}$ sur les paramètres du maximum, puis on va calculer l'ensemble des sources possibles en donnant aux paramètres toutes les valeurs permises par les propriétés $\hat{\mathsf{P}}$. On va donc procéder en trois étapes :

1. Des propriétés P sont trouvées par une analyse des contraintes non-affines du programme. Plus la description de ces contraintes est précise, plus l'ensemble décrit par P est petit. La figure 1 représente le cas où une

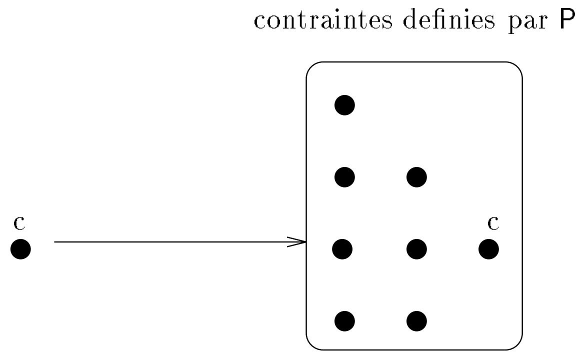


FIG. 1 - *Approximation d'une contrainte.*

   seule contrainte c intervient dans l'expression de la source. c vérifie les propriétés P, ce qui garantit que l'analyse est conservative.

2. Les propriétés $\hat{\mathsf{P}}$ sont dérivées de P. Les propriétés sur les paramètres du maximum sont des conséquences des propriétés P, afin de garantir que l'analyse est toujours conservative. Cette traduction de P en $\hat{\mathsf{P}}$ permet de ne retenir de P que les informations qui sont utiles pour le calcul de la source. Le but est de montrer que, comme sur la figure 2, chaque vecteur vérifiant $\hat{\mathsf{P}}$ est le paramètre du maximum d'une contrainte vérifiant P, c'est-à-dire que l'on n'a pas perdu de précision durant la traduction.

3. Enfin, on construit la source paramétrique et l'on considère toutes les valeurs possibles de cette source pour tous les paramètres vérifiant $\hat{\mathsf{P}}$:

$$\mathsf{S}(y) = \left\{ \max_{1 \leq k \leq m} \max_{0 \leq p \leq d_{\mathsf{S}_k \mathsf{R}}} \left\langle \mathsf{S}_k, \hat{\mathsf{Q}}_k^p(x_k^p, y) \right\rangle \middle| x_k^p \in \mathbb{Z}^{m_k}, \hat{\mathsf{P}}\left(\ldots, x_k^p, \ldots\right) \right\}.$$

   Cet ensemble est calculable si les propriétés $\hat{\mathsf{P}}$ sont affines. Comme le montre la figure 3, à chaque paramètre vérifiant $\hat{\mathsf{P}}$ correspond une source possible, parmi lesquelles se trouve la source exacte.

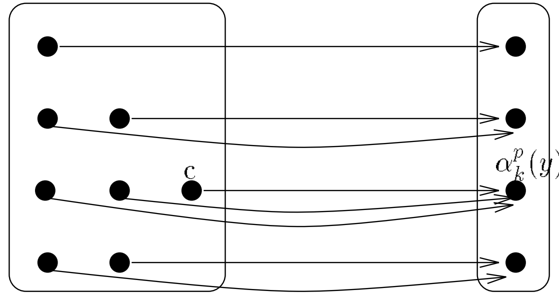


FIG. 2 - *Traduction de* P *en* $\hat{\mathsf{P}}$*.*

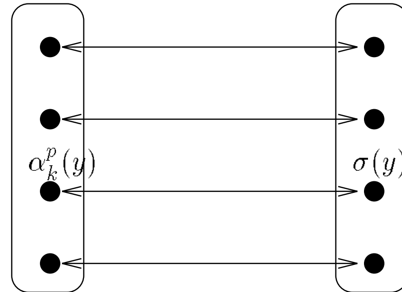


FIG. 3 - *Calcul de l'ensemble de sources.*

IV.4 Amélioration de l'approximation

Le but de cette section est de présenter des techniques permettant de trouver des relations entre contraintes non-affines. Ces relations, ou propriétés, forment l'ensemble P. Dans un premier temps, nous passons en revue les techniques existantes et en déduisons une forme générale de propriétés. Puis nous proposons deux autres analyses.

Forme générale de propriétés

Les techniques utilisées pour trouver des relations entre des expressions restreignent le type d'expressions étudiées et le type de relations trouvées, tout en faisant des approximations conservatives quand c'est nécessaire. On peut distinguer deux types d'analyses : celles qui cherchent des relations entre les variables ou expressions du programme, et celles qui analysent une seule expression afin d'en donner une caractérisation affine aussi précise que possible.

L'analyse proposée par Cousot et Halbwachs [33], dans un formalisme d'in-

terprétation abstraite trouve des relations entre les variables scalaires en examinant les expressions affines du programme. Ces relations sont représentées par des polyèdres. Cette analyse a été étendue pour trouver des relations sous forme de modulos [58, 87, 88] et pour pouvoir examiner des expressions polynômiales [19]. Un autre type d'analyse, basée sur la représentation SSA (mise en assignation unique statique) [37] détecte des relations d'égalité entre les variables. L'idée est de donner systématiquement un nom différent à des variables qui sont écrites par des instructions différentes. Une fonction, usuellement appelée ϕ, permet de déterminer dynamiquement à chaque lecture quelle est le nom correct de la variable à lire. Tu et Padua [122, 124] ont étendu la définition des fonctions ϕ en prenant en compte pour chaque instruction son prédicat d'existence et ont proposé une méthode pour mettre en relation des expressions affines de variables, éventuellement tableaux, en comparant les arguments des fonctions ϕ et les prédicats d'existence. Ces relations sont des inégalités. Enfin, les techniques classiques de détection de variable d'induction [1] ont été généralisées aux variables ou expressions dont l'incrément n'est pas constant [128, 67]. Ces méthodes calculent alors la valeur de l'expression en fonction des compteurs et en déduisent des propriétés de croissance ou de monotonie par exemple.

Le deuxième type d'analyse s'attache à donner des propriétés affines sur des contraintes non-affines. Tel est le cas de la méthode de Maslov et Pugh, présentée en section III.4, et de l'analyse proposée par Dumay [47]. Cette dernière s'appuie sur un ensemble de règles fournies par l'utilisateur, associant à chaque type de fonction non-affine (exponentielle, multiplication,...) une nouvelle variable et un ensemble de propriétés affines sur cette variable. La variable peut alors remplacer la fonction non-affine dans n'importe quelle expression. L'avantage est que l'ensemble de règles est complétable à volonté par l'utilisateur.

Toutes les analyses brièvement décrites ici produisent des relations sur des contraintes non-affines qui peuvent être décrites grâce à des formules logique du premier ordre, les contraintes non-affines étant représentées soit par des prédicats, soit par des inégalités affines composées avec des fonctions non-affines, comme proposé page IV.3.

Caractérisation des paramètres

L'une des premières propriétés dont on dispose est la définition même des paramètres du maximum. La relation $\alpha_k^p(y) = \max \mathsf{L}_k^p(y)_{|m_k} \cap \mathsf{D}_k(y)$ peut être réécrite sous forme de formule logique :

$$l_k^p(x, y) \bigwedge_{i \in \mathsf{C}_k} c_i(x[1..e_i], y) \Longrightarrow x \ll \alpha_k^p(y),$$

où $l_k^p(x, y)$ est la fonction caractéristique de $\mathsf{L}_k^p(y)_{|m_k}$. Cette formule traduit le fait que $\alpha_k^p(y)$ est une borne supérieure de l'ensemble dont il est le maximum,

et

$$(\alpha_k^p \neq -) \Longrightarrow l_k^p(\alpha_k^p(y), y) \bigwedge_{i \in \mathsf{C}_k} c_i(\alpha_k^p(y)[1..e_i], y)$$

signifie que si cet ensemble n'est pas vide, alors $\alpha_k^p(y)$ en est un élément.

Analyse structurelle

Un autre type de propriété se révèle peu coûteux à trouver: il s'agit des propriétés structurelles des boucles `while` et des conditionnelles `if..then..else`. Dans le premier cas, on sait que lorsqu'une itération d'un `while` est exécutée, toutes les itérations précédentes l'ont été aussi. Si c_i est le prédicat du `while` entouré de n autres boucles cela se formalise ainsi :

$$\forall x_1, x_2,\ x_1[1..n] = x_2[1..n] \wedge x_2[n+1] \leq x_1[n+1] \wedge c_i(x_1[1..e_i]) \\ \Longrightarrow c_i(x_2[1..e_i]).$$

Dans le cas de la conditionnelle, lorsque l'instruction est exécutée l'une et seulement l'une des deux branches de la conditionnelle est exécutée. Si c_i et c_j sont les deux contraintes générées par le prédicat non-affine d'un `if..then..else`, alors cette relation s'écrit :

$$\forall x,\ c_i(x[1..e_i]) \Longleftrightarrow \neg c_j(x[1..e_j]).$$

Enfin, une troisième propriété très simple vient s'ajouter aux deux autres. On suppose lors du calcul de la source que la lecture est exécutée. Par conséquent, lorsque la lecture vérifie la contrainte c_i, on ajoute la relation $c_i(y[1..e_i], y)$.

Analyse itérative

Le principe de cette analyse est le suivant : deux contraintes non-affines sont égales si elles sont la composition de la même fonction avec les mêmes variables et si ces variables ont la même source, et donc la même valeur. La condition est suffisante mais pas nécessaire. Cette méthode se base sur les résultat d'une analyse de flot pour trouver des relations d'égalité entre contraints non-affines, qui amélioreront le résultat d'une autre analyse, ce qui explique son nom.

Tester si les fonctions et les variables sont les mêmes est un critère syntaxique. Tester si les sources d'une variable sont les mêmes en deux points du programme se fait de la façon suivante : soient s (resp. s') la conjonction des prédicats se trouvant sur le chemin de la racine à une feuille $\langle \mathtt{S}, x\rangle$ (resp. $\langle \mathtt{S}', x'\rangle$) du quast source d'une variable `v` en un point du programme (resp. en un autre point). La variable prend la même valeur en deux points du programme si :

$$s \wedge s' \wedge (\mathtt{S} = \mathtt{S}') \wedge (x = x').$$

IV.5 Traduction des propriétés

Une fois trouvées des propriétés P sur les contraintes non-affines, elles doivent être traduites en propriétés $\hat{\mathsf{P}}$ sur les paramètres du maximum. Il est alors important de savoir si chaque paramètre vérifiant $\hat{\mathsf{P}}$ est un paramètre du maximum associé à des contraintes vérifiant P. On dira alors qu'*aucun flou n'a été ajouté* par la traduction. Lorsque c'est le cas, trouver des propriétés P améliore la précision de la source. Il est alors possible de décider s'il est intéressant de réaliser des analyses de plus en plus complexes pour trouver de nouvelles propriétés sur les contraintes non-affines afin de parvenir à la précision voulue de la source. Lorsque du flou est ajouté lors de la traduction, il n'y a plus de rétroaction entre les analyses complémentaires calculant P et la précision de la source.

L'idée est d'utiliser la méthode de résolution de Robinson [113] afin de transformer des formules avec des contraintes non-affines en des formules sans contraintes non-affines. Donnons tout d'abord quelques définitions. Les formules de P peuvent être mises sous forme normale conjonctives. Un *littéral* est une formule atomique ou la négation d'une formule atomique et une *clause* est la disjonction de littéraux. Un littéral non-affine est une contrainte non-affine ou sa négation. Les autres littéraux sont appelés *littéraux affines*.

La règle de base de la résolution est la suivante :

Règle 1 (résolution simple) *Étant données deux clauses $r(x) \vee c(f(x))$ et $s(x') \vee \neg c(g(x'))$ où $c(f(x))$ et $\neg c(g(x'))$ sont deux littéraux, f et g deux fonctions affines et x et x' deux variables quantifiées universellement, on dérive la clause :*

$$f(x) = g(x) \Longrightarrow s(x') \vee r(x).$$

Le littéral $f(x) = g(x')$ représente la condition d'*unification* entre $c(f(x))$ et $c(g(x'))$. La clause dérivée est une conséquence des deux premières. De plus, on voit que si $c(f(x))$ et $\neg c(g(x'))$ sont deux littéraux non-affines, elles n'apparaissent plus dans la clause dérivée (sauf peut-être dans les termes $r(x)$ et $s(x')$).

L'algorithme consiste donc à appliquer de façon répétitive la règle de résolution sur les clauses de P et sur les clauses dérivées au fur et à mesure afin de trouver des clauses de littéraux affines. Ces clauses constituent alors $\hat{\mathsf{P}}$. Le théorème suivant, montré en section 4.5.1, page 123, donne un critère d'arrêt garantissant qu'aucun flou n'est ajouté lors de la traduction :

Théorème 1 *Aucun flou n'est ajouté lors de la traduction de P en $\hat{\mathsf{P}}$ si toutes les clauses affines pouvant être dérivées des clauses de P peuvent être dérivées des clauses de $\hat{\mathsf{P}}$.*

Le processus de dérivation de clauses affines à partir d'un ensemble de clauses non-affines est analogue à celui de la démonstration de théorèmes par réfutation de leur contraposée. Tester la satisfaisabilité d'un ensemble de clauses

contenant des contraintes non-affines est un problème indécidable. En revanche, tester la satisfaisabilité d'un ensemble de clauses affines est décidable (théorie de Presburger). Cela montre qu'en général, la construction des clauses de $\hat{\mathsf{P}}$ vérifiant le critère énoncé par le théorème précédent n'est pas un processus fini. L'exemple suivant illustre cette difficulté : considérons deux clauses, $\forall x, \neg c(x) \vee c(x+1)$ et $c(0)$. En combinant de façon répétitive la clause dérivée avec la première des deux clauses initiales, on obtient la suite : $\forall x_1, c(x_1+1) \vee (x_1 \neq 0)$, $\forall x_1, x_2, c(x_2+1) \vee (x_2 \neq x_1+1) \vee (x_1 \neq 0)$, etc. Autrement dit, on va dériver toutes les clauses $c(x+1)$ pour tous les x entiers.

Ce problème d'indécidabilité disparaît si l'on restreint le type de relations considérées sur les contraintes non-affines. Notamment, quand les arguments des contraintes non-affines sont des variables quantifiées universellement qui ne sont pas liées les unes aux autres par des littéraux affines (c'est-à-dire que l'ensemble des valeurs que peuvent prendre les variables universelles est un produit cartésien d'intervalles), alors on peut se ramener à un problème de résolution de calcul propositionnel. Reste que la dérivation exhaustive de toutes les clauses affines est coûteux. On peut d'abord imposer une limite au temps passé à faire des dérivations. On peut également choisir une méthode de résolution de complexité polynômiale et qui ne calcule pas toutes les dérivations possibles. On risque de perdre de la précision mais le résultat final reste quand même conservatif. De plus, les quelques relations sur les paramètres que l'on aura trouvées suffiront peut-être à améliorer significativement l'approximation.

La section 4.5.2 page 125 présente un algorithme basé sur la méthode de résolution linéaire à entrées directes [86] qui n'est pas complète (du flou peut donc être ajouté) mais elle suffit pour les relations du type de celles trouvées par une analyse structurelle, entre autres. La section 4.5.3 page 127 adapte la méthode précédente lorsque les contraintes sont vues comme des inégalités mettant en jeu des fonctions non-affines. Cette méthode permet alors de traiter correctement les relations linéaires entre les variables du programme.

Enfin, un cas particulier est le cas où les relations $\hat{\mathsf{P}}$ contraignent les paramètres du maximum à une seule valeur. L'analyse approchée est alors exacte, malgré la présence des contraintes non-affines. C'est notamment le cas pour l'exemple suivant :

```
  do x1 = 1 while (P(x1))
   do x2 = 1, n
S   A(x2) = ..
   enddo
   do x2 = 1, n
R    .. = A(x2)
   enddo
  enddo
```

Lorsqu'une instance de `R` est exécutée, alors on sait, grâce à l'environnement, que pour la même itération du `while`, une instance de `S` est exécutée et écrit la

valeur lue par l'instance de R. Ainsi, la source de $\langle \mathtt{R}, y\rangle$ est $\langle \mathtt{S}, y\rangle$. L'analyse est exacte.

IV.6 Construction d'ensembles de sources

La troisième et dernière étape de la construction d'une source approchée consiste à utiliser les propriétés $\hat{\mathsf{P}}$ caractérisant les valeurs possibles des paramètres du maximum pour simplifier l'expression de la source floue.

Cette simplification se fait en deux étapes : dans un premier temps, on transforme les clauses de $\hat{\mathsf{P}}$ en conditionnelles dont le seul but est de les combiner avec le quast source pour réaliser des simplifications. Une clause de $\hat{\mathsf{P}}$ peut s'écrire, sans perte de généralité, $\forall x, r(x) \implies e(x) \ll 0$ où r définit un polyèdre et e est une fonction affine. Cette implication est équivalente à l'inégalité $\max\{e(x)\,|r(x)\} \ll 0$. En calculant le maximum, l'inégalité devient une conditionnelle dont les feuilles sont des inégalités, appelée *quast de contexte*. Ce quast est alors combiné au quast source et simplifié pour éliminer les sources potentielles qui sont exclues grâce aux propriétés $\hat{\mathsf{P}}$. La source floue est définie comme un ensemble de quasts. La dernière transformation consiste à la transformer en un quast dont les feuilles sont des ensembles de sources. Il suffit pour cela de propager tous les prédicats mettant en jeu des paramètres vers les feuilles puis d'éliminer ces prédicats en construisant l'union des branches des conditionnelles dans lesquelles ils apparaissent.

IV.7 Implémentation

Nous avons réalisé une implémentation partielle des méthodes présentées dans cette section. Notre prototype, Caravan, utilise le formalisme d'une version précédente de l'analyse floue. De plus, seules certaines propriétés structurelles peuvent être intégrées dans le calcul de la source floue.

Présentation de Caravan

Caravan est un prototype programmé en Objective Caml, qui permet l'analyse floue d'un sous-ensemble de programmes Fortran. Le prototype compte environ 15000 lignes de code et est constitué de différents modules, dont les principaux sont :

L'analyseur syntaxique Le modèle de programme de notre analyse floue est inclus dans le sous-ensemble de programmes Fortran reconnu par le module. L'analyseur syntaxique est composé de deux parties. L'une, écrite en C [44], reconnaît des programmes Fortran 77 et C et produit une représentation intermédiaire commune, qui est à son tour reconnue par un programme en Caml. La première partie de l'analyseur syntaxique est une version améliorée de celui de PAF.

Module de calcul formel Ce module, écrit par P. Boulet, contient les définitions et les outils de base pour la manipulation des fonctions affines, des quasts ainsi que pour les calculs de maxima et minima à l'aide de PIP.

Module de test de dépendances Le test de dépendance est considéré comme une passe de simplification à l'analyse de flot de données. Le test est effectué à l'aide de PIP.

Module FADA Ce module implémente le calcul des sources paramétriques exactes en fonction des domaines de paramètres. Le module prend les clauses de $\hat{\mathsf{P}}$ mises sous forme de quasts de contexte comme entrée.

Module de traduction La traduction de P vers $\hat{\mathsf{P}}$ est effectuée par ce module. Les seules propriétés P qui peuvent être traduites sont des relations d'inclusion entre intersections et unions de domaines de paramètres (il s'agit de l'ancien formalisme de la FADA).

Analyse structurelle Les propriétés structurelles sont trouvées par ce module et écrites sous la forme de relations ensemblistes entre les domaines de paramètres.

Exemple

Pour l'exemple E1, notre prototype produit le texte du programme où chaque instruction est précédée par des commentaires dans lesquels apparaissent un numéro pour l'instruction et éventuellement les sources des variables lues par l'instruction. Nous ne détaillons ci-après que les sources de la variable `s`.

```
C STATEMENT NUMBER: 1
        do i = 1,n
C STATEMENT NUMBER: 2
         s = 10
C STATEMENT NUMBER: 3
         do while (z.ge.0)
C STATEMENT NUMBER: 4
C SOURCE OF s:
C  Parameters :
C  Quast :
C  if _alpha11 >= i then
C    if i >= _alpha11 then
C      if _alpha12 >= 1 then
C        if _while0-1-_alpha12 >= 0 then
C          [i; _alpha12], Statement4
C        else [i], Statement2
C      else [i], Statement2
C    else [i], Statement2
```

```
C  else [i], Statement2
          s = 2*s
          z = z-1
         enddo
C STATEMENT NUMBER: 5
C SOURCE OF s:
C  Parameters :
C  Quast :
C  if _alpha7 >= i then
C    if i >= _alpha7 then
C      if _alpha8 >= 1 then
C        [i; _alpha8], Statement4
C      else [i], Statement2
C    else [i], Statement2
C  else [i], Statement2
         z = s
        enddo
       end
```

Les paramètres du maximum sont notés `_alphan` et une opération $\langle \mathtt{n}, x \rangle$ est notée [x], `Statementn`. La variable `_while0` est le compteur de boucle du `while`.

IV.8 Autres travaux

Les analyses sensibles au flot des données qui sont effectuées à la compilation peuvent être classées en deux catégories :

– Les analyses calculent des dépendances de flot approchées entre les opérations accédant aux mêmes éléments de tableaux. C'est l'approche qui a été choisie pour l'analyse présentée dans cette section, ainsi que pour les analyses décrites par Collard [29], par Duesterwald *et al.* [46] et par Pugh et Wonnacott [106]. Ces analyses sont conservatives, par conséquent si une dépendance exacte existe entre deux opérations, alors une dépendance approchée existe.

– Les analyses manipulent des résumés des éléments de tableaux accédés par des blocs d'opérations. Le flot des données entre ces blocs peut être ensuite éventuellement déduit. Il faut remarquer que contrairement aux analyses précédentes, aucun GFD n'est explicitement calculé. Cette approche comprend la plupart des extensions aux structures de tableaux des analyses classiques de flot sur scalaires. Les méthodes basées sur les *définitions visibles* [64, 114, 59] conservent les informations relatives aux instructions qui ont définis en dernier des éléments de tableaux, ce qui n'est pas le cas des méthodes calculant des *régions de tableaux* [120, 34, 35, 68, 69, 122, 65], c'est-à-dire des résumés des éléments de tableaux utilisés ou définis par un fragment de code.

En général, la seconde approche a un plus large domaine d'application que la première, mais certaines techniques exhibant un parallélisme à grain fin, comme l'ordonnancement, ont besoin des résultats des premières analyses.

Analyses sur des éléments de tableaux

Nous comparons à présent trois méthodes calculant explicitement un GFD approché.

Duesterwald, Gupta et Soffa [46] : leur méthode étend l'analyse des définitions visibles [1] pour scalaires aux structures de tableaux. Pour chaque tableau et chaque instruction, une sur- et sous-estimation de la distance de dépendance entre la lecture courante et la dernière définition du même élément de tableau est calculée. L'approximation se fait sur des intervalles de distances, et non des polyèdres, ce qui peut conduire à des approximations grossières. De plus, les conditionnelles ne sont pas prises en compte, les indices des tableaux doivent être affines en le compte-tours de la boucle la plus profonde et la méthode ne traite principalement que les tableaux monodimentionnels.

Pugh et Wonnacott [105, 106, 129] : ils ont étendu leur méthode basée sur des relations de dépendance présentée en section III.4 afin de pouvoir traiter des fonctions non-affines en les compte-tours. Leur approche est très similaire à la nôtre. Dans leur premiers travaux [105], des sur- et sous-approximations de relations de dépendance étaient calculées en remplaçant les termes non-affines dans les formules par vrai ou faux. Ils ont également proposé une approximation plus fine lorsque les contraintes étaient des égalités de fonctions non-affines. Dans leurs travaux plus récents [106, 129], ils recourent à une analyse SSA pour trouver des relations entre contraintes non-affines, et peuvent utiliser n'importe quelle analyse présentée en section IV.4. On montre en section 4.8.3 page 146 que les mêmes analyses complémentaires permettant d'améliorer la précision du résultat peuvent être utilisés par la FADA ou leur méthode. En revanche, la façon dont les deux analyses de flot intègrent ces informations dans leurs calculs diffère grandement. La méthode de Pugh et Wonnacott introduit très souvent, même pour des propriétés structurelles, un flou indépendant des propriétés sur les contraintes non-affines. Nous avons montré en section 4.8.3 qu'il ne s'agit pas d'un problème dû à la représentation qu'ils utilisent. Nous avons adapté les principes exposés en section IV.5 à la représentation par relations de dépendance afin d'obtenir une méthode dont la précision est similaire à la FADA.

Collard et les précédentes versions de la FADA [29, 30, 12] : Le terme de FADA a été introduit par Collard [29] pour nommer une extension de l'analyse de flot exacte proposée par Feautrier aux programmes à contrôle

dynamiques. La première version de la FADA étendait le modèle des programmes à contrôle statique aux programmes avec `while` et conditionnelles `if..then`. Depuis, nous avons apporté dans [30, 12] et dans cette section plusieurs améliorations à la méthode originale. Nous avons changé le formalisme pour pouvoir traiter une plus large classe de contraintes non-affines, venant par exemple des indices de tableaux, et une plus large classe de propriétés sur ces contraintes non-affines. Simultanément, nous avons également énoncé les conditions pour lesquelles aucune information n'est perdue lors du processus d'intégration de ces propriétés dans le calcul. Nous détaillons ci-dessous les limitations des versions précédentes de la FADA.

Dans [29], les paramètres du maximum ne sont pas définis, les contraintes non-affines sont représentées par des *variables cachées*. Les indices de tableaux non-affines ne sont pas traités et seule la propriété structurelle du `while` est prise en compte. L'exemple E1 est traité correctement mais tous les autres conduisent à des sources très floues.

Dans [30], le formalisme est le même que dans le papier précédent, et l'analyse prend en compte le fait que seule l'une des deux branches d'un `if..then..else` est exécutée lorsque la conditionnelle l'est. L'exemple E2 donne toutefois une source plus floue que celle obtenue avec une analyse structurelle comme celle présentée en section IV.4 car l'analyse ne considère pas le fait qu'au moins l'une des deux branches est exécutée.

Dans [12], on introduit des paramètres du maximum et par conséquent donnons un algorithme de traduction entre des propriétés sur les contraintes non-affines et les propriétés sur les paramètres du maximum. Toutefois, les relations P ne peuvent être que des relations ensemblistes entre domaines de paramètres, et les seules analyses complémentaires sont les analyses structurelle et itérative. Toutes les relations sont traduites sans ajouter de flou.

Analyses recourant à des résumés

L'approximation faite par ces analyses est plus grossière et les dépendances sont représentées par des niveaux de dépendance. Cela s'avère suffisant pour la privatisation. On examine d'abord les techniques de calcul des définitions visibles, puis de régions de tableaux.

De nombreuses extensions des méthodes de calcul des définitions visibles ont été proposées pour les structures de tableaux [64, 114, 59]. Elles ne spécifient que l'instruction de la dernière écriture et non l'opération source, de la même manière que dans la méthode originale pour les scalaires. Pour chaque tableau et pour chaque instruction, des ensembles d'éléments de tableaux sont associés à l'instruction de leur dernière définition. Cependant lorsque plusieurs instructions peuvent définir un même élément, les conditions pour lesquelles la

définition visible est l'une ou l'autre des instructions ne sont en général pas calculées, ce qui conduit à une approximation.

Quant aux régions de tableaux, leur but premier était de représenter les effets des procédures sur les éléments de tableaux [120, 121, 24]. La région de tableau accédée par une référence à un élément de tableau est propagée le long du graphe de flot de contrôle. La complexité du calcul des régions dépend de la représentation choisie ainsi que de l'opérateur permettant de calculer la région résumant les effets de plusieurs flots de contrôle sur un tableau. Différents types de régions peuvent être calculés pour des programmes structurés [69, 122, 34, 65], et on décrit ci-dessous quatre d'entre eux en utilisant la terminologie de Creusillet [34, 35]:

- WRITE(S) : l'ensemble des éléments d'un tableau définis par S;
- READ(S): l'ensemble des éléments d'un tableau lus par S;
- OUT(S): l'ensemble des éléments d'un tableau qui sont vivants après S et qui vont être lus;
- IN(S): l'ensemble des éléments lus par S et qui n'ont pas encore été définis par S. Intuitivement, IN(S) est l'ensemble des éléments importés par S.

En fait, seules des sur- et sous-approximation de ces régions sont calculées. En intersectant des régions IN et WRITE d'instructions différentes, on peut tester si la source d'un élément de tableau provient d'une instruction définissant la région WRITE. Cela constitue le test effectué par la privatisation. En section 4.8.4, on effectue une comparaison détaillée des régions et de la FADA, et on montre que le degré de parallélisme atteint par une privatisation menée à partir du résultat de la FADA est au moins égal à celui auquel on parvient en utilisant des régions, comme formalisées par Creusillet.

IV.9 Conclusion

Nous avons présenté dans cette section une analyse approchée de flot de données capable de traiter n'importe quelle contrainte non-affine. Ces contraintes apparaissent par les indices de tableaux non-affines, les boucles `while`, les prédicats des conditionnelles ou les bornes non-affines des boucles `do`. Nous avons proposé une analyse qui est une extension de l'analyse exacte décrite par Feautrier. Non seulement cette analyse est exacte sur les programmes à contrôle statique mais comme l'approximation est effectuée à la dernière étape de calcul, les techniques usuelles de sources sont les mêmes que celles des programmes à contrôle statique. De plus, notre analyse est capable de prendre en compte une large classe de propriétés sur les contraintes non-affines, comme les propriétés qui sont produites par les nombreuses analyses symboliques existantes. En plus de ces techniques, nous avons proposé une méthode originale, appelée analyse itérative, qui trouve des relations entre les valeurs de contraintes non-affines par

l'examen des sources des variables qu'elles utilisent. Enfin, nous avons énoncé la condition pour laquelle l'approximation effectuée est la meilleure possible. L'analyse approchée a été validée par une implémentation partielle des techniques mises en jeu.

V Applications de l'analyse de flot

Nous décrivons les modifications qu'il est nécessaire d'apporter aux applications traditionnelles de l'analyse de flot exacte pour tenir compte d'un GFD approché. Cette section résume le chapitre 5.

V.1 Vérification de programmes

Dans un programme correct, toutes les variables sont initialisées avant d'être lues. La détection des variables non initialisées est une application simple et directe de l'analyse de flot de données. En effet, une variable est initialisée si et seulement si $\perp$ ne figure pas parmi les sources possibles trouvées par l'analyse. Ce simple fait peut aider le programmeur à vérifier la correction de son code et valider certaines propriétés concernant des contraintes non-affines.

Considérons un quast obtenu par l'analyse de flot de données d'une variable. Si une feuille de ce quast est $\perp$, cela signifie que cette variable n'a pas été correctement initialisée. En effet, la conjonction des prédicats sur le chemin de la racine du quast à $\perp$ représente la condition sur le vecteur d'itération de la lecture pour laquelle il n'y a pas eu d'initialisation. Cette condition est vérifiée au moins pour une valeur du vecteur d'itération si le quast a été simplifié.

Considérons à présent une dépendance directe entre l'instruction R et l'instruction S, à la profondeur p. Le résultat suivant peut être généralisé à une source construite à partir d'un nombre quelconque d'instructions. Lorsque $\perp$ est une des sources possible, cela signifie qu'une contrainte non-affine peut conduire à une initialisation incorrecte de la variable. On donne dans ce qui suit une caractérisation de toutes les contraintes non-affines possibles pour lesquelles le programme est correct. Cette propriété doit être vérifiée *a posteriori* par la contrainte apparaissant réellement dans le programme.

Examinons un quast dans lequel apparaît le paramètre du maximum $\alpha^p_{\mathtt{SR}}(y)$. Trions les prédicats qui sont sur le chemin de la racine à une feuille $\perp$ et soient $r_i(y), 0 \leq i \leq m$, les prédicats dépendant uniquement de y, et $s_j(y, \alpha^p_{\mathtt{SR}}(y)), 0 \leq j \leq n$ les prédicats dépendant également d'un paramètre du maximum. Pour que l'initialisation de la variable soit complète, la feuille $\perp$ ne devrait pas pouvoir être atteinte, c'est-à-dire que la condition suivante doit être remplie :

$$\forall y \in \mathsf{I}(\mathtt{R}), \left(\bigwedge_{0 \leq i \leq m} r_i(y) \right) \wedge \left(\bigwedge_{0 \leq j \leq n} s_j(y, \alpha^p_{\mathtt{SR}}(y)) \right) = \text{faux},$$

ce qui est équivalent à : pour tout $y \in \mathsf{I}(\mathtt{R})$ vérifiant $\wedge_{0\leq i\leq m} r_i(y)$ alors :

$$\bigvee_{0\leq j\leq n} \neg s_j(y, \alpha^p_{\mathtt{RS}}(y)).$$

Grâce à la définition de $\alpha^p_{\mathtt{RS}}(y)$, on obtient la condition équivalente suivante :

$$\forall y \in \mathsf{I}(\mathtt{R}) \text{ tq. } \bigwedge_{0\leq i\leq m} r_i(y), \exists x \text{ tq. } \left(\bigvee_{0\leq j\leq m} \neg s_j(x,y)\right) \wedge \left(\bigwedge_{h\in \mathsf{C}_{\mathtt{SR}}} c_h(x,y)\right) \quad (3)$$

Cette caractérisation des contraintes non-affines est une condition nécessaire et suffisante pour que le programme initialise correctement les variables lues.

Le fragment de code suivant illustre cette propriété :

```
   do x = 1, n
S     A(f(x)) = ..
   enddo
   do y = 1, n
R     .. = A(y)
   enddo
```

La contrainte non-affine est $c_1(x,y) = (f(x) = y)$. Sans information supplémentaire concernant f, la source floue de $\langle \mathtt{R}, y\rangle$ est constituée de $-$ et de $\{\langle \mathtt{S}, x\rangle \,|1 \leq x \leq n\}$. Par conséquent, on va attribuer une propriété à la contrainte non-affine afin que l'initialisation de `A` se fasse correctement. Le quast avec le paramètre du maximum est :

$$\textbf{if } 1 \leq \alpha^0_{\mathtt{SR}}(y) \leq n \textbf{ then } \langle \mathtt{S}, \alpha^0_{\mathtt{SR}}(y)\rangle \textbf{ else } -.$$

L'application directe de la caractérisation donnée par (3) pour $r(y) = (1 \leq y \leq n)$ (l'environnement) et $s(x,y) = \neg(1 \leq x \leq n)$ conduit à :

$$\forall y \text{ s.t. } 1 \leq y \leq n, \exists x \text{ s.t. } 1 \leq x \leq n, f(x) = y.$$

En d'autres termes, `A` est initialisé ssi f est une permutation sur $1..n$.

La vérification de cette condition peut être laissée au programmeur ou soumise à un vérificateur d'assertions du type de Floyd [56]. La vérification de code peut également être menée sur des fragments de code plus courts comme les corps des procédures, afin de comparer les variables et les conditions pour lesquelles elles sont initialisées à l'extérieur du code avec ce que le programmeur attendait.

V.2 Détection des récurrences

Redon définit dans [111] une méthode pour détecter et normaliser les récurrences d'un programme. Cette méthode est d'un intérêt évident pour la maintenance du code et pour les compilateurs paralléliseurs : la normalisation permet au programmeur de retrouver la sémantique de l'opération effectuée, quelque soit la manière dont elle a été implémentée. En outre, la détection en elle-même autorise des améliorations significatives de l'ordonnancement, du placement et du code produit par un paralléliseur qui peut faire appel à des bibliothèques ayant des versions optimisées des récurrences, si elles existent.

La détection des récurrences est basée sur un système d'équations linéaires récurrentes (SELR), obtenu grâce aux graphes de flot de données des variables du programme. La détection peut utiliser le résultat d'une analyse floue produisant un résultat exact. L'analyse structurelle permet la détection de récurrences à l'intérieur de boucles `while` et de branches de conditionnelles. L'analyse itérative permet l'utilisation de variables qui ne sont pas des compte-tours comme indices de la récurrence, tant que l'on dispose d'assez d'information sur ces variables pour que la source soit exacte.

On présente dans ce qui suit les conditions pour lesquelles la détection des récurrences est possible à partir d'une source floue. Le lecteur trouvera dans [111] les détails concernant la méthode générale de détection de récurrences pour des analyses exactes de flot de données. Chaque SELR donne l'expression d'une variable du programme en fonction d'autres variables. Cette expression est une fonction conditionnelle dépendant des compte-tours et des paramètres dûs au flou, lorsque la variable a une source floue. Chaque valeur possible définie par la conditionnelle est appelée une clause. L'algorithme de détection construit un graphe dont les arêtes sont les couples d'équations tels qu'une variable définie dans la première équation est utilisée dans une clause de la seconde. Les circuits de ce graphe sont alors normalisés de sorte qu'un algorithme de reconnaissance de formes appliqué sur chaque équation puisse décider s'il s'agit d'une récurrence. Les dépendances qui n'apparaissent pas dans un circuit du graphe n'interfèrent pas avec l'algorithme. Par conséquent, les variables dont les sources sont floues ne doivent pas être des accumulateurs de la récurrence, à moins que la seule source floue corresponde à la valeur initiale de l'accumulateur.

V.3 Expansion statique maximale

L'expansion des structures de données est une technique bien connue pour éliminer les dépendances qui ne reflètent pas le flot des données et qui gênent la parallélisation. Lorsque le programme est à contrôle dynamique ou que les tableaux ont des indices non-affines, on peut toutefois être amené à effectuer un calcul à l'exécution afin de préserver le flot des données, après expansion. Dans le cadre de la mise en *assignation unique statique* ou *static single-assignement* (SSA), on introduit des fonctions appelées ϕ aux points de contrôle

où plusieurs définitions visibles sont possibles [36, 37]. Ces fonctions ϕ représentent la partie du calcul à réaliser à l'exécution et constituent un surcoût notable lorsque les données sont réparties entre les processeurs ou en présence de tableaux. Le but de cette section est d'éviter ce coût et de trouver l'expansion de tableaux «statique», dans le sens «faite à la compilation», qui expanse le plus possible les structures de données. On appellera cette expansion une *expansion statique maximale*. Ce travail a été réalisé en collaboration avec Cohen et Collard [10].

Principe de la méthode

On pose quelques notations tout d'abord. Soit W l'ensemble des opérations d'écriture du programme, f la fonction qui fait correspondre aux opérations les cellules mémoires dans lesquelles elles écrivent et on note f' une expansion, c'est-à-dire une nouvelle fonction des opérations vers les cellules mémoires écrites.

Définition 4 (Expansion statique) *Une expansion statique est une fonction f' des opérations vers les cellules mémoires telle que :*

$$\forall \tau, \zeta \ : \ (\exists \nu \ : \ \tau \in \sigma(\nu) \wedge \zeta \in \sigma(\nu) \wedge f(\tau) = f(\zeta)) \Longleftrightarrow f'(\tau) = f'(\zeta).$$

On est intéressé par une expansion statique qui occupe le plus de cellules mémoires possibles.

Définition 5 (Expansion statique maximale) *Une expansion statique est maximale sur l'ensemble W si, pour toute expansion statique f'',*

$$\forall \tau, \zeta \in \mathsf{W} \ : \ f'(\tau) = f'(\zeta) \Longrightarrow f''(\tau) = f''(\zeta).$$

Intuitivement, si f' est maximale alors aucune fonction f'' ne peut associer deux cellules mémoires à deux opérations si f' ne le fait pas. Cette définition n'est cependant pas constructive. On va chercher les ensembles d'opérations sur lesquels une expansion statique maximale f' est constante. Pour cela, on définit la relation R sur les opérations :

$$\tau \mathrm{R} \zeta \Longleftrightarrow \exists \nu \ : \ \tau \in \sigma(\nu) \wedge \zeta \in \sigma(\nu).$$

On peut montrer (voir section 5.4.4) qu'une expansion statique maximale est définie par :

$$\forall \tau, \zeta \in \mathsf{W} : \ \tau \mathrm{R}^* \zeta \wedge f(\tau) = f(\zeta) \ \Longleftrightarrow \ f'(\tau) = f'(\zeta),$$

avec R^* la clôture transitive de R. Cela signifie intuitivement que les ensembles d'opérations sur lesquels l'expansion maximale est constante sont les ensembles d'opérations qui accèdent à la même cellule mémoire ($f(\tau) = f(\zeta)$) et qui sont sources possibles d'une même variable ($\tau \mathrm{R} \zeta$). Autrement dit, si $\mathsf{W}(c) = \{\tau \in \mathsf{W} \,|\, f(\tau) = c\}$ est l'ensemble des écritures écrivant la cellule mémoire c et

$\mathsf{M} = f(\mathsf{W})$ est l'ensemble des cellules mémoires, les ensembles sur lesquels f' est constante, que l'on notera W/f', sont définis par :

$$\mathsf{W}/f' = \bigcup_{c \in \mathsf{M}} \mathsf{W}(c)/\mathrm{R}^*.$$

Il ne reste plus ensuite qu'à numéroter (arbitrairement) ces classes et à faire l'expansion effective des structures. Si $n(\tau)$ est le label de la classe de l'opération τ, alors pour chaque écriture τ, on expanse le tableau $\mathtt{A}(f(\tau))$ en $\mathtt{A}(f(\tau), n(\tau))$. Un accès en lecture de l'élément $\mathtt{A}(g(\zeta))$ est transformé en $\mathtt{A}(g(\zeta), n(\sigma(\zeta)))$. La définition de n garantit en effet que toutes les opérations sources possibles de ζ qui accèdent à cet élément de tableau portent le même numéro.

Domaine de la méthode

La méthode d'expansion statique maximale dépend de deux choses :

- l'expression de la fonction source (peut-être floue) σ;
- le calculateur de la clôture transitive R^*.

On peut montrer que la méthode produit une expansion qui est maximale, étant donné une expression de σ et un calculateur de clôture transitive, et ce quelque soit leur précision, pourvu que les approximations qu'ils font soient conservatives.

Exemple

Considérons le tableau `A` du programme de la figure 4.a. Puisque `T` est toujours exécuté lorsque j est égal à N, une valeur lue par $\langle \mathtt{S}, i, j\rangle$, $j > N$, n'est jamais définie par une instance $\langle \mathtt{S}, i', j'\rangle$ avec $j' \leq N$. La figure 4.b décrit le flot des données entre les instances de `S`. Une flèche de (i', j') vers (i, j) signifie que l'instance (i', j') peut être la source de (i, j).

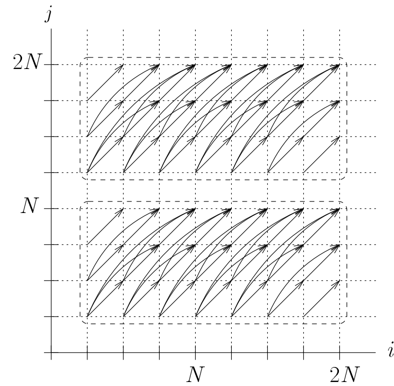
Formellement, la source d'une instance de `S` est :

$$\sigma(\mathtt{A}, \langle \mathtt{S}, i, j\rangle) = \left| \begin{array}{l} \textbf{if } j \leq N \\ \textbf{then} \left\{ \langle \mathtt{S}, i', j'\rangle \;\middle|\; \begin{array}{l} 1 \leq i' \leq 2N \\ \wedge 1 \leq j' < j \wedge i' - j' = i - j \end{array} \right\} \\ \textbf{else} \left\{ \langle \mathtt{S}, i', j'\rangle \;\middle|\; \begin{array}{l} 1 \leq i' \leq 2N \\ \wedge N < j' < j \wedge i' - j' = i - j \end{array} \right\} \\ \qquad \cup \left\{ \langle \mathtt{T}, i', N\rangle \,|\, 1 \leq i' < i \wedge i' = i - j + N \right\} \end{array} \right.$$

On note que le domaine d'itération de `S` peut être séparé en deux sous-ensembles disjoints en regroupant les opérations qui apparaissent dans le même flot. Ces sous-ensembles forment une partition du domaine d'itération. Une cellule mémoire différente peut être associée à chaque sous-ensemble, une cellule ne sera pas accédée par des opérations en dehors du sous-ensemble. Cette partition est

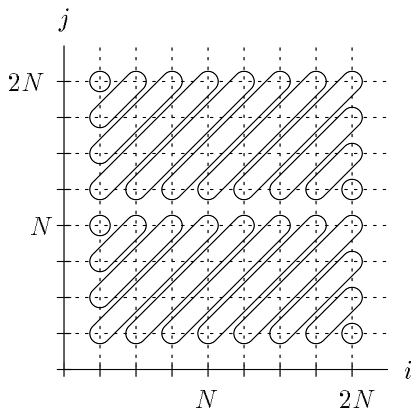
```
  real A(4*N-1)
  do i=1, 2*N
    do j=1, 2*N
      if .. then
S       A(i-j+2*N) = .. A(i-j+2*N)
      endif
T     if j = N then
        A(i+N) = ..
      endif
    enddo
  enddo
```

(a)

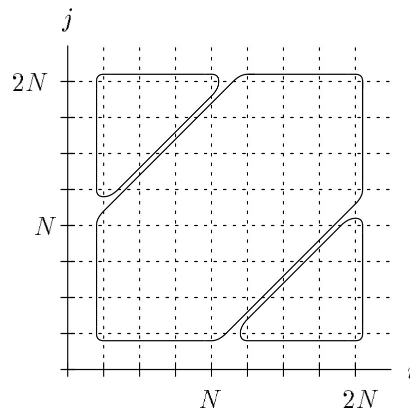


(b)

FIG. 4 - *Exemple d'expansion.*



(a)



(b)

FIG. 5 - *Partition du domaine d'itération ($N = 4$).*

donnée en figure 5.a. On peut seulement dupliquer les éléments de `A` qui sont utilisés deux fois. Ce sont les éléments `A`(c), $1+N \leq c \leq 3N-1$. Ils sont accédés par les opérations dans la partie centrale de la figure 5.b. Attribuons aux sous-ensembles de la partie inférieure de cette ensemble le numéro 1, et à ceux de la partie supérieure le numéro 2. On ajoute une dimension au tableau `A`, de taille 2. Les éléments `A`(c), $1 \leq c \leq N$ sont accédés par les opérations du coin supérieur gauche de la figure 5.b et n'ont qu'un sous-ensemble chacun. De même pour les éléments accédés par les opérations du coin droit. Le programme transformé après expansion statique maximale est en figure 6. Il est important de noter que ce programme à le même degré de parallélisme que si on l'avait mis en assignation unique, sans souffrir du surcoût à l'exécution.

```
  real A(4*N,2)
  do i=1, 2*N
    do j=1, 2*N

C     expansion of statement S

      if -2*N+1 <= i-j <= -N then
        if .. then
          A(i-j+2*N,1) = .. A(i-j+2*N,1) ..
        endif
      elsif -N+1 <= i-j <= N-1 then
        if j <= N then
          if .. then
            A(i-j+2*N,1) = .. A(i-j+2*N,1) ..
          endif
        elsif .. then
          A(i-j+2*N,2) = .. A(i-j+2*N,2) ..
        endif
      elsif .. then
        A(i-j+2*N,1) = .. A(i-j+2*N,1) ..
      endif

C     expansion of statement T

      if j = N then A(i+N,2) = .. endif
    enddo
  enddo
```

FIG. 6 - *Expansion statique maximale.*

V.4 Ordonnancement

Un ordonnancement donne une date d'exécution à chaque opération du programme. En général, cette date est logique et toutes les opérations sont suppo-

sées prendre une durée unitaire [51]. Des algorithmes d'ordonnancement ont été présentés dans [53, 54]. Comme nous l'expliquerons ici, il n'y a pas de difficulté à étendre ces algorithmes au cas d'une analyse de flot de données floue.

Des opérations en dépendance doivent être exécutées séquentiellement et les autres opérations peuvent être exécutées en parallèle. Si les dépendances de sortie et les anti-dépendances peuvent être éliminées (par privatisation ou expansion), alors le degré maximal de parallélisme peut être atteint en ne tenant compte que des dépendances directes : une opération doit être exécutée après toutes ses sources. Un ordonnancement θ doit alors vérifier, dans le cas de l'analyse exacte :

$$\theta(\langle \mathtt{R}, y\rangle) \geq \theta(\sigma(\mathtt{A}, \langle \mathtt{R}, y\rangle)) + 1,$$

pour n'importe quelle variable $\mathtt{A}$ lue par $\langle \mathtt{R}, y\rangle$. Dans le cas de l'analyse floue, toutes les sources possibles d'une lecture doivent précéder la lecture :

$$\forall \tau \in \mathsf{S}(\mathtt{A}, \langle \mathtt{R}, y\rangle) \ : \ \theta(\langle \mathtt{R}, y\rangle) \geq \theta(\tau) + 1. \tag{4}$$

Dans le résultat de l'analyse correspondante, τ dépend d'un paramètre de flou x_τ et apparaît dans une feuille gouvernée par un prédicat affine $r_\tau(y, x_\tau)$: $\tau = \zeta_\tau(y, x_\tau)$ où ζ est affine. On peut alors raffiner (4) en :

$$\forall x_\tau \ : \ r_\tau(y, x_\tau) \Longrightarrow \theta(\langle \mathtt{R}, y\rangle) \geq \theta(\zeta(y, x_\tau)) + 1. \tag{5}$$

Supposons que l'ordonnancement θ soit une fonction affine avec des coefficients inconnus. Puisque tout est affine dans (5), on peut appliquer la même technique que dans le cas exact [53, 54]. Comme on l'a montré dans la section précédente, il se peut que l'on ne puisse pas à la compilation éliminer toutes les anti-dépendances et dépendances de sortie. Dans ce cas, ces dépendances doivent également être prises en compte lors du calcul de l'ordonnancement.

V.5 Conclusion

L'ordonnancement et la détection des récurrences peuvent être aisément adaptés aux résultats produits par une analyse approchée.

En ce qui concerne l'expansion, il s'agit d'une optimisation classique éliminant les dépendances basées sur une réutilisation de la mémoire. Le code généré doit garantir que toutes les lectures accèdent à la bonne cellule mémoire après expansion. Quand l'analyse de flot est approchée, le principal inconvénient de la méthode est qu'elle a besoin d'un calcul à l'exécution pour déterminer complètement l'identité d'une cellule mémoire lue.

Nous avons présenté une nouvelle méthode d'expansion : une cellule peut être expansée autant de fois qu'il y a de classes indépendantes. L'algorithme peut se satisfaire des résultats de n'importe quelle analyse de flot, quelque soit sa précision, et n'a pas besoin d'un calcul de clôture transitive exact. Cependant, l'expansion statique trouvée est maximale en fonction de l'information dont on dispose à la compilation.

VI Conclusion

VI.1 Contributions

L'analyse exacte du flot des données décrit pour chaque opération le flot entre les variables du programmes. L'avantage d'une telle analyse est qu'elle rend possible, outre des applications comme le débogage, des techniques très agressives de parallélisation de programmes à la compilation. Cependant, le domaine de cette analyse est plus ou moins limité aux programmes à contrôle statique, et donc elle ne peut être utilisé que sur de petits fragments de code réel. De plus, la complexité au pire de l'analyse est élevée, ce qui restreint d'autant plus l'éventail des programmes qui sont traités efficacement.

Cette thèse propose des méthodes pour pallier ces deux inconvénients : nous avons décrit un algorithme pour calculer très efficacement, avec au pire une complexité polynômiale, les graphes de flot de données des programmes à contrôle statique et nous avons présenté un cadre général pour faire des analyses approchées du flot sur des programmes avec des boucles `while`, des `if..then..else` aux prédicats quelconques, et n'importe quel forme d'indices de tableaux. De plus, les techniques qui calculent le graphe de flot exact peuvent être réutilisées telles quelles, aussi bien les optimisations de l'analyse exacte que notre méthode polynômiale.

En outre, on a montré que la précision de notre analyse peut être améliorée à l'aide d'analyses complémentaires qui trouvent des propriétés sur les contraintes non-affines, comme par exemple c'est le cas pour les nombreuses analyses symboliques existantes. En plus de ces méthodes, nous avons proposé également une analyse itérative dont le principe est d'utiliser le résultat de l'analyse de flot d'une variable pour améliorer la précision d'une autre. Le calcul du graphe de flot de données est alors réalisé en deux étapes, la première est l'intégration des propriétés sur les contraintes non-affines dans le calcul, la seconde, indépendante de la première, est le calcul des sources. L'intégration des relations entre contraintes non-affines consiste à transformer un système de clauses logiques dans lequel apparaissent les contraintes non-affines en un système de clauses qui ne les utilise plus. Ceci est réalisé en appliquant de façon répétée une règle de résolution sur le système initial. En général, une analyse approchée ne peut pas garantir en un temps fini un résultat dont la précision correspond aux propriétés sur les contraintes, pour des raisons de décidabilité. Cependant, nous avons trouvé une condition nécessaire et suffisante pour que l'optimalité soit atteinte. Un prototype, bien que basé sur une version précédente de notre formalisme, valide cette approche choisie pour une analyse approchée.

Notre approche pour traiter les contraintes non-affines n'est pas dépendante de la représentation des dépendances utilisée. Suite à une comparaison avec les travaux de Wonnacott [129] et de Creusillet [34], nous avons montré que notre analyse approchée donne de meilleurs résultats pour les programmes dans son domaine que d'autres analyses et permet ainsi l'utilisation de techniques de

parallélisation plus agressives.

La plupart des applications traditionnelles de l'analyse exacte de flot de données pour tableaux ne nécessite qu'une adaptation mineure pour pouvoir prendre en compte les résultats d'une analyse approchée. Ce n'est pas le cas de l'expansion de tableau, et nous avons présenté une méthode pour faire l'expansion des structures de données sans pour autant recourir à la restauration dynamique du flot.

VI.2 Perspectives

De nombreux développements de l'analyse approchée de flot de données elle-même peuvent être envisagés. Dans un premier temps, un certain nombre de techniques décrites dans cette thèse peuvent être améliorées :

- les méthodes de résolution : nous n'avons décrit qu'une règle générale de résolution suivie ensuite de la résolution à entrées directes. Cependant, il existe plusieurs autres types de résolutions, chacun adapté à une forme particulière de clauses (comme par exemple la résolution SLD pour les clauses de Horn, utilisées par Prolog).
- le calcul des quasts de contexte : les règles de construction de ces quasts, données en section 4.6.1 peuvent être optimisées, ainsi qu'on l'a laissé entendre en section 4.6.3;
- l'extension de l'analyse approchée aux programmes avec appels de procédure : Leservot [82] a proposé une extension de la méthode polyédrique à ces programmes. L'adaptation de sa méthode à une analyse approchée devrait être relativement simple, mais n'a pas été formalisée.

Sur un plan plus théorique, mise à part l'analyse itérative, il n'y a pas de rétroaction entre les analyses qui trouvent des relations sur les contraintes non-affines et l'analyse approchée du flot des données. Nous avons montré qu'il était possible de déterminer si une relation améliore la précision de la source et si elle peut être complètement prise en compte par l'analyse. Ces deux critères n'ont pas été vraiment utilisés jusqu'à présent, mais devraient permettre de décider s'il est intéressant de trouver d'autres propriétés sur une contrainte non-affine. Cette idée de rétroaction cependant n'est utilisable que lorsque l'analyse symbolique est capable de raffiner son résultat progressivement, ou d'intégrer les résultats d'une analyse de flot de plus en plus précise. Ceci n'est possible pour l'instant qu'avec l'analyse itérative.

De plus, un important travail d'optimisation de l'analyse reste à faire concernant le choix de la représentation des dépendances de flot. Suivant l'idée présentée en section 4.8.4, la représentation des dépendances pourrait être dégradé pendant l'analyse, soit pour des raisons d'efficacité, quitte à perdre de la précision, soit parce qu'il n'y a aucune raison de garder une description précise de flot des données. Le premier cas peut être justifié par les gros codes, car

par exemple la probabilité qu'une écriture soit source d'une lecture décroit avec la distance entre les deux instructions. L'idée de partitionner le code en différent fragments qui peuvent être analysés séparément a été traité dans le cas des analyses exactes par Berthou [15]. La deuxième raison peut être motivée par le degré de parallélisme désiré. Ainsi, la privatisation de tableaux n'a besoin que d'une information concernant les dépendances entre itérations, et pour l'analyse interprocédurale, les régions abstraient les effets des procédures sur les éléments de tableaux (voir la thèse de Leservot [82] pour une adaptation de l'analyse exacte de flot de données à l'analyse interprocédurale). Plus généralement, la précision de l'analyse, soit contrôlée par sa représentation soit par les propriétés sur les contraintes non-affines, devrait dépendre également de l'application visée. Il semble clair de dire que la plupart des analyses de dépendances ont étés adaptées jusqu'à présent à un type particulier d'application et ne peuvent pas changer facilement la précision de leur résultat. Une dernière remarque à propos de la représentation des dépendances : l'analyse approchée que nous avons proposée dépend largement du formalisme d'analyse exacte. Les contraintes non-affines ont étés éliminées afin que les techniques utilisées dans le cas exact puisse être appliquées. Généraliser cette approche aux programmes récursifs n'est pas immédiat puisqu'il n'existe pas d'analyse exacte pour de tels programmes.

Enfin, nous n'avons pas adapté certaines applications de l'analyse exacte du flot de données. C'est le cas de l'optimisation mémoire décrite par Lefebvre [80, 81]. On ne sait pas encore comment mettre en relation cette optimisation avec l'expansion mémoire statique présenté en section 5.4.

Chapter 1

Introduction

1.1 Context

The constant improvements in processor technology has lead to a fast increase of processor frequency and raw computing power. However this progression has not been followed by a similar advance in memory capacity and performances, and the relative slowness of memory accesses, even with a hierarchy of memories, prevents these processors from reaching their peak performances. Sequential machines no longer meet the performance requirements of many applications, among which the famous *Grand Challenge*, simulations and multimedia applications. Simulations, which take a more and more important place in many fields either for economic reasons (in aerodynamics for instance) or because experimentation is not possible (in seismology, cosmology, ...), need to manipulate very large amounts of data in reasonable time and multimedia applications impose heavy constraints on execution time.

Since sequential computers do not meet performance requirements, the idea is to divide a program or the input data into parts that are then treated by as many processors. This idea of parallel execution first gave rise to vector processors such as the Cray I. The same instructions were then performed over a vector of data. However the efficiency of such architectures tightly depends on the regularity of the parallelized code. A more general way to improve performances at the processor level is achieved through Very-Long Instruction Word (VLIW) or superscalar architectures (processors Alpha from Digital, for instance). Several chunks of instructions are then executed in parallel. Finally, another approach consists in multiplying the number of processors in a machine, expecting ideally a similar multiplication of the performances. The memory is either distributed (as in the IBM SP2 or for networks of workstations) or shared between the processors (as in the Cray YMP).

In order to achieve high performances on parallel computers, a program, or at least the algorithm it implements, must contain a significant degree of parallelism. Even then, either the programmer and/or the compiler has to

expose this parallelism and apply the necessary optimizations to adapt it to the particular characteristics of the target machine. Moreover, the program should be portable in order to cope with the fast obsolescence of parallel machines. Two possibilities are offered to the user to meet these requirements:

- Explicitly parallel languages. Most languages proposed are parallel extensions to sequential languages, such as OCCAM, Fortran D [72], Vienna Fortran [132], HPF [57], ... Some extensions also appear under the form of libraries: PVM [119] and MPI [63] for instance. This approach makes the programmation of high performance parallel algorithms possible. However, besides parallel algorithmics, the user is also in charge of more technical and machine-dependent operations, such as the distribution of data on the processors depending on their memory capacities, communications and synchronizations, ... which require a deep knowledge of the target architecture. Some efforts have been done in HPF so as to make the compiler take care of some parts of this job, but it seems that, up to now, the programmer still needs to have a thorough knowledge of what the compiler does ... As for portability issues, this has been one of the main concerns of HPF and of the more low-level libraries, PVM and MPI. However, portability does not mean effectiveness and the choice of the parallel model inevitably affects the performances.

- Automatic parallelization of a high level sequential language. The obvious advantages of this approach are the portability, the simplicity of programmation and the fact that even old undocumented sequential codes may be automatically parallelized (in theory). However the task alloted to the compiler-parallelizer is overwhelming. Indeed, the program has first to be analyzed in order to understand, at least partially, what is performed and where the parallelism lies. The compiler then has to take some decisions about how to expose this parallelism in the best way so as to take into account the specificities of the target architecture. Even for simple programs and a simplified model of parallel machine, "optimality" in both of these steps is out of reach, for decidability reasons. As a matter of fact, a wide panel of parallelization techniques exists, of various complexity and efficiency. The difficulty relies in the choice of the methods to apply, choice which is often guided, so far, by the experience of the programmer. Thus, alike HPF, the process of parallelization requires an effort from both the programmer and the compiler.

  The usual source language for automatic parallelization is Fortran77. Indeed, many scientific applications have been written with this language, and Fortran allows only relatively simple data structures (scalar and arrays) and control flow. Several studies however deal with the parallelization of C or of functional languages such as Prolog or Lisp. Many non-commercial automatic parallelizers and parallelizing tools already ex-

ist: Bouclette from LIP at Lyon [21], Loopo from FMI at Passau [62], Parafrase-2 and Polaris from CSRD at the University of Illinois [100, 20], PAF from Versailles University, PIPS from Ecole des Mines [73] and SUIF from Stanford University [3], among others, as well as some commercial parallelizing tools, such as CFT, FORGE, FORESYS or KAP.

This thesis will focus on the techniques of automatic parallelization and more specifically on the kind of analysis that finds where the parallelism of a program lies.

1.2 Dependence Analysis

Automatic parallelization needs a thorough understanding of the behavior of the program to parallelize. This knowledge then allows the compiler to expose the underlying parallelism, exploit data locality, optimize memory usage through code transformations, or detect recurrences in order to yield an efficient parallel program.

The behavior of a program on a set of input values is completely determined by the evolution of the values taken by the variables through execution. Gathering information on data values is a classical task in advanced compilers, known as *Dataflow Analysis* [1]. The properties can be collected by executing the program in a new semantic that approximates the original semantic of the language. How to build this approximated semantic is explained in [32]. The first techniques based on dataflow analysis for automatic parallelization were designed for the collection of properties on the values of scalar variables. A dataflow equation is associated to every statement of the program and this system is then solved either iteratively or directly [115]. These methods have been extended since then to array structures but they either consider an array as an indivisible object or collect information only about sets of array elements.

Vectorization and parallelization techniques are mainly based on the discovery of the parallelism hidden by independent references to distinct parts of arrays. Two operations are said to be in dependence if they access the same memory cell and if one of these accesses is a write. As for many properties of the program, dependences may be found with traditional dataflow analysis but more specific methods have been designed to handle dependences between array structures. Many dependence tests have been devised in order to check the legality of code transformations [7]. However, most of these tests are not exact, and even when they are, cannot distinguish between true dependences, which describe a real information flow, and spurious dependences, in which the value purported to be transmitted is destroyed before being used. More precise methods have been designed to compute, for every array cell read in an expression (called the *sink*), the very operation which produced it (called the *source*). These methods are called *Array Dataflow Analyses* [23, 52] or *Value*

Based Dependence Analyses [103]. Such accurate information significantly improves the quality of operation reordering and memory optimizations among array variables, therefore the performances of the parallel program. Current exact array dataflow dependence analyses tightly depend on two things:

- Each operation, that is, each execution of a statement, is defined by its statement and by a finite representation of the logical date of execution. The finiteness of the description is necessary for a compile-time analysis. Moreover, the representation of logical time must be in relation with control structures in order to be able to use the results of the analysis for code transformations. Hence the execution trace is not a suitable representation in general and programs with general recursive procedure calls or unstructure `goto`*s* are not handled by exact analyses. Most frameworks are limited to intraprocedural program fragments where the only control structures are the sequence, the loops (`do` or `while` loops) and the conditionals. The logical time is then represented as the integer vector of the surrounding loop counters.

- Once the dependence problem has been formulated, linear integer programming tools are used so as to find an exact solution that verifies all the constraints of the dependence. This entails that all constraints must be affine with respect to the variables and parameters of the integer program. In most frameworks, the variables chosen are the loop indices. Thus any array subscript, loop bound or conditional predicate that appears in the dependence problem must be affine with respect to the loop counters. Programs with a single procedure and where the only control structures are the sequence, loops and conditionals, fulfilling these constraints of linearity are called *static control* programs.

Exact array dataflow analyses make therefore quite stringent hypotheses on the input programs and the use of integer programming tools shows that in general, this kind of analysis is expensive. Some efforts have been done in order to reduce this cost and we also address this problem in this thesis. We present a method that is very efficient on most usual programs in the scope of exact analyses and that handles some simple non-affine cases. But the main purpose of this thesis is to provide a general framework for approximate array dataflow analyses in presence of non-affine constraints. The scope of such analyses thus includes program fragments containing `while` loops, non-affine conditional predicates and non-affine array subscripts. The idea is to compute for each sink not one, but a set of possible sources. The analysis, although approximate, operates at the operation level and at the array element level. Note that we do not address in this thesis the problem of complex control flows introduced by interprocedural calls or `goto`s. So as to improve the quality of the approximation, we also present two methods, called structural and iterative analyses, which find relations between non-affine constraints. And most

of all, given any set of properties on non-affine constraints, either obtained by the preceding analyses or by any symbolic analysis, we propose a technique to automatically compute the set of possible sources. We establish moreover the conditions for which this set is the smallest set one can obtain. This last result is very important since it proves that our analysis, although more expensive, still provides more accurate results than other approximate methods and thus justifies its use for some applications. Finally, as our analysis is an extension of exact array dataflow analysis, we describe the necessary adaptations of traditional dependence analysis applications and the improvements gained. We will especially study the case of memory expansion.

1.3 Overview

This thesis is organized as follows: Chapter 2 describes in detail the role of dependence analysis in automatic parallelization, and the different abstractions that have been proposed. Chapter 3 presents the exact array dataflow analysis as formalized by Feautrier, improved by some optimizations given in Section 3.2.3. In Section 3.3, a polynomial-time alternative algorithm is proposed, and a discussion and comparison with other methods is performed in Section 3.4. The general framework for approximate array dataflow analyses is presented in Chapter 4. The techniques that improve the approximation are described in Section 4.4. Accuracy issues are discussed in Section 4.5 and comparison with other existing frameworks takes place in Section 4.8.

Finally, several applications are detailed in Chapter 5, such as program checking, recurrence detection, memory expansion and scheduling. We will focus more specifically on the problem of array expansion after an approximate analysis in Section 5.4.

Chapter 2

Dependence Analysis in Parallelization

There are many ways to represent dependences and as many ways to use this information in order to parallelize a program. This chapter focuses on the kind of dependence information that is needed by the techniques to expose parallelism.

We first introduce the notations that will be used in this dissertation, present the different dependence abstractions in Section 2.2 and then propose how to expose parallelism from the dependence graph in Section 2.3.

2.1 Notations and Definitions

All along this thesis, we will use the following notations: The k-th entry of vector x is denoted by $x[k]$. The dimension of a given vector x is denoted by $|x|$. The sub-vector built from components k to l is written as: $x[k..l]$. If $k > l$, then this vector is by convention the vector of dimension 0, which is written $[]$. Furthermore, $\ll$ denotes the strict lexicographic order on integral vectors and will be consider as the default order relation on sets of vectors. Maxima and minima on sets of elements are taken with respect to the implicit order on these elements. (For instance, max on a set of vectors means $\max_{\ll}$).

By convention, program statements are labeled by capital letters in type-writer style. Sets are denoted by capital letters and operations (instances of statements) by letters of the Greek alphabet. Finally, functions use the same typographic conventions as the objects they return.

An instance of statement `S` is denoted by $\langle \mathtt{S}, x \rangle$, where x, the iteration vector of `S`, is the vector built from the counters of loops surrounding `S` –including `while` loops –from outside inward. *Structure parameters* are variables that are defined at most once. Their values are linked to the size of the problem addressed by the program.

Non-affine constraints are equations or inequalities which depend on variables other than loop counters and structure parameters, and/or are non-linearly dependent on loop counters and structure parameters. For example, non-linear constraints may come from predicates of `if` or `while` constructs or from array subscripts. Obviously, some non-linear constraints can be removed by replacing some variables by their expression in terms of loop counters and structure parameters (induction variable detection and forward substitution). Similarly, some `while` loops can be transformed into `do` loops. We will suppose here that these simplifications have been performed, when possible, by a previous phase of the compiler.

2.2 Dependences Abstractions

We first give in this section a formal definition of the dependences and describe some commonly used dependence abstractions.

2.2.1 Definition of dependences

In a sequential program, the execution order between operations, denoted $\prec$, is total: Given two operations, one executes before another. But in fact, this total order is enforced by the semantic of the control structures of sequential languages (loops, sequence, ...), and may not be necessary for the outcome of the program. The goal of dependence analysis is to extract from the sequential total order $\prec$ the "smallest" partial order $\prec_{//}$ that preserves the behavior of the program, that is, the values assigned to the variables during execution. Operations that are not comparable in this order may be executed in any order or in parallel. Two operations are in dependence if they access the same resource and one of them changes the state of this resource. The resource can be of any kind (disk, modem, printer, etc.), and we consider henceforth, without loss of generality, that it is a memory cell. Bernstein [13] distinguished three kinds of data dependences:

- Flow-dependences: An operation writing into a memory cell is followed by an operation reading the same cell.
- Output-dependences: Two operations write into the same cell.
- Anti-dependences: An operation reading a memory cell is followed by a write operation into this cell.

Anti- and output-dependences are the result of memory reuses. They do not convey any particular signification for the underlying algorithm and only reflect programming techniques. Moreover, it is worthwhile noticing that many dependences are obtained by transitivity. Indeed, an output-dependence followed by a flow-dependence gives rise to another flow-dependence. However, the value

produced by a write operation of a flow-dependence obtained this way is *killed* or *covered* by the second write of the output-dependence. Flow-dependences that cannot be derived by transitivity are called *direct* or *value-based* dependences. The values then flow from write to read operations. The sub-graph of the dependence graph formed by all direct dependences is called the *Data-Flow Graph* (DFG).

To sum up, there exists a dependence from the operation $\langle \mathtt{S}_1, x_1\rangle$ to $\langle \mathtt{S}_2, x_2\rangle$, accessing respectively the memory cells c_1 and c_2, if and only if:

- $\langle \mathtt{S}_1, x_1\rangle$ and $\langle \mathtt{S}_2, x_2\rangle$ are executed.
- $\langle \mathtt{S}_1, x_1\rangle$ and $\langle \mathtt{S}_2, x_2\rangle$ access the same memory cell: $c_1 = c_2$.
- $\langle \mathtt{S}_1, x_1\rangle$ is executed before $\langle \mathtt{S}_2, x_2\rangle$: $\langle \mathtt{S}_1, x_1\rangle \prec \langle \mathtt{S}_2, x_2\rangle$.
- One of the two operations is a write.

Moreover, the sequential order $\prec$ can be defined explicitly: Let $d_{\mathtt{S}_1\mathtt{S}_2}$ be the number of loops surrounding both $\mathtt{S}_1$ and $\mathtt{S}_2$. Since the quantity $d_{\mathtt{SS}}$ occurs very often in the following, it will be abbreviated as $d_{\mathtt{S}}$. Let $\lhd$ be the textual order of the program. $\mathtt{S}_1 \lhd \mathtt{S}_2$ iff $\mathtt{S}_1$ occurs before $\mathtt{S}_2$ in the source text. The sequential execution order is then:

$$\langle \mathtt{S}_1, x_1\rangle \prec \langle \mathtt{S}_2, x_2\rangle \Longleftrightarrow \bigvee_{p=0}^{d_{\mathtt{S}_1\mathtt{S}_2}} \langle \mathtt{S}_1, x_1\rangle \prec_p \langle \mathtt{S}_2, x_2\rangle$$

where

$$0 \le p < d_{\mathtt{S}_1\mathtt{S}_2} : \langle \mathtt{S}_1, x_1\rangle \prec_p \langle \mathtt{S}_2, x_2\rangle \Longleftrightarrow (x_1[1..p] = x_2[1..p]) \wedge (x_2[p+1] < x_2[p+1]),$$

$$p = d_{\mathtt{S}_1\mathtt{S}_2} : \langle \mathtt{S}_1, x_1\rangle \prec_p \langle \mathtt{S}_2, x_2\rangle \Longleftrightarrow x_1[1..d_{\mathtt{S}_1\mathtt{S}_2}] = x_2[1..d_{\mathtt{S}_1\mathtt{S}_2}] \wedge \mathtt{S}_1 \lhd \mathtt{S}_2.$$

We say that there is a dependence *at depth* p from an operation τ to ζ if the two operations are in dependence and if $\tau \prec_p \zeta$.

2.2.2 Traditional dependence representations

In general, the number of dependences in a program depends on the number of iterations of the loops, which can be unknown at compile-time. Thus the dependences cannot be represented in extenso and finite abstractions need to be used instead. Moreover, the conditions of dependence are in general approximated, meaning that the smallest partial order $\prec_{//}$ is not extracted from $\prec$ for all programs. Many representations have been designed and we present thereafter the most usual ones. We will illustrate each of these on the following program taken from [43].

```
program sample
 do i=1, n
```

```
    do j=1, n
S     a(i,j) = a(j,i) + a(i,j-1)
    enddo
  enddo
```

By decreasing order of accuracy:

- Dependence relation/Quast [102, 52]: Dependences from $\langle \mathtt{S}_1, i_1\rangle$ to $\langle \mathtt{S}_2, i_2\rangle$ are represented as $\{(i_1, i_2)\,|P(i_1, i_2)\}$ where P is a Presburger formula, that is, a first order predicate of integer additive arithmetics. Another representation uses quasts (Quasi-Affine Selection Tree), and is equivalent to dependence relations as far as dependences are concerned. We will describe quasts more thoroughly in the following chapter.

  In program `sample`, there are three kinds of dependences:

  - a flow-dependence from $\langle \mathtt{S}, i, j\rangle$ to $\langle \mathtt{S}, j, i\rangle$, provided that $1 \le i < j \le n$
  - an anti-dependence from $\langle \mathtt{S}, j, i\rangle$ to $\langle \mathtt{S}, i, j\rangle$, provided that $1 \le i < j \le n$.
  - a flow-dependence from $\langle \mathtt{S}, i, j\rangle$ to $\langle \mathtt{S}, i, j+1\rangle$ provided that $1 \le i \le n, 1 \le j < n$.

  All these dependences are represented exactly with the dependence relation abstraction. They are, respectively:

$$\{(i,j),(i',j')\,|i' = j \wedge j' = i \wedge 1 \le i < j \le n\}\,,$$
$$\{(i,j),(i',j')\,|i' = j \wedge j' = i \wedge 1 \le i < j \le n\}\,,$$
$$\{(i,j),(i',j')\,|i' = i \wedge j' = j+1 \wedge 1 \le i \le n \wedge 1 \le j < n\}\,.$$

- Dependence polyhedron/cone [74] : Dependences from $\langle \mathtt{S}_1, i_1\rangle$ to $\langle \mathtt{S}_2, i_2\rangle$ are approximated by the set $\{i_2 - i_1)\,|P(i_1, i_2)\}$ where P defines a convex polyhedron approximating the conditions for which there is dependence. In the dependence cone abstraction, the polyhedron is replaced by a cone.

  The three dependences of program `sample` can be represented exactly in the polyhedron abstraction.

- Distance/Direction vectors [127] : Dependences from $\langle \mathtt{S}_1, i_1\rangle$ to $\langle \mathtt{S}_2, i_2\rangle$ are represented by an approximation of $i_2 - i_1$. For each of its component, the difference is either a constant, or approximated by its sign (+, −), or $*$ if the sign is unknown.

  The dependences of program `sample` are approximated respectively by the vectors $(+,-)$, $(+,-)$ and $(0,1)$.

- Dependence levels [2]: The level of a loop is the number of its surrounding loops, plus one. (The outermost loop has a level of 1.) Given a dependence between two operations, if these operations do not execute at the same

iteration of the same loop, then this loop *carries* the dependence. A dependence between two operations is represented by the level of the outermost loop carrying the dependence.

The dependence levels in program `sample` are respectively 1, 1 and 2.

2.3 Exposing parallelism

The application field of dependence analysis is very wide, from debugging to reengineering or code optimization. Parallelization is one of these optimization techniques. The choice of dependence abstraction is often guided by the method used to expose parallelism and we briefly present thereafter some techniques for code transformation, operation scheduling, array privatization, array expansion and recurrence detection, and the accuracy of dependence abstraction they require.

2.3.1 Code transformations/ Scheduling

Vectorizing and parallelizing compilers often resort to code transformation in order to exhibit parallelism, improve memory locality or take advantage of cache memories. There exists a lot of literature about such transformations. (See [127, 133] for a description.) A transformation can be applied only if dependence relations are still preserved after the transformation. Many dependence tests of various accuracy have been devised (Banerjee test [7], λ test [85], PIP test [52], PIPS test [73], Omega test [102], ...). If dependences have been found, depending on the precision, dependence abstractions then contain either not enough or sufficient or too much information to decide the legality of a transformation. Yang [130, 131] found the minimal data dependence abstraction corresponding to each loop transformation.

For instance, loop permutation can transform a loop nest with an outermost parallel loop into a loop nest with an innermost parallel loop. Such transformation is necessary in order to exploit this parallelism on a vector of processors. In the following loop nest:

```
do i=1, n
 do j=1, n
  a(i,j) = a(i,j-1) + ..
 enddo
enddo
```

The i loop is parallel. The minimal dependence abstraction for loop permutation is dependence direction. In this case, dependence testing shows that there are flow-dependences and they are represented by the vector

$(0,+)$. The transformation is legal because the vector obtained after transformation, $(+,0)$, is lexicopositive, which is the legality condition for loop permutation. The vectorized code is then:

```
do j=1, n
 a(1:n,j) = a(1:n,j-1) + ..
enddo
```

Applying successive code transformations so as to parallelize a program has one major drawback: the optimal application order is unknown, and it is likely that such an order does not exist. Hence more systematic methods have been devised, deciding a priori the order in which the compiler should attempt to apply the transformations: this is the case of Allen and Kennedy's algorithm [2]. Dependences are represented as dependence levels, and loop splitting and loop parallelization are the only transformations applied on the code. The idea is to split loop nests into as many nests as there are strong connected components in the dependence graph, and then to parallelize as many loops as possible in each nest. As only two kinds of transformations are used, their method do not extract all possible parallelism. Nonetheless, Darte and Vivien [42] have shown that this method is optimal with respect to the dependence abstraction it uses.

An alternative approach, based on matrix transformations, has been proposed and used for an important subset of loop nests: nests with uniform dependences, that is, with dependences of constant distance vector. This property is shared by all systolic array algorithms and many linear algebra codes also belong to this class. The hyperplane method, described by Lamport [78], partitions the iteration space of a loop nest into parallel hyperplanes associated to operations that can be computed in parallel. The transformed code then looks like:

```
do t = 0 to L
 doall p in H(t)
  ...
 endoall
 synchronize
enddo
```

where the outmost loop on `t` is sequential and the other loops are parallel. The body of the loop is considered in this method as a unique compound instruction, and all statements in this block are scheduled with the same linear function. The hyperplanes $H(t)$ are parallel and $H(t+1)$ is obtained from $H(t)$ by translation. The translation vector is denoted π and called the time vector. A dependence is preserved by this transformation if and only if its dependence vector d verifies $\pi d > 0$, that is, $\pi d \geq 1$. Indeed, this constraint ensures that an operation that depends on an other one is still executed after this one, once the transformation is performed. From this constraint, Lamport finds an algorithm that builds a

vector π. Darte et al. [38, 39] have shown that the time vector could be chosen so as to minimize the latency L.

This approach has been extended to affine dependences w.r.t. loops indices and to affine schedules by statements. Two methods have been proposed to build the transformed loop nest:

1. Step by step transformation: The final loop nest is built by successive applications of loop interchange, loop reversal or loop skewing transformations. All three are modeled as unimodular transformations on the iteration space. Wolf and Lam [126] presented an algorithm that proceeds in this manner, using distance/direction vectors as dependence abstractions.

2. Parametric integer programming solution: The system of dependence constraints on scheduling functions is solved by integer programming in the polyhedron model and the bounds of the new loop nest are found by computing lexicographic maxima and minima on the iteration polyhedron. In this method, explicit scheduling functions are computed first, assigning to every operation of the program a date of execution, and then building from these functions the corresponding code. This is the approach proposed by Feautrier, with either a monodimentional or multidimensional time [53, 54] and requires the computation of the DFG, that is, an exact dependence analysis. A more efficient method, mostly targeting uniform dependences, has been developed since then by Darte and Robert [40, 41].

The second method, based on the polyhedral abstraction, is more precise than the first one. The second approach is the only one that finds a degree of parallelism in the example `sample` presented in the previous section page 63 [43]. Since this method is based on integer linear programming, its scope is restricted to static control programs and is much more expensive. Darte and Vivien discussed the accuracy of dependence abstractions and the maximal degree of parallelism that can be expected. They proposed a method of intermediate complexity between Wolf and Lam's and Feautrier's methods, based on the polyhedral abstraction [43] and mostly aimed at perfect loop nests.

We have focused on the computation of scheduling functions, which are directly constrained by the dependences. Data locality can be exposed by partitioning the innermost loop nests, which is also performed during the construction of the new loop nest. We will not discuss these techniques in this thesis and the reader is referred to the wide literature about this subject ([75, 126, 53, 38, 41, 22] to name a few).

2.3.2 Avoiding memory reuse

As hinted in Section 2.2.1, anti- and output-dependences do not characterize an algorithm, but only one of its implementation. Several studies [83, 17, 99] of the Perfect Club Benchmarks [14] have shown that a significant degree of

parallelism can be exposed by eliminating spurious dependences. Concerning array structures, two techniques can be distinguished: array privatization and array expansion.

Array privatization

Array privatization transforms variables used as temporaries into private copies for each processor in the parallelized program. This technique cuts down the number of synchronization or communication steps and discovers some parallelizable loops. An array element or array section is privatizable in a loop if all read operations of an iteration are preceded by a write in the same iteration. Each iteration, therefore each processor can then allocate its own copy of array elements. Recognition of the privatizable array elements or sections is achieved by the computation of array regions [123, 122, 69, 34] or by array dataflow analysis [93, 103]. The scope of the first kind of analysis is wider and interprocedural analyses are possible.

Some iterations of a loop may need the values of array elements defined outside of the loop and may thus prevent the array from being privatizable. This problem is known as the *copy-in* problem and is solved:

- Either by copying the needed values of the array into the local variables, before processing the loop. If there are many iterations that require copy-in or if that number is unknown at compile-time, such copy can add a significant overhead to the computation of the loop and create memory congestion, depending on the architecture of the machine.

- Or by peeling off the iterations that need copy-in and handling them separately. Of course, this is only possible is these iterations are clearly identifiable at compile-time.

Symmetrically, privatized variables may be live after exiting the loop. The same solutions are proposed to handle this *copy-out* problem.

Array expansion

Array expansion eliminates spurious dependences by increasing the number of elements allocated to an array structure. This is achieved by adding new dimensions to array structures or reindexing arrays. To restore the flow of data after expansion, in general an exact array dataflow analysis is required. As the expansion is not limited to an inter-iteration expansion as for privatization, more parallelism can be exposed. As a matter of fact, array privatization can be considered as a particular case of array expansion, as far as static control programs are concerned. Total array expansion transforms a program into *single assignment* (SA) form, that is, array elements are defined only once during program execution. All spurious dependences are then eliminated and the dependence graph is the DFG. This systematic technique has been presented

by Feautrier [49] and scheduling methods taking as input the DFG assume that the program is in SA form. Total array expansion has two obvious drawbacks: The scope of a compile-time total array expansion is restricted up to now to the domain of exact array analysis and in many cases, expansion of all array structures is not necessary or should be avoided because of the large amounts of memory space required.

In the systolic field, most programs are in SA form and the problem of reducing memory usage has been the subject of several studies [26, 125, 60, 25]. The main idea is to use the information provided both by the dependence graph and by the schedule in order to recreate some spurious dependences compatible with the scheduling functions. Recently, Lefebvre and Feautrier [80, 81] described a method that, given a schedule, *a posteriori* recreates the minimal array structures compatible with this schedule. Unlike the previous techniques, such a method does not rely on the old array structures. Once again, the only programs treated so far are static control programs.

2.3.3 Detection of recurrences

Studies of the Perfect Club Benchmarks [17] have shown that recurrence detection (including induction variable recognition) was one of the key techniques that made a parallelizer able to produce efficient code. Most analyses are based on abstract interpretation [33, 66, 67] or pattern matching [129, 124]. Redon [111, 110] described a technique based on array dataflow analysis detecting recurrences on array structures, with any associative operator. This method can be considered as a technique of algorithm recognition in some ways and is able to take advantage of the techniques of induction variable detection. Once recurrences have been detected, some parallelism may be exposed by reordering operations [112]. Moreover, the specificities of the target architecture may allow efficient computation of some recurrences. Finding an appropriate placement for these recurrences, that is generating the parallel code for these recurrences, has been tackled by Barreteau [9].

2.4 Conclusion

There is more in dependence analysis than just dependence testing. Petersen and Padua [98] have shown that complex dependence tests do not extract much more parallelism than simple tests, and several studies have revealed that an efficient parallelization required a deeper knowledge of the data flow than what is provided by a dependence test. In spite of their cost and the small number of programs on which they apply, value-based dependence analyses based on a polyhedron abstraction enable some of the most precise techniques of parallelization.

Chapter 3

Exact Array Dataflow Analysis

This chapter presents techniques for the computation of exact array dataflow dependences. For each operation reading a memory cell, such analysis finds the latest write into this cell which precedes the read. The formalism and the approach used by the polyhedral method are at the heart of most of the analyses proposed in this thesis. So as to reduce the cost of the polyhedral method, a simpler algorithm taking advantage of the specificities of most usual programs is described in Section 3.3. Both methods are then compared to other exact analyses in Section 3.4 and the chapter is closed by a discussion about the possibilities to extend such techniques beyond the scope of the polyhedral model.

3.1 Program Model

In order to produce an exact result, this method applies only to programs fragments fulfilling the following conditions:

1. The only data structures are of base types (integers, reals, etc.) and arrays thereof.

2. The only control structures are the sequence, the `if..then..else` construct and the `do` loop.

3. Loop bounds, predicates of conditionals and array subscripts are quasi-affine expressions of structure parameters and loop counters. An expression is quasi-affine with respect to some variables if it involves affine combinations, Euclidean divisions by an integer constant and modulos of integer constant remainder with respect to these variables.

4. Basic statements are assignments to scalars or array elements.

5. No pointer, `EQUIVALENCE` or aliasing is allowed.

Such a program is called a *static control* program.

3.2 Polyhedral Method

Feautrier defined a method based on a polyhedral representation to compute exact dataflow dependences [49]. It is described thereafter with three possible optimizations. This method is implemented in the parallelizer PAF [97].

3.2.1 Formal Solution

Let $\langle \mathtt{R}, y\rangle$ be an operation that reads the element $\mathtt{A}(g(y))$ of an array $\mathtt{A}$. We are interested in finding the source of the value of $\mathtt{A}(g(y))$ for $\langle \mathtt{R}, y\rangle$. As a matter of fact, given the syntactical expression $\mathtt{A}(g(y))$ and statement $\mathtt{R}$, we look for a source function depending on the values of structure parameters and of y. Let $\mathtt{S}_1, \ldots, \mathtt{S}_n$ be the statements assigning a value to $\mathtt{A}$ and let $x_1, \ldots, x_n$ denote their iteration vectors. $\mathtt{S}_i$ is of the form:

$$\mathtt{A}(f_i(x_i)) = \cdots .$$

Let $\mathsf{Q}_i(y)$ denote the set of iteration vectors x_i such that $\langle \mathtt{S}_i, x_i\rangle$ writes $\mathtt{A}(g(y))$. A candidate source $\langle \mathtt{S}_i, x\rangle$, with $x \in \mathsf{Q}_i(y)$ has to satisfy several constraints:

Existence Predicate: $\langle \mathtt{S}_i, x\rangle$ is a valid operation:

$$x \in \mathsf{I}(\mathtt{S}_i).$$

Subscript Equation: $\langle \mathtt{S}_i, x\rangle$ and $\langle \mathtt{R}, y\rangle$ access the same memory cell:

$$f_i(x) = g(y).$$

Sequencing Predicate: $\langle \mathtt{S}_i, x\rangle$ is executed before $\langle \mathtt{R}, y\rangle$:

$$\langle \mathtt{S}_i, x\rangle \prec \langle \mathtt{R}, y\rangle .$$

Environment: The source has to be computed under the hypothesis that $\langle \mathtt{R}, y\rangle$ is a valid operation:

$$y \in \mathsf{I}(\mathtt{R}). \tag{3.1}$$

From this we deduce the definition of $\mathsf{Q}_i(y)$:

$$\mathsf{Q}_i(y) = \{x \,|x \in \mathsf{I}(\mathtt{S}_i), f_i(x) = g(y), \langle \mathtt{S}_i, x\rangle \prec \langle \mathtt{R}, y\rangle\}\,. \tag{3.2}$$

The operation producing the value read by $\mathtt{A}(g(y))$ for $\langle \mathtt{R}, y\rangle$ is the latest operation writing into this array cell. Hence, the source function is:

$$\sigma(y) = \max \bigcup_{i=1}^{n} \{\langle \mathtt{S}_i, x\rangle \,|x \in \mathsf{Q}_i(y)\}\,.$$

This definition of a source function for a sink and an array structure is independent of the abstraction used for the dependences. The following section explains how, in a polyhedral abstraction, such function can be computed.

3.2.2 Evaluation Techniques

The existence predicate, subscript equation and environment are conjunctions of quasi-affine inequalities and the sequencing predicate is a disjunction of such inequalities. Therefore the set $\mathsf{Q}_i(y)$ is a union of sets defined by quasi-affine inequalities:

$$\mathsf{Q}_i(y) = \bigcup_{p=0}^{d_{\mathtt{S}_i\mathtt{R}}} \mathsf{Q}_i^p(y),$$

where:

$$\mathsf{Q}_i^p(y) = \{x \,|x \in \mathsf{I}(\mathtt{S}_i), f_i(x) = g(y), \langle \mathtt{S}_i, x\rangle \prec_p \langle \mathtt{R}, y\rangle\}\,. \tag{3.3}$$

Array subscripts with modulos or Euclidean divisions give rise to a quasi-affine subscript equation $f_i(x) = g(y)$. Modulos and Euclidean divisions appearing in loop bounds or conditional predicates give rise to a quasi-affine existence predicate.

The following property then makes the computation of σ in two steps possible.

Property 3.1 *For any subsets* E_i *of a totally ordered set* E*:*

$$\max \bigcup_{1\le i\le n} \mathsf{E}_i = \max_{1\le i\le n} \max \mathsf{E}_i.$$

Proof The proof is trivial provided that we define a special element, denoted $-$, such that $- = \max\emptyset$ and $-$ is lower than any other element of any set E_i.

The first step is the computation of the functions:

$$\varsigma_i^p(y) = \langle \mathsf{S}_i, \max \mathsf{Q}_i^p(y) \rangle \,, \tag{3.4}$$

in the environment given by (3.1). Then, thanks to Property 3.1, the expression of the source is:

$$\sigma(y) = \max_{1 \le i \le n} \max_{0 \le p \le d_{\mathsf{S}_i \mathsf{R}}} \varsigma_i^p(y). \tag{3.5}$$

The computation of the $\varsigma_i^p(y)$'s boils down to a problem of parametric integer linear programming. Indeed, for each Euclidean division $\lfloor \frac{r}{s} \rfloor$ appearing in the definition of $\mathsf{Q}_i^p(y)$, a wild-card variable q is added to the problem as well as the constraint $0 \le r - qs < q$. q is then the last component of the vector to maximize. After resolution, q is substituted by $\lfloor \frac{r}{s} \rfloor$ in the result. The output of the resolution is a quast.

Definition 3.1 (Quast) *A* Quasi-affine Selection Tree *(quast) is many level conditional in which:*

- *Predicates are tests for the positiveness of quasi-affine forms in the loop counters and structure parameters.*
- *Leaves are either operations whose iteration vector components are quasi-affine, or* $-$*, corresponding to a read of an uninitialized location.*

Several variants of quasts will be introduced all along this thesis.

The resolution of parametric integer programming is handled by PIP [50] in PAF. The quast defining ς_i^p with respect to y will be denoted $\varsigma_i^p(y)$. To finish the computation of the source function, as the maximum is associative, it is enough to know how to compute the maximum of two quasts. Basically, this is done with the help of the following rules (and their symmetric counterparts).

Rule 3.1 $\max(\tau, -) = \tau$.

Rule 3.2 *If* $\tau_2 =$ **if** c **then** τ_3 **else** τ_4 *then:*

$$\max(\tau_1, \tau_2) = \textbf{if } c \textbf{ then } \max(\tau_1, \tau_3) \textbf{ else } \max(\tau_1, \tau_4).$$

Rule 3.3 *If* τ_1 *and* τ_2 *are two operations then:*

$$\max(\tau_1, \tau_2) = \textbf{if } \tau_1 \prec \tau_2 \textbf{ then } \tau_2 \textbf{ else } \tau_1.$$

Note that the $\varsigma_i^p(y)$'s may be combined in any order, it does not change the definition of function σ (its mathematical definition). It can be shown that any repeated application of the three previous rules on the expression $\max(\tau_1, \tau_2)$ produces a quast. This quast is not unique and we say that a quast is a

combination of τ_1 and τ_2 if it can be obtained by repeated application of the previous rules on the expression $\max(\tau_1, \tau_2)$. However, the order in which the rules are applied has an impact on the complexity of the resulting quast. This problem is discussed in the following section.

We illustrate these evaluation techniques with the following example.

```
    program static
    integer a(2*n)

    do i=0,n
S0   a(i)=..
     do j=0,i
S1    a(2*i-2*j)=..
     enddo
R    ..=a(i)
    enddo
```

The source of `a(i)` of statement `R` comes either from statement $\mathtt{S}_0$ or $\mathtt{S}_1$ and the dependence depth is 0 or 1. The sets of candidate sources are:

$$
\begin{array}{rcl}
\mathsf{Q}_0^0(i) & = & \{i' \,|0 \le i' \le n, i' = i, i' < i\}\,, \\
\mathsf{Q}_0^1(i) & = & \{i' \,|0 \le i' \le n, i' = i\}\,, \\
\mathsf{Q}_1^0(i) & = & \{(i', j') \,|0 \le i' \le n, 0 \le j' \le i, 2i' - 2j' = i, i' < i\}\,, \\
\mathsf{Q}_1^1(i) & = & \{(i', j') \,|0 \le i' \le n, 0 \le j' \le i, 2i' - 2j' = i, i' = i\}\,.
\end{array}
$$

Computing the maximum of these sets in the environment $0 \le i \le n$ with PIP gives the definitions of the functions ς_i^p:

$$
\begin{array}{rcl}
\varsigma_0^0(i) & = & - \\
\varsigma_0^1(i) & = & \langle \mathtt{S}_0, i \rangle \\
\varsigma_1^0(i) & = & \left| \begin{array}{ll} \textbf{if } i \bmod 2 \le 0 & \\ \textbf{then} & \left| \begin{array}{ll} \textbf{if } i \bmod 2 \ge 0 & \\ \textbf{then} & \left| \begin{array}{l} \textbf{if } i \ge 2 \\ \textbf{then } \left\langle \mathtt{S}_1, i-1, \left\lfloor \frac{i-2}{2} \right\rfloor \right\rangle \\ \textbf{else } - \end{array} \right. \\ \textbf{else } - & \end{array} \right. \\ \textbf{else } - & \end{array} \right. \\
\varsigma_1^1(i) & = & \left| \begin{array}{ll} \textbf{if } i \bmod 2 \le 0 & \\ \textbf{then} & \left| \begin{array}{l} \textbf{if } i \bmod 2 \ge 0 \\ \textbf{then } \left\langle \mathtt{S}_1, i, \left\lfloor \frac{i}{2} \right\rfloor \right\rangle \\ \textbf{else } - \end{array} \right. \\ \textbf{else } - & \end{array} \right. \quad .
\end{array}
$$

Finally, the source $\sigma(i)$ is obtained by combining in any order the above quasts. Applying straightforwardly the three rules given in this section of the expression $\max(\varsigma_0^0(i), \max(\varsigma_0^1(i), \max(\varsigma_1^0(i), \varsigma_1^1(i))))$ leads to the unsimplified expression:

$$\sigma(y) = \left|\begin{array}{l} \textbf{if } i \bmod 2 = 0 \\ \textbf{then} \left|\begin{array}{l} \textbf{if } i \geq 2 \\ \textbf{then} \left|\begin{array}{l} \textbf{if } i \bmod 2 = 0 \\ \textbf{then } \langle \mathtt{S}_1, i, \lfloor \frac{i}{2} \rfloor \rangle \\ \textbf{else } \langle \mathtt{S}_0, i \rangle \end{array}\right. \\ \textbf{else} \left|\begin{array}{l} \textbf{if } i \bmod 2 = 0 \\ \textbf{then } \langle \mathtt{S}_1, i, \lfloor \frac{i}{2} \rfloor \rangle \\ \textbf{else } \langle \mathtt{S}_0, i \rangle \end{array}\right. \end{array}\right. \\ \textbf{else} \left|\begin{array}{l} \textbf{if } i \bmod 2 = 0 \\ \textbf{then } \langle \mathtt{S}_0, i \rangle \\ \textbf{else } \langle \mathtt{S}_0, i \rangle \end{array}\right. \end{array}\right.$$

Note that, for the sake of clarity, the equalities $i \bmod 2 = 0$ are written in the quast as an equality instead of two tests for positiveness.

3.2.3 Optimizations

In general, the computation of the quasts $\varsigma_i^p(y)$ and the construction of the source are expensive operations. Indeed, the resolution of an integer program is an NP-complete problem [116] and Rule 3.2 of page 74 shows that the size of a combination grows exponentially with the number of quast to be combined. We present thereafter three optimization techniques so as to improve the efficiency of the polyhedral method. Although the first two optimizations have already been described thoroughly by other authors, we give a brief reminder of the formalism used so as to be able to adapt these techniques to approximate dataflow analyses in Chapter 4.

Simplifications

The quast obtained by the three preceding rules may contain infeasible paths. Determining which path is feasible has two advantages: not only does it help in reducing the size of the quast but also it enables the application of compile-time techniques such as scheduling or privatization (see Chapter 5).

The notion of context of a node is introduced so as to enable the simplification of quasts, node by node.

Definition 3.2 (Context) *The* context *of a node of a quast is the conjunction of the predicates or negation of predicates that are true on the path from the root to this node.*

Simplification is performed on-the-fly during the combination of quasts by application of the rules:

Rule 3.4 *Let* $\tau =$ **if** c **then** τ_1 **else** τ_2 *be a subtree of a quast,* p *its context and* e *the current environment. If* $e \wedge p \wedge c$ *is unsatisfiable then replace* τ *with* τ_2*. If* $e \wedge p \wedge \neg c$ *is unsatisfiable then replace* τ *with* τ_1*.*

Rule 3.5 *Quasts of the form* **if** c **then** τ **else** τ *are rewritten* τ.

The predicates of a quast are tests for the positiveness of quasi-affine forms. Obviously the negation of such a predicate is still a predicate of this kind, therefore the satisfiability test needed for simplifications has to be done on a system of such predicates.

Simplification may limit the exponential explosion introduced by Rule 3.2 page 74 but may degrade the performances of the algorithm in the worst case. Whether the aim of simplification is to prune out all infeasible paths or to reduce the size of quasts, several satisfiability tests are proposed, from the resolution of an integer linear program to a simple comparison between parallel constraints (Details in the thesis of Leservot [82]).

> Applied on the source quast obtained for program `static`, the simplification produces the quast:
>
> $$\sigma(i) = \textbf{if } i \bmod 2 = 0 \textbf{ then } \left\langle \mathtt{S}_1, i, \left\lfloor \frac{i}{2} \right\rfloor \right\rangle \textbf{ else } \langle \mathtt{S}_0, i \rangle$$
>
> Note that as the simplification is applied on-the-fly, the previous unsimplified expression of the source never appears.

Lazy algorithm

So far, there is no order in the computation of the quasts $\varsigma_i^p(y)$ and in their combination. Intuitively, as the source of a read is the latest operation writing into the array cell accessed by the read, the latest write operations executed before the read are then more likely to contribute to the source function. Developing this idea first suggested by Feautrier [52], Maslov proposed a *lazy algorithm* [90]. It consists in computing and combining the quasts $\varsigma_i^p(y)$ by order of increasing distance between the write and the read. When combining two quasts $\tau_1(y)$ and $\tau_2(y)$ where the leaves of $\tau_1(y)$ different from $-$ are greater than those of $\tau_2(y)$, Rule 3.3 page 74 ensures that the quast obtained by replacing every occurrence of $-$ in $\tau_1(y)$ by $\tau_2(y)$ is a combination of $\tau_1(y)$ and $\tau_2(y)$. Thus the lazy algorithm simplifies the combination step and possibly, saves some computations of maximum.

Formally, let $<_\varsigma$ be the partial strict order defined on the ς_i^p's by: For all i, j, p, q,

$$p < q \Longrightarrow \varsigma_i^p <_\varsigma \varsigma_j^q,$$
$$\mathtt{S}_i \lhd \mathtt{S}_j \wedge p = d_{\mathtt{S}_i\mathtt{S}_j} \Longrightarrow \varsigma_i^p <_\varsigma \varsigma_j^p.$$

If $\varsigma_i^p <_\varsigma \varsigma_j^q$ then for any leaves $\langle \mathtt{S}_i, x_i^p(y)\rangle$ of $\varsigma_i^p(y)$ and $\left\langle \mathtt{S}_j, x_j^q(y)\right\rangle$ of $\varsigma_j^q(y)$, $\langle \mathtt{S}_i, x_i^p(y)\rangle \prec \left\langle \mathtt{S}_j, x_j^q(y)\right\rangle$. Indeed, when $p < q$, the sequencing predicate ensures that $x_i^p(y)[1..p] = y[1..p] = x_j^q(y)[1..p]$. Moreover if $p = d_{\mathtt{S}_i\mathtt{R}}$ then $\mathtt{S}_i \lhd \mathtt{S}_j$, otherwise $x_i^p(y)[p+1] < y[p+1] = x_j^q(y)[p+1]$. In both cases, it leads to

$\langle \mathtt{S}_i, x_i^p(y)\rangle \prec \langle \mathtt{S}_j, x_j^q(y)\rangle$. Finally, when $p = q, \mathtt{S}_i \lhd \mathtt{S}_j$ and $p = d_{\mathtt{S}_i\mathtt{S}_j}$, the result is obvious.

Hence, the idea of the algorithm is to compute ς_i^p by decreasing $<_\varsigma$-order. Let $\mathsf{R}_1, \ldots, \mathsf{R}_m$ define the smallest sets of functions ς_i^p, such that any element of R_j is $<_\varsigma$-lower than any element of R_k when $j > k$. The lazy algorithm is an incremental algorithm that computes the maximum φ_i of R_i and combines it with the quast ϕ_{i-1} produced by the combination of $\varphi_1, \ldots, \varphi_{i-1}$ (or $-$ if $i = 1$). The combination is fairly simple: ϕ_i is obtained by replacing all occurrences of $-$ in ϕ_{i-1} by φ_i. The algorithm stops at step i if there is no $-$ in ϕ_i or if $i = m$.

As the algorithm is incremental, simplifications need to be done at step i only on the subtrees added to ϕ_{i-1}.

The functions ς_i^p for program `static` are ordered as follows:

$$\varsigma_1^0, \varsigma_0^0 <_\varsigma \varsigma_0^1 <_\varsigma \varsigma_1^1.$$

Therefore we compute first the quast $\phi_1 = \varsigma_1^1(i)$, then $\varsigma_0^1(i)$ and $\phi_2 = \max(\phi_1, \varsigma_0^1(i))$. We would then proceed by the computation of $\varsigma_0^1(i)$ or $\varsigma_0^0(i)$ indifferently but the algorithm then stops since there is no $-$ in ϕ_2.

Parametric read

Given a data structure, the maximum of a set of candidate sources is computed for at most each pair of read and write statements accessing this structure and for each loop surrounding these two statements. As this computation boils down to the resolution of an integer linear program, the overall cost is high. The complete knowledge of the read operation is however not necessary for this resolution. Indeed we show thereafter that the maximum only need to be computed with respect to the iteration vector and the memory cell accessed by the read operation, both considered as parameters.

Let us consider the set of operations whose iteration vector is in $\mathsf{Q}_i^p(y)$. It depends on statement `R` for several reasons:

- Statement `R` gives the length of y. However in $\mathsf{Q}_i^p(y)$, only the sub-vector $y[1..p]$ or $y[1..p+1]$ is used. The range of dependence depths between statement $\mathtt{S}_i$ and any sink can be evaluated syntactically. Hence the maximum may be computed with respect to a vector z of sufficient length, independent of any sink.

- In the sequencing predicate, statement `R` is used in $d_{\mathtt{S}_i\mathtt{R}}$: this syntactic value only contributes to the evaluation of the range of dependence depths.

- In the subscript equation, the term $g(y)$ comes from statement `R`: this term may be replaced by a parameter a for the resolution of the integer linear program.

Thus we can define a set $\tilde{Q}_i^p$ of iteration vectors of candidate sources for a parametric read, defined by: For each $p < d_{S_i}$ in the range of dependence depths,

$$\tilde{Q}_i^p(z, a) = \{x \,|x \in I(S_i), f_i(x) = a, x[1..p] = z[1..p], x[1..p+1] < z[1..p+1]\}\,,$$

and if $p = d_{S_i}$ is in the range of dependence depths,

$$\tilde{Q}_i^p(z, a) = \{x \,|x \in I(S_i), f_i(x) = a, x[1..p] = z[1..p]\}\,.$$

We then compute $\max \tilde{Q}_i^p(z, a)$ and for each sink $\langle R, y\rangle$, substitute a by $g(y)$, $z[1..p]$ by $y[1..p]$ and $z[p+1]$ by $y[p+1]$ if $p+1 \leq d_R$ otherwise by $+\infty$. The cost of the substitution is much lower than the cost of the resolution of an integer linear program. Of course, this method is interesting when there is more than one read statement accessing a data structure.

The size of an unsimplified quast obtained after substitution is in general greater than the size of a quast obtained in a standard way. Indeed, we have shown that one way to compute the quast $\max Q_i^p(y)$ is to compute the quast $\max \tilde{Q}^p(z, a)$ for a parametric read and then substitute to z and a the values associated to the particular read. The substitution may turn some paths of the quast into infeasible paths. A direct computation of $\max Q_i^p(y)$ in general avoids the creation of these infeasible paths (This is the case when a quast is computed with PIP.) So in the worst case, we replace the computation of many quasts by the computation of one quast and by many substitutions and simplifications, which in theory are as expensive as the computations of quasts. However in practice, it is worth computing a quast for a parametric read since simple and inexpensive simplifications are sufficient to prune off all infeasible paths.

3.3 Simplified Polynomial Algorithm

The polyhedral method handles any static control program, but it does not take into account the specificities of most of the real programs. The high cost of the method stems from the algorithm used to resolve linear integer programs and from the computation of the effects produced by other writes on a dependence (that is, the combination of quasts). Several efficient methods [93, 70] have been devised to cope with the first problem. They apply to a subset of static control programs that contain most usual cases and ensure polynomial cost with respect to the program size. We present thereafter a new algorithm that outperforms these methods and applies on a subset of static control programs, followed by a technique that can be used to build polynomial-sized sources.

3.3.1 Another adaptation of Fourier-Motzkin elimination

The algorithms used to compute a maximum or to test the emptiness of a set are in general derived either from the simplex algorithm or from Fourier-Motzkin elimination. There is a lot of literature about both algorithms. (See

[116] for instance). Fourier-Motzkin elimination is very intuitive and it can be easily adapted. Furthermore, Pugh and Wonnacott [104] have shown that this method combined to a simplification of parallel redundant constraints fits particularly well the kind of constraints that arise in a dataflow dependence problem. Indeed, none of the examples studied by Pugh and Wonnacott leads to the expected explosion in the number of final inequalities. They explain their good results by the high frequency of unit coefficients and by the sparsity of constraints.

The new technique we present in the following is an adaptation of Fourier-Motzkin elimination. The aim of this method is to ensure a very low polynomial time complexity to compute the functions ς_i^p's even in the worst case. We adopt the same notations as in Section 3.2 page 3.2 and the program fragment analyzed is supposed to have static control.

3.3.2 Principle of the algorithm

The algorithm calculates the lexicographical maximum of the set $\mathsf{Q}_i^p(y) = \{x \,|f_i(x) = g(y), \langle \mathtt{S}_i, x\rangle \preceq_p \langle \mathtt{R}, y\rangle, x \in \mathsf{I}(\mathtt{S}_i)\}$ in two steps:

- Elimination of affine equations, possibly producing modulo equations (see Section 3.3.3).
- Elimination of the variables $x[k]$ by decreasing order of index, on a system of affine inequalities and modulo equations (see Section 3.3.4).

We then obtain the value of the maximum of $\mathsf{Q}_k^p(y)$ and the conditions for which it exists. The final result is a quast where the context of the only leaf different from bottom is the conjunction of these conditions and the iteration vector of the operation is the value of the maximum.

In order to have an as large as possible subset of programs on which the algorithm applies, the structure of quasts is extended: predicates may now contain functions max or min, applied to quasi-affine forms. When the algorithm does not succeed, one reverts to the previous method.

For any vector of $\mathsf{Q}_k^p(y)$, the first p coordinates are equal to $y[1..p]$. When $p = d_{\mathtt{S}_i\mathtt{R}}$, the quast $\varsigma_i^p(y)$ is obvious, the context of the leaf $\langle \mathtt{S}_i, y[1..p]\rangle$ consists in the conjunction of the constraints defining $\mathsf{Q}_i^p(y)$. Henceforth we suppose $p < d_{\mathtt{S}_i\mathtt{R}}$.

3.3.3 Handling the subscript equation

The equation $f_i(x) = g(y)$ is a system of quasi-affine equations on the components of x. It can be considered as the conjunction of a system of affine equations and of a system of modulo equations. Using the computation of a Hermite reduced form, the system of affine equations can be rewritten:

- $a_k x[i_k] = \sum_{j<i_k} b_{kj} x[j] + c_{i_k}$,

- $0 = c_k$,

where a_k, b_{kj} and c_k are integers and $a_k > 0$. So as to be able to handle the integer constraints produced by this system, it is assumed that either $|a_k| = 1$ or only one b_{kj} is different from zero. We get a system formed by equations of the following forms:

$$x[i_k] = \sum_{j<i_k} b_{kj}x[j] + c_{i_k}, \quad (3.6)$$
$$\text{or} \quad a_k x[i_k] = b_{kj}x[j] + c_{i_k}, \text{ for } j < i_k, \quad (3.7)$$
$$\text{or} \quad 0 = c_k. \quad (3.8)$$

As equations of the third kind do not depend on x, they are added to the context of the solution. $x[i_k]$ or $a_k x[i_k]$ is replaced by its equivalent expression in $x[1..i_k - 1]$ in the system of affine inequalities $x \in \mathsf{I}(\mathtt{S}_i)$. Moreover, when $x[i_k]$ has a non unit coefficient, we add to the system of modulo equations:

$$b_{kj}x[j] + c_{i_k} = 0 \bmod a_k,$$

since $x[i_k]$ is an integer.

If $x[k]$ appears in one or several modulo equations, the coefficients are re-indexed into:

$$b_{hk}x[k] = c_{hk} \bmod a_{hk}. \quad (3.9)$$

Let us consider in Example `static` presented in Section 3.2.2 the dependence between statements $\mathtt{S}_1$ and $\mathtt{R}$ at depth 1. The subscript predicate is $2j' = i$. Thus $2j'$ is replaced by i in $0 \leq j' \leq i$ and the following modulo equation is added to the context:

$$i = 0 \bmod 2.$$

Finally, there exists a dependence if the equation $i = 0 \bmod 2$ is verified. Moreover, the value of j' is then $\lfloor \frac{i}{2} \rfloor$.

3.3.4 Elimination step

Let k be the highest index of a not yet eliminated component of x. We first describe the constraints on $x[k]$ and then show how to eliminate $x[k]$ and obtain the same kind of constraints on $x[k-1]$.

Suppose that $x[k]$ appears in at least one congruence (3.9). Using a variant of the Chinese remainder theorem, this system of equations is equivalent to a system of at most as many modulo equations, where only one depends on x. This equation is then:

$$x[k] = c \bmod a. \quad (3.10)$$

All other modulo equations arising from application of the Chinese remainder theorem are added to the context. When $x[k]$ does not appear in any congruence then we set $c = 0$ and $a = 1$.

The system of inequalities using $x[k]$ can be written:

$$\max_i(l_i(x[1..k-1])) \leq dx[k] \leq \min_j(u_j(x[1..k-1])), \tag{3.11}$$

where the l_i's and u_j's are affine forms and $d > 0$. Our aim is to eliminate $x[k]$ of (3.10) and (3.11). So as to take into account integer constraints on $dx[k]$, we will use the following obvious property:

Property 3.2 *Let u, v and α be integers. Then:*

$$\exists x \in \mathbb{Z} \text{ s.t. } v \leq \alpha x \leq u \Leftrightarrow \alpha \left\lceil \frac{v}{\alpha} \right\rceil \leq u \Leftrightarrow v \leq \alpha \left\lfloor \frac{u}{\alpha} \right\rfloor.$$

(3.10) shows that $d(x[k]-c)$ is a multiple of ad. Hence Property 3.2 proves that $x[k]$ exists if and only if:

$$\max_i(l_i(x[1..k-1])) - cd \leq ad \left\lfloor \frac{\min_j(u_j(x[1..k-1])) - cd}{ad} \right\rfloor, \tag{3.12}$$

or

$$ad \left\lceil \frac{\max_i(l_i(x[1..k-1])) - cd}{ad} \right\rceil \leq \min_j(u_j(x[1..k-1])) - cd. \tag{3.13}$$

The maximum of $x[k]$ is:

$$c + a \left\lfloor \frac{-cd + \min_j(u_j(x[1..k-1]))}{ad} \right\rfloor.$$

Now, if $k > p+1$, we want to transform (3.12) or (3.13) into an upper or lower bound of the kind of (3.11) on $x[k-1]$ so that the elimination can proceed. This cannot be done in general since the inequalities are not affine, due to the presence of floor and ceiling functions. However, in many cases we find that either $ad = 1$, or either the upper or lower bound of (3.11) is constant. In these cases, the elimination can proceed in the same manner for $x[k-1]$. Besides, the elimination of $x[k]$ yields at most a quadratic number of inequalities with respect to the number of bounds on $x[k]$. To keep the algorithm polynomial, it is assumed that either the lower or upper bound of $x[k]$ is unique. The number of inequalities produced is then the number of terms in the lower or upper bounds of $x[k]$.

Before starting the next elimination step, (3.12) or (3.13) is quickly compared with the constraints in the environment and simplified if need be.

The last step of the computation of $\max \mathsf{Q}_i^p$ is the substitution of the maximum of $x[p+1]$ in the expression of the maximum of $x[p+2..n]$, then of the maximum of $x[p+2]$ in the expression of the maximum of $x[p+3]$, and so on.

In order to ensure that the size of the maximum of $x[k]$ is polynomial after substitution, the expression of the upper bound of $x[k]$ in (3.11) must be:

$$u(x[1..k-1]) + \min_j(u_j),$$

where the u_j's are constants and u is an affine form.

> Consider the dependence of depth 0 in Example `static` between statements S_1 and `R`. Eliminating j' produces the system on i':
>
> $$\begin{cases} 0 \le i' \le i-1 \\ 0 \le 2i' - i \le 2i' \end{cases}$$
>
> in the environment: $0 \le i \le n, i \bmod 2 = 0$. In other words,
>
> $$i \le 2i' \le 2(i-1).$$
>
> There exists a value for i' if and only if $i \ge 2$. In this case, the maximum of i' is $\frac{2(i-1)}{2} = i-1$, and by substitution, the maximum of j' is $\frac{2(i-1)-i}{2} = \frac{i-2}{2}$.

3.3.5 Complexity of the algorithm

Let l be the number of loops surrounding the write statement. We consider as a basic operation a multiplication or an addition and we study the worst-case complexity with respect to l.

The linear part of the system $f(x) = g(y)$ has at most l variables and d rows, where d is the dimension of the array. Hence, this system is transformed into a Hermite reduced form in $O(l^2)$ operations. This transformation is performed only once for all dependence depths. The substitution following this step takes $O(l^2)$ operations. As for the elimination itself, note that the projection of $x[k]$ is common to the computation of all $\varsigma_i^p(y)$'s, with $p < k$. Hence it is done only once for all dependence depths. When $x[k]$ is eliminated, $O(l(l-k))$ operations are performed to produce (3.10) from the at most $l-k$ congruences (3.9). Inequalities (3.11) take $O(k(l-k))$ operations to compute. Lastly, the final substitution giving the expression of the maximum is achieved in $O((l-p)^2)$ operations. Therefore, the global cost to compute all $\varsigma_i^p(y)$'s for all p is $O(l^3)$.

We now determine the size of the resulting quast $\varsigma_i^p(y)$. The transformations of the subscript predicate and of (3.9) produce a number of predicates depending on the number of dimensions of the array. The elimination of $x[k]$ produces $O(l-k)$ inequalities therefore when $x[p+1]$ is eliminated, $O(l-p+1)$ predicates make up the context of the solution. The size of the maximum of $x[k]$ is in $O((l-k))$ before substitution and the size of the quast $\varsigma_i^p(y)$ is in $O(l^3)$.

3.3.6 Domain of the algorithm

The domain of the above algorithm is examined in more details in this section and we describe some non-linear cases that can be naturally coped with.

The algorithm applies only if:

1. The subscript equation is equivalent to a system of equations of the kind of (3.6), (3.7) or (3.8) of page 81.
2. During the elimination of a loop index $x[k]$, either the lower or the upper bound of $x[k]$ is unique. (Multiple constant bounds are made unique by using max and min functions.) If the unique bound of $x[k]$ is not constant then there is no integer constraints on $x[k]$.
3. Upper bounds of $x[k]$ only differ by a constant.

These conditions are tested during the execution of the algorithm. If the algorithm is not applicable, we resort to the polyhedral method. Notice that the above conditions are always verified by a two loop nests, which represents 92% of real programs, according to Shen *et al.* [117]. Moreover, the same study shows that more than 82% of linear array references are linear forms with unit coefficients. Hence the first assumption is likely to be verified. In practice, the method computes $\varsigma_i^p(y)$ for p from $d_{\mathtt{S},\mathtt{R}}$ down to the lowest depth for which the conditions are fulfilled.

3.3.7 Extension of the domain to parametric coefficients

Besides, it is interesting to note that this algorithm can handle some cases where coefficients of loop counters are non-linear. Indeed, every step of the resolution can be performed symbolically, provided that the sign of every parametric coefficient is known for the elimination step. In order to handle parametric coefficients, the definition of quasts needs then to be extended again with two functions that can be used in the predicates: gcd, and Euclid, which returns the coefficient u of a in Bezout's equation: $ua + vb = \gcd(a, b)$ when applied to a and b. These two functions may appear in modulo equations resulting from the application of the Chinese remainder theorem. Parametric coefficients arise in a number of situations:

- Rectangular linearized subscripts. In Figure 3.1.a, array `a` is a linearized square matrix.
- Computations with data partitioned into blocks of parametric sizes (tiling), such as the matrix block multiplication.
- Replacement of an induction variable appearing in multiple loops. The program fragment of Figure 3.1.b is an excerpt of program `MDG` from the Perfect Benchmarks. By induction variable substitution, references to array `a` are transformed into $n * j + i - n$.

```
    do i=0, n-1
     do j=i, n-1
      do k=i, n-1
R      ..=a(n*(n-1-i)+n-1-k)
S1     a(n*j+i)=..
      enddo
     enddo
    enddo
```

(a)

```
do i=1,n
 jj=i
 do j=1,m
  a(jj)=..a(jj)
  jj=jj+n
 end do
end do
```

(b)

Figure 3.1: Examples with parametric coefficients

Analysis of programs with parametric coefficients with our method has, however, some limitations: When our technique does not apply, there is no safety net and the analysis is not possible, neither with our simplified method nor with the standard polyhedral technique. Moreover, the source quast produced from a program with parametric coefficients may not be easily used by the usual applications of dataflow analyses. Hence this method fits only some applications, such as program checking (see Chapter 5), and can only handle some particular cases of non-affine constraints.

Let us consider dependences between statements $\mathtt{S}_1$ and R of Figure 3.1.a. In order to find the dependence at depth p, we have to compute the lexicographic maximum of (i', j', k') verifying:

- $0 \leq i', j', k' \leq n-1$, the existence predicate;
- $i' + nj' = n^2 - ni - 1 - k$, the subscript equation;
- $i' < i$, $i' = i \wedge j' < j$, or $i' = i \wedge j' = j \wedge k' < k$, depending on the dependence depth.

For these dependences, the environment is the conjunction of $0 \leq i \leq n-1, i \leq j \leq n-1$ and $i \leq k \leq n-1$.

The first step of the algorithm is to transform the subscript equation: it gives the expression of j': $n - i - \frac{i'+k+1}{n}$ and produces the constraint $i' + k + 1 = 0 \bmod n$. j' is replaced by its expression in the constraints in which it appears:

$$ni' \leq n^2 - ni - i' - k - 1 \leq n^2 - n,$$

and

$$n^2 - ni - i' - k - 1 \leq nj - n,$$

when the dependence is of depth 1.

The elimination of k' leads to the constraint $i \leq n-1$, which is implied by the environment. Thus the expression of $\varsigma_2^2(i,j,k)$ is:

$$\varsigma_2^2(i,j,k) = \left| \begin{array}{l} \textbf{if } 2ni \leq n^2 - i - k - 1 \\ \textbf{then} \left| \begin{array}{l} \textbf{if } n \leq (n+1)i + k + 1 \\ \textbf{then} \left| \begin{array}{l} \textbf{if } i + k = -1 \bmod n \\ \textbf{then } \langle \mathtt{S}_2, (i,j,n-1) \rangle \\ \textbf{else } - \end{array} \right. \\ \textbf{else } - \end{array} \right. \\ \textbf{else } - \end{array} \right.$$

Likewise, the expression of $\varsigma_2^1(i,j,k)$ is:

$$\varsigma_2^1(i,j,k) = \left| \begin{array}{l} \textbf{if } 2ni \leq n^2 - i - k - 1 \\ \textbf{then} \left| \begin{array}{l} \textbf{if } n \leq (n+1)i + k + 1 \\ \textbf{then} \left| \begin{array}{l} \textbf{if } i + k = -1 \bmod n \\ \textbf{then} \left| \begin{array}{l} \textbf{if } (n+1)(n-i) - k - 1 \leq nj \\ \textbf{then } \langle \mathtt{S}_2, (i,j,n-1) \rangle \\ \textbf{else } - \end{array} \right. \\ \textbf{else } - \end{array} \right. \\ \textbf{else } - \end{array} \right. \\ \textbf{else } - \end{array} \right.$$

Lastly, for the dependence of depth 0 the constraints on i' are:

$$\begin{array}{c} (n+1)i' \leq n^2 - ni - k - 1 \\ 0 \leq i' \leq i - 1 \\ n - ni - k - 1 \leq i' \\ i' = -k - 1 \bmod n \end{array}$$

As $n - ni - k - 1 \leq 0$ when $0 \leq i - 1$, the constraint $n - ni - k - 1 \leq i'$ can be simplified. The elimination of i' produces:

$$(n+1)\left\lceil \frac{k+1}{n} \right\rceil \leq n - i + k + 1,$$

$$n \left\lceil \frac{k+1}{n} \right\rceil \leq i + k,$$

and the maximum of i' is:

$$\min\left(n \left\lfloor \frac{i+k}{n} \right\rfloor - k - 1, \left\lfloor \frac{n(n-i) - k - 1}{n+1} \right\rfloor \right).$$

Finally, the expression of $\varsigma_2^0(i,j,k)$ is:

$$\varsigma_2^0(i,j,k) = \left| \begin{array}{l} \textbf{if } (n+1)\left\lceil \frac{k+1}{n} \right\rceil \leq n - i + k + 1 \\ \textbf{then} \left| \begin{array}{l} \textbf{if } n \left\lceil \frac{k+1}{n} \right\rceil \leq i + k \\ \textbf{then } \left\langle \mathtt{S}_2, \min\left(n \left\lfloor \frac{i+k}{n} \right\rfloor - k - 1, \left\lfloor \frac{n(n-i)-k-1}{n+1} \right\rfloor \right) \right\rangle \\ \textbf{else } - \end{array} \right. \\ \textbf{else } - \end{array} \right.$$

Program	Using PIP		Our method
	Calls for dep.	Pivoting steps	Pivoting steps
`ffc`	4	7170	513
`matmul`	4	4222	229
`chales`	18	10882	717
`choles`	12	9217	614
`burg2`	26	20332	2698
`mod1`	2	1826	382
`mod3`	2	520	13
`van4`	2	5376	588
`yacobi`	105	594897	79649
`gosser`	13	10932	713
`lczos`	60	32773	2554
`relax`	6	25766	2931
`across`	5	468	64

Table 3.1: Comparison of the number of pivots

3.3.8 Performances

This polynomial algorithm has been implemented in `Le_Lisp` inside the project PAF. The program PIP written in C is used in its multi-precision version[27] when our method cannot be applied and for simplification. The implementation only handles up to now constant coefficients. The optimizations described in Section 3.2.3 are applied.

The method proves to be efficient on numerous programs. For all programs in Table 3.1, the maxima of candidate source sets can be computed without resorting to PIP. As for time performances, compared to those of the method using PIP, they are not really significant due to the difference between the performances of the two languages. Hardly any gain of time is observed between the simplified algorithm and the polyhedral method. So we take instead the number of pivoting steps occurring during the analyses as a measure of performances. Table 3.1 shows the number of calls to PIP in the method using only PIP and the total number of pivoting steps in both methods. The first two columns show the number of PIP calls in the method using only PIP and the total number of pivoting steps. The third column is the number of equivalent pivoting steps with the simplified technique. It appears that we gain a factor of 5 up to 40 in the number of pivoting steps when using the polynomial method instead of the general technique.

3.3.9 Alternative technique for combining quasts

Direct application of Rules 3.2 and 3.3 of Section 3.2.2 for the computation of the quast $\sigma(y)$ leads in the worst case to an explosion of its size. In this section, we propose a method to build a quast $\sigma(y)$ of polynomial size, with respect to the number of write statements. The general idea is to compute a quast in which internal nodes define a union of polyhedra instead of a polyhedron. In practice, this means that the functions min and max are used in the definition of $\sigma(y)$, in the same manner as they are used to define $\varsigma_i^p(y)$ in Section 3.3.1. For the sake of clarity, we will use the operators $\wedge$ and $\vee$ on conditionals instead of max and min. This technique yields a quast that is in general much more difficult to simplify than quasts with simple predicates and therefore is not appropriate for all applications. We tackle the problem of simplification at the end of this section.

Construction of $\sigma(y)$

It is supposed that every quast $\varsigma_i^p(y)$ has only one leaf different from $-$. This hypothesis is verified for the expressions of $\varsigma_i^p(y)$ produced by the algorithm of Section 3.3.1. In other words, if the quasts $\varsigma_i^p(y)$, for all p, i are re-indexed into φ_k then every φ_k must be of the following form:

$$\textbf{if } c_k \textbf{ then } \tau_k \textbf{ else } -,$$

where c_k is a predicate defining a polyhedron or a union of polyhedra and τ_k is an operation.

Let us consider for instance φ_1. A possible combination of φ_1 and φ_2 is a quast of the form:

$$
\begin{array}{|l}
\textbf{if } c_1 \\
\textbf{then} \begin{array}{|l}
\textbf{if } c_2 \\
\textbf{then} \begin{array}{|l}
\textbf{if } \tau_2 \preceq \tau_1 \\
\textbf{then } \tau_1 \\
\textbf{else } \tau_2
\end{array} \\
\textbf{else } \tau_1
\end{array} \\
\textbf{else } \ldots
\end{array}
$$

where the ellipsis does not contain any occurrence of τ_1. The predicate $c_1 \wedge (\neg c_2 \vee \tau_2 \preceq \tau_1)$ is the condition for which τ_1 is the latest operation among all possible operations of φ_1 and φ_2. The previous quast can be rewritten as:

$$\textbf{if } c_1 \wedge (\neg c_2 \vee \tau_2 \preceq \tau_1) \textbf{ then } \tau_1 \textbf{ else } \ldots$$

By immediate recurrence, it can be shown that the condition for which τ_1 is the source is:

$$c_1 \bigwedge_{k \neq 1} (\neg c_k \vee \tau_k \preceq \tau_1)\,.$$

A similar formula can be obtained for each τ_k, by permutation. Note that this predicate is very similar to a predicate defining a dependence relation in Pugh and Wonnacott's formalism. A more detailed comparison is given in Section 3.4.1.

An naive construction of the source would give:

$$\sigma(y) = \left| \begin{array}{l} \textbf{if } c_1 \bigwedge_{k\neq 1} (\neg c_k \vee \tau_k \preceq \tau_1) \\ \textbf{then } \tau_1 \\ \textbf{else} \left| \begin{array}{l} \textbf{if } c_2 \bigwedge_{k\neq 2} (\neg c_k \vee \tau_k \preceq \tau_2) \\ \textbf{then } \tau_2 \\ \textbf{else} \ldots \left| \begin{array}{l} \textbf{if } c_i \bigwedge_{k\neq i} (\neg c_k \vee \tau_k \preceq \tau_i) \\ \textbf{then } \tau_i \\ \textbf{else} \ldots \end{array} \right. \end{array} \right. \end{array} \right. \tag{3.14}$$

As a matter of fact, the average size of the predicates of this quast can be divided by two, thanks to a simplification. Indeed, consider the context of τ_2, it is the conjunction of two predicates:

$$\left(\neg c_1 \bigvee_{k>1} (c_k \wedge \tau_1 \preceq \tau_k)\right) \wedge \left(c_2 \wedge (\neg c_1 \vee \tau_1 \preceq \tau_2) \bigwedge_{k\neq 2} (\neg c_k \vee \tau_k \preceq \tau_2)\right).$$

Suppose this predicate is true. If $\neg c_1$ or $c_2 \wedge \tau_1 \preceq \tau_2$ in the first conjunct is true then the clause $\neg c_1 \vee \tau_1 \preceq \tau_2$ of the second conjunct is verified. If $c_k \wedge \tau_1 \preceq \tau_k$, for a $k > 2$, as the clause $\neg c_k \vee \tau_k \preceq \tau_2$ in the second conjunct is verified, it implies that $\tau_1 \preceq \tau_k \preceq \tau_2$. Therefore $\neg c_1 \vee \tau_1 \preceq \tau_2$. Consequently, the predicate $c_2 \bigwedge_{k\neq 2} (\neg c_k \vee \tau_k \preceq \tau_2)$ in (3.14) can be simplified: $c_2 \bigwedge_{k>2} (\neg c_k \vee \tau_k \preceq \tau_2)$. More generally, the predicate $c_i \bigwedge_{k\neq i} (\neg c_k \vee \tau_k \preceq \tau_i)$ is rewritten as $c_i \bigwedge_{k>i} (\neg c_k \vee \tau_k \preceq \tau_i)$. Thus the final expression of σ is:

$$\sigma(y) = \left| \begin{array}{l} \textbf{if } c_1 \bigwedge_{k>1} (\neg c_k \vee \tau_k \preceq \tau_1) \\ \textbf{then } \tau_1 \\ \textbf{else} \left| \begin{array}{l} \textbf{if } c_2 \bigwedge_{k>2} (\neg c_k \vee \tau_k \preceq \tau_2) \\ \textbf{then } \tau_2 \\ \textbf{else} \ldots \left| \begin{array}{l} \textbf{if } c_i \bigwedge_{k>i} (\neg c_k \vee \tau_k \preceq \tau_i) \\ \textbf{then } \tau_i \\ \textbf{else} \ldots \end{array} \right. \end{array} \right. \end{array} \right.$$

A more complete simplification can be obtained by application of the techniques proposed by Pugh and Wonnacott [103]. However this only simplifies the predicate of each internal node independently of its context and this slows down the method.

Complexity of the combination

If l is the maximum number of nested loops in the program and m the number of write statements then there are at most lm quasts $\varsigma_i^p(y)$. The predicate

$c_i \bigwedge_{k>i} (\neg c_k \vee \tau_k \preceq \tau_i)$ consists in $\mathcal{O}(lm - i)$ predicates c_k or $\tau_k \preceq \tau_i$ of size in $O(l)$. Therefore $\sigma(y)$ has lm internal nodes and its size is $O(l^2 m)$.

3.4 Related Work

Most array dataflow dependence analyses are approximate. They will be detailed in Section 4.8. The first studies about exact analyses were due to Brandes [23] and Feautrier [49]. The main issues of such analyses are their complexity and their small domain of application. We give in the following a description of the different techniques proposed to improve one or the other and a detailed comparison with the methods we have described in previous sections.

3.4.1 Pugh and Wonnacott's approach

Pugh and Wonnacott [103, 129] proposed an approach for exact array dataflow dependence analysis that is based on a formalism similar to Brandes'. Their method applies on any static control program and consists in building a relation between operations that are in dependence, that is, given a write and read statement, a relation between the iteration vectors of the operations in dependence [101]. Using the same notations as in previous section, the memory-based dependence relation between statements $\mathtt{S}_i$ and $\mathtt{R}$ is defined by:

$$\mathsf{M}_{\mathtt{S}_i \to \mathtt{R}} = \{x_i \to y \,|x_i \in \mathsf{I}(\mathtt{S}_i) \wedge y \in \mathsf{I}(\mathtt{R}) \wedge \langle \mathtt{S}_i, x_i\rangle \prec \langle \mathtt{R}, y\rangle \wedge f_i(x_i) = g(y)\} .$$

Definitions of dependence relations are represented by formulae in Presburger arithmetic. (First order logic applied to the theory of addition on integers). The value-based dependence between $\mathtt{S}_i$ and $\mathtt{R}$ is:

$$\mathsf{M}_{\mathtt{S}_i \to \mathtt{R}} - \bigcup_j \mathsf{M}_{\mathtt{S}_i \to \mathtt{S}_j} \circ \mathsf{M}_{\mathtt{S}_j \to \mathtt{R}} .$$

In Presburger arithmetic, this is equivalent to:

$$\left\{ x_i \to y \,\middle|\, \begin{array}{l} x_i \in \mathsf{I}(\mathtt{S}_i) \wedge y \in \mathsf{I}(\mathtt{R}) \wedge \langle \mathtt{S}_i, x_i\rangle \prec \langle \mathtt{R}, y\rangle \wedge f_i(x_i) = g(y) \\ \bigwedge_j \neg(\exists x_j, x_j \in \mathsf{I}(\mathtt{S}_j) \wedge \langle \mathtt{S}_i, x_i\rangle \prec \langle \mathtt{S}_j, x_j\rangle \prec \langle \mathtt{R}, y\rangle \wedge f_j(x_j) = g(y)). \end{array} \right\} \quad (3.15)$$

This formula is equivalent to the expression of a maximum. Indeed, given a set A:

$$\{\max \mathsf{A}\} = \left\{x \,\middle|x \in \mathsf{A} \wedge \neg(\exists x', x' \in \mathsf{A} \wedge x \ll x')\right\} .$$

Computing value-based dependences

Basically, the computation of value-based dependence relations is performed in two steps:

- The ordering of memory-dependences into groups, so as to optimize the computation. Partial covers are dependences that kill or cover partially

the dependences from more distant writes. The computation of groups of partial covers is similar to the lazy technique. Pugh and Wonnacott also use some other memory-based dependence information as well as distance/direction dependence vectors to improve the efficiency of their algorithm.

- The computation and simplification of dependence relations. Existentially quantified variables are first eliminated by the Fourier-Motzkin algorithm. Then, the formula is put into disjunctive normal form. To prevent the explosion in size of the formula resulting from this transformation, negated predicates are simplified using the **gist** operator. Intuitively, **gist** p **given** q are the constraints of p that are not implied by q. Pugh and Wonnacott show that $q \wedge \neg p$ is equivalent to $q \wedge \neg(\mathbf{gist}\ p\ \mathbf{given}\ q)$. This simplification using **gist** resort to many techniques of increasing complexity and possibly to a satisfiability test handled by the Omega Test [102].

Importance of the representation

Pugh and Wonnacott have pointed out that the representation of the source with a quast might lead to an exponential number of leaves whereas a Presburger formula would not blow-up [103]. Indeed, the rules used to combine two quasts entail that each test of one of the quast that is irrelevant for the other and that cannot be simplified may lead to the duplication of the latter quast. However, simplification rules limit in many case such a combinatorial explosion.

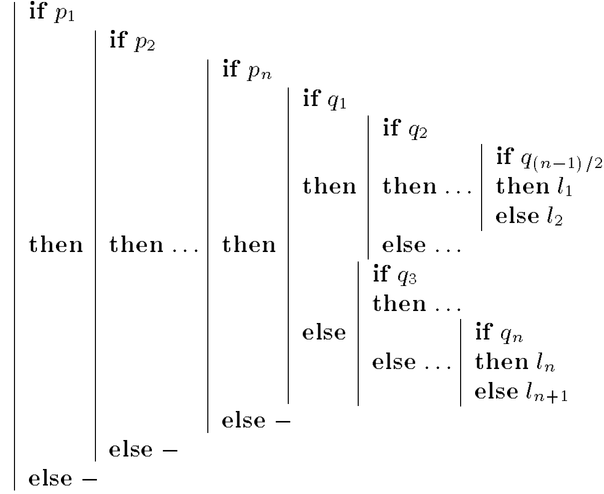
In spite of this drawback, the representation of the source function by a quast has two advantages:

- Quasts can be used at run-time or placed directly into the code. This property is used for memory expansion for instance (see Section 5.4 on page 166).

- When combining quasts, tests that belong to different quasts appear only once in the result of the combination. Thus this can save some space

therefore some time in simplifications.

Consider the following quast:



We suppose that no further simplification is possible and all leaves l_i are operations of different statements. This quast has $2n$ internal nodes. To represent the same source with dependence relations , one has to build as many dependence relations as statements in the quast, that is, $n+1$. Each relation is defined by the conjunction of the $n \times \log_2(n)$ predicates on the path from the root to the leaf. Therefore, the source is represented by $n(n+1)\log_2(n)$ predicates with dependence relations, and only $2n$ predicates with a quast.

Besides, memory-based dependence relations are defined as sets. This approach leads to the definition of value-based dependence relations for every write statement (3.15), with existentially quantified variables in every negated clause. The projection of these variables is performed by Fourier-Motzkin elimination and can be expensive. This cost is mainly avoided with quasts by computing once and for all the maxima $\varsigma_i^p(y)$ and then comparing operations, instead of sets of operations.

Comparison with our simplified algorithm

The simplified algorithm outperforms on its domain the general method described by Pugh and Wonnacott. To compare both techniques, we consider a value-based dependence when there is only one write statement. If l is the number of loops surrounding the write, the elimination of existentially quantified variables in one of the negated clause is performed in $O(l^2)$ operations in the worst case. Then, as there are at most l negated clauses and since value-based dependence relations are computed for each dependence depth separately, the overall cost to compute dependence relations for all depths is in $O(l^4)$, whereas it is in $O(l^3)$ for the simplified algorithm.

Moreover, Pugh and Wonnacott do no use max and min functions, thus some disjunctions may appear in the expression of dependence relations each time a negated clause is put into DNF, increasing the cost of the overall computation. Finally, the handling of equations by the Omega Test does not enable parametric coefficients for loop counters.

3.4.2 Maslov's approach

Maslov described his lazy algorithm with a formalism of dependence relations [90], similar to the representation introduced by Pugh. However, Maslov computes maxima over dependence relations. He describes two separate algorithms, the first one, $Relmax1$, computes the maximum of the write iteration vectors for one dependence relation, and the second one, $Relmax2$, computes the maximum of write iteration vectors for several dependence relations.

The first algorithm uses Fourier-Motzkin elimination and adds wild-card variables in order to handle integer constraints. This technique provides a result that is less precise than the output given by a simplex algorithm but is more general than our simplified method. If l is the number of loops surrounding the write statement, $Relmax1$ takes $O(l^4)$ operations, even on the subset of problems handled by our technique, which is more complex than what we obtained.

The second method is equivalent to our method of combination of quasts, transposed to a representation of dependence relation. It has the advantages and shortcomings due to its representation.

3.4.3 Maydan et al's approach

Maydan, Amarasinghe and Lam [93] defined an algorithm for computing value-based dependences for only a subset of static control programs. The algorithm applies if:

- Write operations do not self-interfere: self-interference means that the subscript function f is bijective with respect to the loop indices used by the write. A loop index is said to be unused if it does not appear in f nor in the bounds of any used loop index.

- There are no partial degeneracies, that is, loop upper bounds are either always greater or always lower than lower bounds. Partial degeneracies can be in some cases eliminated by code transformation [5].

The result of their method is a *Last Write Tree* (LWT), which is similar to a quast. A LWT represents a direct dependence when there is only one write statement and it takes into account all dependence depths. When there are

multiple writes, Maydan proposed a method [92] that is similar to ours (see Section 3.2.2). The computation of the LWT is in two steps:

- Computation of the LWT without the unused loop indices. All tests in the tree decide of the dependence depth. As the subscript function f is bijective, the loop indices of the write are easily found. There is no test concerning the execution of loops, since there are no partial degeneracies.
- Restoration of the unused indices.

The domain of application of their method is strictly included in the domain of our simplified algorithm. Moreover, when Maydan's techniques apply, it can easily be shown that our algorithm requires $O(l^2)$ operations to compute all of the ς_i^p's whereas the computation of the equivalent LWT takes $O(l^3)$ operations, where l is the number of loops surrounding the write. The LWT has then $O(l^2)$ nodes, whereas the size of a quast $\varsigma_i^p(y)$ is then of $O(l-p)$. Maydan shows that its algorithms applies on about 97% of the array writes with affine index expression of the Perfect Club programs. However it does not apply in the case of linearized expressions.

3.4.4 Heckler and Thiele's approach

Heckler and Thiele [70] proposed also an efficient method for computing value-based dependences on a subset of static control programs. The key idea of their algorithm is to calculate the solutions of the system of affine Diophantine equations coming from the subscript predicate.

Given a write and a read statements, the solutions of the subscript predicate are written in a parametric form and then the value of the parameters are computed so as to correspond to the lexicographic maximum of the iteration vector verifying the existence and sequencing predicates. Heckler and Thiele assume strong conditions on this system of inequalities in order to find very easily a correct value for each parameter, without resorting to Fourier-Motzkin elimination. When these conditions are not met, they resort to PIP to find the correct values of the parameters. As there are fewer parameters than loop indices, their method is in any case faster than Feautrier's.

The restrictions on the input problem are stronger than those used for our simplified algorithm. The complexity of Heckler and Thiele's algorithm is the complexity of the computation of a Hermite or Smith normal form. When multiple writes occur, they resort to a method similar to ours. However, they do not handle structural parameters nor linearized subscripts.

3.4.5 Going beyond static control programs

Several techniques extend the scope of exact array dataflow analysis while preserving the exactness. Such is the case of some induction variable substitutions,

constant propagation and restructuration of `while` loops [4]. Berthou [15] studied a technique so as to limit the range of statements to analyze in real programs with an exact analysis, without losing accuracy.

In Section 3.3.6 we proposed an adaptation of our simplified algorithm so as to handle linearized subscripts. Maslov [89] suggests to use a simple test that detects linearization when loops are rectangular. The problem can then be treated as if the program was a static control program. Maslov and Pugh [91] describe a much more general algorithm to cope with some polynomial array subscripts. Their technique is based on factorization of the constraints and on the *affinization* of quadratic (in)equalities. Affinization consists in replacing a test for the positiveness of a polynomial function by the system of its tangent inequalities in all integer points of the iteration domain. Note that this process is finite since the iteration domain of static control programs is finite and known at compile-time. Parametric iteration domains do not seem to be handled by this approach. This method is able to cope with triangular linearization (subscripts of the form $\mathtt{i} * (\mathtt{i} - \mathtt{1})/\mathtt{2} + \mathtt{j}$ obtained by linearization of a triangular matrix).

Beside polynomial expressions with respect to structure parameters or loop indices, Maslov [90] also describes a way to deal with symbolic constants, which are variables that are not assigned within the scope of loop range studied. This technique can easily be applied to our method. Finally, Leservot [82] proposed an extension of exact array dependence analysis to *generalized instructions*. Generalized instructions are instructions that write and read many array elements, such as for instance fragments of code or procedure calls. The effects of a generalized instruction on array structures are summarized in array regions.

3.5 Conclusion

Exact array dataflow analysis is a powerful technique that computes exactly direct dependences on static control programs. The main issues of this kind of analysis are its high complexity and its small domain of application. Several solutions have been proposed so as to reduce the high complexity of the method, by further reducing the domain of application to the most usual cases of static control programs. As for the limitation on the input programs, the intra-procedural limitation has been partially removed and non-affine constraints can be handled in some cases, when they can be either ignored or transformed equivalently into affine constraints.

Chapter 4

Approximate Array Dataflow Analysis

When non-affine constraints appear in the expression of the source (3.2), the techniques of computation presented in the previous chapter do no longer work. We present in this chapter a general framework to compute an approximate dataflow graph when constraints of this kind appear.

This chapter is organized as follows: Section 4.2 describes the features of real programs that make an exact analysis fail, presents four motivating examples illustrating these features and gives an intuition of the approximate analysis. Section 4.3 introduces our framework. Then in Section 4.4 we propose several methods to find properties on non-affine constraints and describe in Sections 4.5-4.6 how to integrate them in the computation of the sources. We conclude with the presentation of our prototype, Caravan, and with a comparison with other works.

4.1 Program Model

We will focus on programs respecting the following constraints:

1. The only data structures are of base types (integers, reals, etc.) and arrays thereof.
2. The only control structures are the sequence, the `do` loop, the `while` loop[1], and the `if..then..else` construct. `goto`s and procedure calls are forbidden.
3. Basic statements are assignments to scalars or array elements.
4. No pointer, `EQUIVALENCE` or aliasing is allowed.

[1] Similarly to `do` loops, an iteration of a `while` loop is denoted by giving its ordinal number w in the iteration sequence.

4.2 Which Approximation?

We describe in this section the kind of non-affine constraints that arise in real programs. Four examples, each of them capturing a particular feature of non-affine constraints are presented. They will illustrate the techniques proposed in this chapter and are used at the end of this section to give the intuition of our approximate analysis.

4.2.1 The need for an approximation

We have seen in Section 3.4.5, page 94, how to extend the domain of exact array dataflow dependence analysis to some particular non-static programs. However, as soon as we extend our program model to include general conditionals, non-linear bounds, `do` loops with non-linear bounds and non-linear array subscripts, even dependence testing becomes undecidable. The difficulty of the problem can easily be illustrated by the following example:

```
if n>2 then
 if x>0 then
  if y>0 then
   if z>0 then
    a(x**n) = ..
    .. = a(y**n + z**n)
   endif
  endif
 endif
endif
```

Finding the dependences in this program is as difficult as proving Fermat's last theorem.

Two causes explain the failure of intraprocedural dataflow analyses based on an affine framework.

Non-affine References Shen et al. [117] have shown that on a set of benchmarks and real programs, around 34% of array subscripts were non-linear, and a majority, 80%, were in fact affine expressions with respect to non loop index variables. These variables are either scalars, or arrays such as in `a(b(x))`. Polynomial subscripts come from linearized subscripts, induction variable substitution but in some cases are more complex; In the program `OCEAN` of the Perfect Club Benchmarks, subscripts of the form $ai+bj+c$ where i and j are loop indices, a and b are subroutine parameters and c a parameter, appear in many important loop nests.

Dynamic Control `while` loops and conditionals may use predicates whose value can only be evaluated at execution time. `while` loops and conditionals are very common in real programs and it is important to be

able to analyze in detail such control structures. Blume [16] gives some real examples such as `SPICE` where `do` loops with abnormal exits or `if` statements followed by `goto`s are in large proportions and hinder usual parallelism detection. `do` loops with non-affine bounds can considered as a particular case of `while` loops.

One way to deal with intractable terms is to instrument the program so that, when run, it produces the exact dependence flow graph [79, 94, 98]. This approach has one main drawback: The dependence graph depends on the input values of the variables, therefore several executions may produce different graphs. Exposing parallelism in these conditions is more difficult, since a code transformation that is legal for one execution may appear illegal for another one. Several versions of the parallel code can be produced, introducing run-time tests so as to determine whether some parts of the code can be run in parallel. In that case, the overhead due to the tests may be significant [98]. Another approach is to speculate on the dataflow dependences and suppose that some loops are parallel [108, 109]. During the parallel execution, some conditions are checked so as to decide whether it is necessary to revert to a sequential execution or not. We will adopt *a priori* another approach, which consists in computing at compile-time an approximated DFG. We will show in Section 4.3.3 that this is not in contradiction with a run-time method determining the exact DFG.

Traditional alias analyses resort to complementary techniques so as to improve their output when dealing with non-affine terms. For instance, several studies [48, 18] on the Perfect Club Benchmarks have shown the importance of symbolic analysis [67] in the detection of parallelism.

Hence, it appears that an extension of exact value-based dependence analysis should be able to take into account non-linear references and dynamic control, with the possibility to improve the accuracy of the analysis with the results of other analyses, such as symbolic analysis or abstract interpretation. The analysis we are looking for should fulfill the following requirements:

- Conservative: An approximated value-based dependence is an over-approximation of the exact dependence. No approximated dependence implies no exact dependence.

- Exact on static fragments: Exact array dataflow dependence analysis is a particular case of the approximate analysis.

- Information on non-affine constraints provided by complementary analyses may be taken into account. The nature of such information will be discussed in Section 4.4. Intuitively, the amount of information on non-affine terms provides an estimation of the accuracy of the output of the analysis. However, this may not be always true for two reasons: Either because the properties collected about non-affine constraints are not relevant for the computation of the dataflow dependence, or because the

analysis is not able to handle all relevant information implied by these properties. This problem of accuracy will be tackled in Section 4.5.

Collard [29, 30] extended usual exact analysis to `while` loops, at the cost of accuracy, and we propose a generalization of his approach.

4.2.2 Motivating examples

The following four examples are used throughout this chapter to illustrate the working principles and applicability of our approximate dataflow analysis. Each of them capture one of the non-affine features described above.

```
    do x1 = 1 to n
S1   s = ..
     do x2 = 1 while (P(x1,x2))
S2    s = .. s ..
     enddo
R    .. = .. s ..
    enddo
```

Example E1

```
   do x = 1, n
    if (P(x)) then
S1   s = ..
    else
S2   s = ..
    endif
   enddo
R  if (n>0) then .. = .. s ..
```

Example E2

`while` loop Example E1 is a kernel that appears in several convolution codes[2]. Each operation $\langle \mathtt{S}_1, x1\rangle$ assigns a new value to variable `s`. In turn, statement $\mathtt{S}_2$ assigns `s` an undefined number of times (possibly zero). The value read in `s` by statement `R` is thus defined either by $\mathtt{S}_1$, or by some instance of $\mathtt{S}_2$, in the same iteration of the `do` loop (the same x_1). Moreover, when a value of `s` is read in an instance of $\mathtt{S}_2$, it comes either from the previous iteration of the `while` loop or from the previous assignment of statement $\mathtt{S}_1$ if the read occurs in the first iteration of the `while` loop.

`if..then..else` construct The very simple piece of code E2 illustrates a typical case of `if..then..else` construct: The value of `s` read by statement `R` comes either from an instance of $\mathtt{S}_1$ or from an instance of $\mathtt{S}_2$ and the iteration for which the value of `s` read by `R` is written is n.

Non-affine array subscript In Example E3, the difficulty consists in finding that the value of ii used in an instance of statement $\mathtt{S}_4$ is the same as in the instance of statement $\mathtt{S}_2$ in the following iteration. It is then possible to conclude that any definition of `A` performed by an instance of $\mathtt{S}_4$ is killed by the assignment done during the following iteration by statement $\mathtt{S}_2$, excepted for

[2]Such codes include `horn.c` by T. Burkit, implementing Horn and Schunck's algorithm to perform 3D Gaussian smoothing by separable convolution, and `singh.c`, written by J. Barron, implementation of Ajit Singh, ICCV, 1990, pages 168–177.

```
S1  ii = ..
    do x = 1, n
S2   A(ii) = ..
S3   ii = ..
S4   A(ii) = ..
    end do
    do x = 1, n
R    .. = .. A(x) ..
    enddo
```

Example E3

```
    if (P) then
S1    jlow = 2, jup = jmax -1
    else
S2    jlow = 1, jup = jmax
    endif
    do x = 1, n
S3   A(jlow : jup) = ..
     if (P) then
R     .. = A(2 : jmax-1)
     endif
    enddo
```

Example E4

the last iteration of the loop. Hence, the source of $\mathtt{A}(x)$ for an operation $\langle \mathtt{R}, x\rangle$ comes either from an instance of $\mathtt{S}_2$, from operation $\langle \mathtt{S}_4, n\rangle$ or does not exist.

Non-affine loop bounds Example E4, taken from [124], requires a deeper understanding of the values stored in the variables. The predicate P has the same value in both conditionals, so for any instance of R, $\mathtt{jlow} = 2$ and $\mathtt{jup} = \mathtt{jmax} - 1$ since the values of $\mathtt{jlow}, \mathtt{jup}$ and jmax are not modified inside the loop. This entails that any value of A read in statement R is defined by the instance of statement $\mathtt{S}_3$ of the same iteration of the do loop. In order to find the relations between the variables of the program, we will resort in this case to the analysis described by Tu and Padua [124]. Note that we only need to know in that example that P keeps the same value in the two predicates and that $1 \leq \mathtt{jlow} \leq 2$, $\mathtt{jmax} - 1 \leq \mathtt{jup} \leq \mathtt{jmax}$ to conclude. These inequalities can be found by other analyses, such as the one performed by PIPS.

4.2.3 From exact to approximate analysis

As soon as we extend our program model to include conditionals, while loops, do loops with non-linear bounds or subscripts, the algorithm used in exact array dataflow analysis breaks down. The reason is that the existence predicate and subscript equation may contain intractable terms. One possibility is to ignore them. In this way, the existence predicate $x \in \mathsf{I}(\mathtt{S})$ for statement S is replaced by $x \in \hat{\mathsf{I}}(\mathtt{S})$, where $\hat{\mathsf{I}}(\mathtt{S})$ is a superset of $\mathsf{I}(\mathtt{S})$ which is obtained by ignoring non-linear constraints. Supposing for the moment that the subscript condition is still linear, we may obtain an approximate set of candidate sources:

$$\hat{\mathsf{Q}}^p(y) = \left\{ x \,\middle|\, x \in \hat{\mathsf{I}}(\mathtt{S}), f(x) = g(y), \langle \mathtt{S}, x\rangle \preceq_p \langle \mathtt{R}, y\rangle \right\}. \tag{4.1}$$

However, we can no longer say that an iteration vector of a possible source is given by the lexicographic maximum of this set, since the result may precisely be one of the candidates which is excluded by the non-linear part of $\hat{\mathsf{I}}(\mathtt{S})$. One

solution is to take all of $\hat{\mathsf{Q}}^p(y)$ as an approximation to the direct dependence. If we do that, and with the exception of very special cases, computing the maximum of approximate iteration vectors has no meaning, and the best we can do is to use their union as an approximation. Can we do better than that? Let us consider some examples.

`while` **loop** What is the source of `s` in statement `R` in E1? There are two possibilities, statements $\mathtt{S}_1$ and $\mathtt{S}_2$. Intuitively, `R` always reads a value of `s` that has been defined in the same iteration of the outer loop. In the case of $\mathtt{S}_1$, the source candidate of $\langle \mathtt{R}, x\rangle$, for any value x of `x1`, is exactly $\langle \mathtt{S}_1, x\rangle$. Things are more complicated for $\mathtt{S}_2$, since we have no idea of the iteration count of the `while` loop. We may, however, give a name to this count, say α, and write the set of iteration vectors of the candidate sources as:

$$\mathsf{Q}_2^1(x) = \{z\,|z[1] = x, 1 \leq z[2], z[2] = \alpha\}\,.$$

We may then compute the maximum of this set, which is simply

$$\varsigma_2^1(x) = \textbf{ if } \alpha > 0 \textbf{ then } \langle \mathtt{S}_2, x, \alpha\rangle \textbf{ else } -.$$

The dependence between $\mathtt{S}_2$ and `R` at depth 0 is killed by the dependence between $\mathtt{S}_1$ and `R` at depth 1, therefore we do not compute it. The last step is to take the maximum of the two sources, which is

$$\textbf{if } \alpha > 0 \textbf{ then } \langle \mathtt{S}_2, x, \alpha\rangle \textbf{ else } \langle \mathtt{S}_1, x\rangle\,.$$

We have thus formally derived the expected precise result. The trick here has been to give a name to an unknown quantity, α, and to solve the problem with α as a parameter.

`if..then..else` **construct** Example E2 is slightly more complicated: We assume, from the environment that $n \geq 1$. What is the source of `s` in statement `R`? We may build an approximate candidate set from $\mathtt{S}_1$ and another one from $\mathtt{S}_2$. Since both are approximate, we cannot do anything beside taking their union, and the result is highly inaccurate.

Another possibility is to partition the set of candidates according to the value x of the loop counter. Let us introduce a new boolean function $b(x)$ which represents the outcome of the test at iteration x. The xth candidate may be written

$$\tau(x) = \textbf{ if } b(x) \textbf{ then } \langle \mathtt{S}_1, x\rangle \textbf{ else } \langle \mathtt{S}_2, x\rangle\,.$$

We then have to compute the maximum of all these candidates (this is an application of Property 3.1, page 73). It is an easy matter to prove that $x < x' \Longrightarrow \tau(x) \prec \tau(x')$, so the source is $\tau(n)$. Since we have no idea of the value of

$b(n)$, the best we can do is to say that we have a source *set*, or a *fuzzy* source, which is obtained by taking the union of the two arms of the conditional:

$$\sigma([]) = \{\langle \mathsf{S}_1, n\rangle, \langle \mathsf{S}_2, n\rangle\}. \tag{4.2}$$

Notice here the precision we have been able to achieve. However, the technique we have used here is not easily generalized. Another way of obtaining the same result is the following. Let $\mathsf{L} = \{x \,|1 \leq x \leq n\}$. Observe that the candidate set from S_1 (resp. S_2) can be written $\{x \,|x \in \mathsf{D}_1 \cap \mathsf{L}\}$ (resp. $\{x \,|x \in \mathsf{D}_2 \cap \mathsf{L}\}$) where $\mathsf{D}_1 = \{x \,|b(x) = \text{true}\}$ and $\mathsf{D}_2 = \{x \,|b(x) = \text{false}\}$. Obviously,

$$\mathsf{D}_1 \cap \mathsf{D}_2 = \emptyset, \tag{4.3}$$

and

$$\mathsf{D}_1 \cup \mathsf{D}_2 = \mathbb{Z}. \tag{4.4}$$

We have to compute $\alpha = \max(\max \mathsf{D}_1 \cap \mathsf{L}, \max \mathsf{D}_2 \cap \mathsf{L})$. It is a general property that (4.4) implies that:

$$\alpha = \max \mathsf{L} = n. \tag{4.5}$$

By (4.3) we know that α belongs either to D_1 or D_2 which gives again the result (4.2).

To summarize these observations, our method will be to give new names (or *parameters*) to the result of maxima calculations in the presence of non-linear terms. These parameters are not arbitrary. The sets they belong to – the parameter domains –, or the non-linear constraints involved, are in relations to each others, as for instance equations (4.3) and (4.4). These relations can be found simply by examination of the syntactic structure of the program, or by more sophisticated techniques. From these relations between the non-linear constraints follow relations on the parameters, like equation (4.5), which can then be used to simplify the resulting fuzzy sources. In some cases, these relations may be so precise as to reduce the source set to a singleton, thus giving an exact result. Examples E3 and E4 can be handled too by applying this method, but they require a more detailed formalism, presented in the following section.

4.3 Description of the Formalism

We present in this section a formal definition of *Fuzzy Array Dataflow Analysis* (FADA). Note that this formalism differs from the formalism proposed by Collard for the first version of FADA [29] (The differences are described on page 4.8.1.) First of all, we define a representation for non-affine constraints. Thanks to this representation, the expression of the source boils down to a

computable expression with linear constraints and unknown parameters. When these parameters take all the values of a set defined by linear constraints, we get a set of possible sources, called the fuzzy source. How this set of values is built will be the subject of the next sections.

As in the previous chapter, we examine the dependences between a statement R reading array A with subscript $g(y)$ and the statements $\mathtt{S}_i, 1 \leq i \leq n$ assigning a value to A with the respective subscripts $f_i(x_i)$ where x_i is the iteration vector of $\mathtt{S}_i$. We also use the same notations as in the precedent chapter.

4.3.1 Non-affine constraints

Let us first have a close look at the non-affine constraints. Notice that they come either from the predicate of a `while` or `if`, from a non-affine loop bound appearing in the existence predicate, or from a non-affine array subscript appearing in the conflicting access predicate. Each constraint can be numbered according to its apparition order in the text of the program. Let C denote the set of integers indexing non-affine constraints. Given a constraint c_h, $h \in \mathsf{C}$, we note $\mathtt{T}_h$ the statement in which it appears. This statement is either the `then` or `else` branch of a conditional, or a loop with non-affine bounds, or an assignment statement in which a non-affine subscript is used in an array access.

If c_h appears in the set of candidate sources $\mathsf{Q}_k^p(y)$, a write operation $\langle \mathtt{S}_k, x\rangle$ with $x \in \mathsf{Q}_k^p(y)$ depends on the value of c_h for an instance of $\mathtt{T}_h$. More precisely, the iteration vector of this instance of $\mathtt{T}_h$ is a prefix of the iteration vector x of the write. Let us denote this prefix $x[1..e_h]$. If $\mathtt{T}_h$ is a conditional or an assignment then $e_h = d_{\mathtt{T}_h}$ since all the surrounding loop counters of $\mathtt{T}_h$ may be used by c_h. When $\mathtt{T}_h$ is a `do` or a `while`, then $e_h = d_{\mathtt{T}_h} + 1$, since the loop index of the loop defined by $\mathtt{T}_h$ does not belong to the surrounding loop indices of $\mathtt{T}_h$ but are involved in the definition of c_h. In the definition of $\mathsf{Q}_k^p(y)$ as given by (3.3), the expression of the non-affine constraint c_h is $c_h(x[1..e_h], y)$, with $x[1..e_h] \in \mathsf{I}(\mathtt{T}_h)$. c_h depends on y in the case it comes from the subscript predicate. However, since the only term depending on p is the linear sequencing predicate, non-affine constraints cannot depend on the dependence depth p.

Moreover, any non-affine constraint is a boolean function which can be written as an (in)equality with non-affine terms. Formally, if c_h is defined by an inequality, we write it:

$$c_h(x, y) = \left(a_h(x, y) + \sum_k \beta_{hk} n_k(x, y) \geq 0 \right), \tag{4.6}$$

where a_h is an affine form, β_{hk} integer coefficients and the n_k's non-affine terms. c_h is the composition of an affine predicate with the non-affine terms n_k. Note that the functions n_k can represent variables not among the loop counters. In that case, if `s` is the name of the variable, we will denote $s(\langle \mathtt{T}, x\rangle)$ the value of this variable at operation $\langle \mathtt{T}, x\rangle$.

Definition 4.1 (parameter domain) *Let* $\mathsf{C}_k \subseteq \mathsf{C}$ *denote the set of the indices of the constraints involved in the computation of* $\mathsf{Q}_k^p(y)$ *and* m_k *denote* $\max_{h\in\mathsf{C}_k} e_h$*. The set:*

$$\mathsf{D}_k(y) = \left\{ x \;\middle|\; x \in \mathbb{Z}^{m_k},\ \bigwedge_{h\in\mathsf{C}_k} c_h(x[1..e_h], y) \right\},$$

is the set of iteration vectors of dimension m_k *for which all of the constraints indexed by* C_k *are true. This set is called the* parameter domain *associated to the dependence between an instance of* $\mathtt{S}_k$ *and* $\langle \mathtt{R}, y\rangle$*.*

Note that m_k does not depend on y and that $m_k \le d_{\mathtt{S}_k}$. By convention, when all constraints in $\mathsf{Q}_k^p(y)$ are linear, $\mathsf{D}_k(y) = \mathbb{Z}^{m_k}$.

We illustrate these definitions on the examples of Section 4.2.2, page 100.

`while` loop For Example E1, there is only one non-affine constraint c_1 in the program and $c_1(x, y) = (P(x[1], x[2]) = \text{true})$. The set of iteration vectors of the candidate sources for the dependence at depth 1 between an instance of statement $\mathtt{S}_2$ and $\langle \mathtt{R}, y\rangle$ is:

$$\mathsf{Q}_2^1(y) = \{x\,|1 \le x[1] \le n, 1 \le x[2], c_1(x), x[1] = y\}.$$

The parameter set $\mathsf{D}_2(y)$ is the set defined by all non-affine constraints in $\mathsf{Q}_2^1(y)$: $\mathsf{D}_2(y) = \{x\,|c_1(x)\}$. The domain is the same for the dependence between two instances of $\mathtt{S}_2$ and there is no non-affine term in the dependence between $\mathtt{S}_1$ and $\mathtt{R}$, thus the parameter domain is $\mathsf{D}_1(y) = \mathbb{Z}$.

`if..then..else` construct For Example E2, $c_1(x) = (P(x) = \text{true})$ is the constraint associated to the true branch of the conditional, $c_2(x) = (P(x) = \text{false})$ is the constraint associated to the false branch. The parameter domains are: $\mathsf{D}_1([]) = \{x\,|c_1(x)\}$ and $\mathsf{D}_2([]) = \{x\,|c_2(x)\}$.

Non-affine array subscript The non-affine terms appearing in Example E3 for the dependences between $\mathtt{S}_2$, $\mathtt{S}_4$ and $\mathtt{R}$ come from the subscript equation: $c_1(x, y) = (ii(\langle \mathtt{S}_2, x\rangle) = y)$ and $c_2(x, y) = (ii(\langle \mathtt{S}_4, x\rangle) = y)$. The parameter domains associated to the dependences between an instance of $\mathtt{S}_2$ and $\langle \mathtt{R}, y\rangle$ and between an instance of $\mathtt{S}_4$ and the same operation are respectively: $\mathsf{D}_2(y) = \{x\,|c_1(x, y)\}$, $\mathsf{D}_4(y) = \{x\,|c_2(x, y)\}$.

Non-affine loop bounds Finally, for Example E4, the two non-affine constraints associated to the `if..then..else` construct are: $c_1([]) = (P = \text{true})$, $c_2([]) = (P = \text{false})$. The vector assignment in statement $\mathtt{S}_3$ contains an implied loop and the statement can be rewritten as:

```
do x' = jlow, jup
 A(x') = ..
enddo
```

The non-affine terms are then:

$$c_3(x) = (x[2] \geq jlow(\langle \mathsf{S}_3, x[1] \rangle)),$$

and

$$c_4(x) = (x[2] \leq jup(\langle \mathsf{S}_3, x[1] \rangle)).$$

The parameter domain associated to the dependence between an instance of S_3 and $\langle \mathtt{R}, y \rangle$ is: $\mathsf{D}_3(y) = \{ x \,|c_3(x) \wedge c_4(x) \}$.

4.3.2 Parameterization

Let us recall the definition of the source:

$$\sigma(y) = \max_{1 \leq k \leq n} \max_{\leq p \leq d_{\mathsf{S}_k\mathtt{R}}} \varsigma_k^p(y),$$

and the definition of the maximum of a candidate source set:

$$\varsigma_k^p(y) \quad = \quad \left\langle \mathsf{S}_k, \max \mathsf{Q}_k^p(y) \right\rangle . \tag{4.7}$$

$\varsigma_k^p(y)$ cannot be calculated exactly when non-affine constraints appear in the definition of $\mathsf{Q}_k^p(y)$. We could approximate $\varsigma_k^p(y)$ and then take the maximum of the approximated $\varsigma_k^p(y)$'s. However, making the approximation in the last step of the computation of the source would provide more accurate results. This is the purpose of the parameterization: to encode non-affine constraints as parameters so as to achieve the computation of the source with respect to these parameters, and then, eventually, make an approximation.

We first partition each set $\mathsf{Q}_k^p(y)$ into subsets defined by parametric linear constraints. Let $\mathsf{L}_k^p(y)$ denote the set of vectors of dimension d_k defined by the linear constraints appearing in $\mathsf{Q}_k^p(y)$. The set of iteration vectors of the candidate sources is:

$$\mathsf{Q}_k^p(y) \quad = \quad \mathsf{L}_k^p(y) \bigcap \{ x \,|x[1..m_k] \in \mathsf{D}_k(y) \} .$$

Partitioning $\mathsf{Q}_k^p(y)$ is obtained by partitioning $\mathsf{D}_k(y)$ as the union of its elements:

$$\mathsf{D}_k(y) = \bigcup_{x \in \mathsf{D}_k(y)} \{x\}.$$

Let $\hat{\mathsf{Q}}_k^p(x, y) = \mathsf{L}_k^p(y) \cap \{ z \,|z[1..m_k] = x \}$ denote a subset of the partition of $\mathsf{Q}_k^p(y)$. Note that x is the vector of the first m_k coordinates of any vector in $\hat{\mathsf{Q}}_k^p(x, y)$. Then:

$$\mathsf{Q}_k^p(y) \quad = \quad \bigcup_{x \in \mathsf{D}_k(y)} \hat{\mathsf{Q}}_k^p(x, y). \tag{4.8}$$

From (4.7), (4.8) and Property 3.1, we obtain:

$$\varsigma_k^p(y) \quad = \quad \left\langle \mathsf{S}_k, \max_{x \in \mathsf{D}_k(y)} \left(\max \hat{\mathsf{Q}}_k^p(x, y) \right) \right\rangle . \tag{4.9}$$

An elementary source operation $\hat{\varsigma}_k^p(x,y)$ can then be evaluated for each subset $\hat{\mathsf{Q}}_k^p(x,y)$:

$$\hat{\varsigma}_k^p(x,y) \quad = \quad \left\langle \mathsf{S}_k, \max \hat{\mathsf{Q}}_k^p(x,y) \right\rangle, \tag{4.10}$$

which is computable by parametric integer programming. From (4.9) and (4.10), we have:

$$\varsigma_k^p(y) \quad = \quad \max_{x \in \mathsf{D}_k(y)} \hat{\varsigma}_k^p(x,y). \tag{4.11}$$

If the maximum defined by (4.11) exists, then it is reached for one vector of $\mathsf{D}_k(y)$. Such a vector is called a *parameter of the maximum*:

Definition 4.2 (parameter of the maximum) *The integer vector of* $\mathsf{D}_k(y)$ *for which the maximum* (4.11) *is reached is called the* parameter of the maximum of D_k for statement S_k at depth p *and denoted* $\alpha_k^p(y)$. *(If the maximum does not exist, we set* $\alpha_k^p(y)$ *to an undefined value.) According to the definition of* $\hat{\mathsf{Q}}_p^k(y)$, $\alpha_k^p(y)$ *is defined by:*

$$\alpha_k^p(y) = \max \mathsf{L}_k^p(y)_{|m_k} \cap \mathsf{D}_k(y). \tag{4.12}$$

The set $\mathsf{L}_k^p(y)_{|m_k}$ is the projection of $\mathsf{L}_k^p(y)$ on $\mathbb{Z}_k^m$:

$$\mathsf{L}_k^p(y)_{|m_k} = \left\{ x \left| \exists z, z[1..m_k] = x, z \in \mathsf{L}_k^p(y) \right. \right\}.$$

The following equality always holds:

$$\varsigma_k^p(y) = \hat{\varsigma}_k^p(\alpha_k^p(y), y). \tag{4.13}$$

Thus it implies that the source can be written as:

$$\sigma(y) \quad = \quad \max_{1 \le k \le m} \max_{0 \le p \le d_{\mathsf{S}_k \mathtt{R}}} \hat{\varsigma}_k^p(\alpha_k^p(y), y), \tag{4.14}$$

which is still computable by parametric integer programming. Note that when there is no non-affine constraints involved in the computation of $\sigma(y)$ then $\alpha_k^p(y) = \max \mathsf{L}_k^p(y)_{|m_k}$ and the source computed in (4.11) is the exact source. Hence, the exact array dataflow analysis presented in the preceding chapter is a particular case of FADA. Moreover, the techniques presented in the previous chapter to compute the expression of the source (3.5) can also be used to compute the expression of the source (4.14) w.r.t. the parameters $\alpha_k^p(y)$. In particular, the optimizations described in Section 3.2.3, page 76, and our simplified algorithm can be used.

4.3.3 Fuzziness

To sum things up, we enumerated each set $\mathsf{D}_k(y)$ of non-affine constraints by a parameter. Among these parameters, we distinguished one element for each p, the parameter of the maximum $\alpha_k^p(y)$ of $\mathsf{D}_k(y)$. The benefit is that the expression (4.14) can be computed with the same methods as in the case of exact array dataflow analysis, using parametric integer programming tools. The source is then a function of the parameters of the maximum and is equivalent to the definition of the source with respect to non-affine constraints. No approximation has been done on the source, yet.

However, parameters of the maximum cannot themselves be computed, because the sets $\mathsf{D}_k(y)$ of non-affine constraints cannot be handled. The expression of the source given by (4.14) could be used at run-time to determine dynamically the exact source of a variable, provided that the parameters of the maximum are computed and updated dynamically too. We do not consider this possible use of (4.14) in the sequel.

A very simple method is to compute a *set of possible sources* – or a fuzzy source – by giving all possible values to the parameters. This would mean that we would not even try to take non-affine constraints into account. Obviously, this is a safety net for a fuzzy array dataflow analysis and is similar to the "panic mode" in Wonnacott's work [107].

A better approach is to reduce the number of possible sources. To do that, we will try to find properties (call them P) on the non-affine constraints. From these properties we will deduce linear properties (call them $\hat{\mathsf{P}}$) on the parameters $\alpha_k^p(y)$. The benefit of this approach in two steps is that we can then prove, for some P, that the set of sources we obtain is the most precise that can be derived. That is, there is no loss of information when deriving $\hat{\mathsf{P}}$ from P.

Therefore, the method to be presented in the next sections will proceed in three steps:

1. Properties P will be obtained by an analysis of the non-affine constraints of the program (Section 4.4). The more accurate the description of the non-affine constraints, the smaller the set of possible constraints described by P. In Figure 4.1, we represented the case where only one non-affine constraint, c, is involved in the computation of the source. c belongs to the set of constraints verifying the conditions of P, ensuring that the analysis is conservative.

2. Properties $\hat{\mathsf{P}}$ are then derived from P. The properties $\hat{\mathsf{P}}$ on the parameters are consequences of the properties P, so as to perform a conservative analysis. The objective is then to show that, as in Figure 4.2, each vector verifying the constraints of $\hat{\mathsf{P}}$ corresponds to at least one constraint verifying the conditions of P. This would prove that there is no loss of information during this step, as far as the computation of the source is concerned.

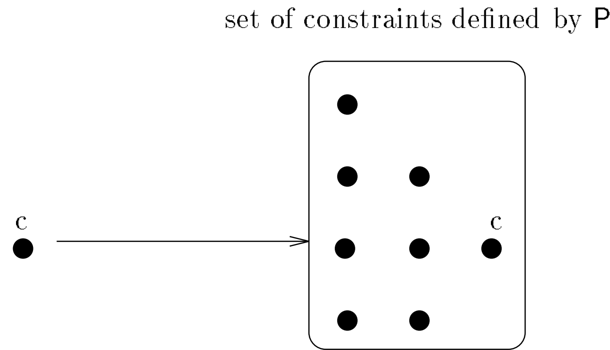


Figure 4.1: Approximating a non-affine constraint.

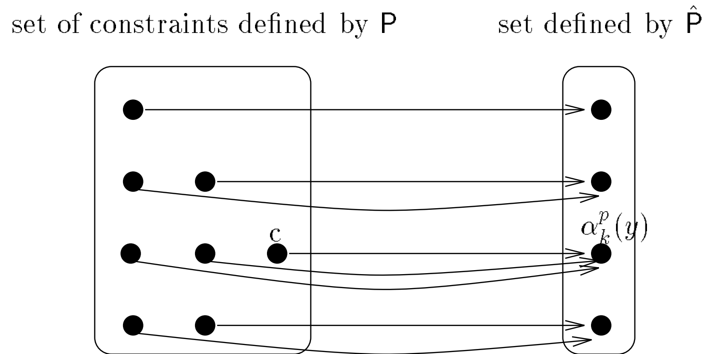


Figure 4.2: Translating P into $\hat{\mathsf{P}}$.

3. We then build a parametric source (4.14), taking into account Properties $\hat{\mathsf{P}}$. The set of all possible sources is obtained by giving to the parameters all the values verifying $\hat{\mathsf{P}}$:

$$\mathsf{S}(y) = \left\{ \max_{1 \le k \le m} \max_{0 \le p \le d_{\mathsf{s}_k \mathtt{R}}} \hat{\varsigma}_k^p(x_k^p, y) \,\middle|\, x_k^p \in \mathbb{Z}^{m_k}, \hat{\mathsf{P}}(\ldots, x_k^p, \ldots) \right\}, \tag{4.15}$$

which can be computed exactly if $\hat{\mathsf{P}}$ is a conjunction or disjunction of quasi-affine constraints. The expression $\max_{1 \le k \le m} \max_{0 \le p \le d_{\mathsf{s}_k \mathtt{R}}} \hat{\varsigma}_k^p(x_k^p, y)$ is computed w.r.t. x_k^p with the techniques presented in the previous chapter. As illustrated in Figure 4.3, to each parameter corresponds a different source.

The fuzziness of the source depends on the precision with which $\hat{\mathsf{P}}$ abstracts the relations existing among the parameters of the maximum $\alpha_k^p(y)$, $k = 1..m$. Let us come back to our four examples presented in Section 4.2.2, page 100.

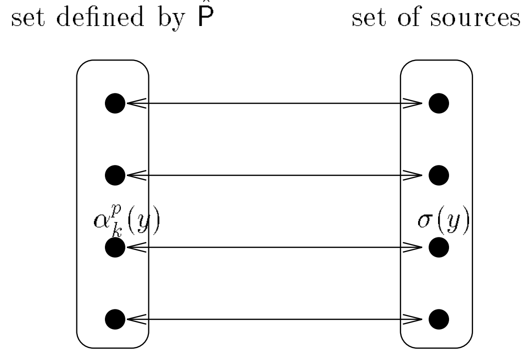


Figure 4.3: Computing set of sources.

4.3.4 Example E1: `while` loop

In Example E1, consider the dependence between an instance of $\mathtt{S}_2$ and $\langle \mathtt{R}, y\rangle$ at depth 1. The set $\mathsf{L}_2^1(y)$ defined by the linear constraints of $\mathsf{Q}_2^1(y)$ is equal to $\{x \,|1 \le x[1] \le n, 1 \le x[2], x[1] = y\}$ and it is partitioned into the subsets:

$$\hat{\mathsf{Q}}_2^1(x, y) = \{z \,|1 \le z[1] \le n, 1 \le z[2], z[1] = x[1] = y, z[2] = x[2]\},$$

for all $x \in \mathsf{D}_2(y) = \{x \,|c_1(x[1], x[2])\}$. The environment is $1 \le y \le n$ and in this context, an elementary source $\hat{\varsigma}_2^1(x, y)$ can then be computed by any of the method of Chapter 3:

$$\hat{\varsigma}_2^1(x, y) = \left| \begin{array}{ll} \textbf{if } x[1] = y & \\ \textbf{then} & \left| \begin{array}{l} \textbf{if } 1 \le x[2] \\ \textbf{then } \langle \mathtt{S}_2, x\rangle \\ \textbf{else } - \end{array} \right. \\ \textbf{else } - & \end{array} \right.$$

The dependence between an instance of $\mathtt{S}_1$ and $\langle \mathtt{R}, y\rangle$ at depth 1 is $\varsigma_1^1(y) = \langle \mathtt{S}_1, y\rangle$, thus, as the lazy method applies, we can conclude that the exact source of `s` at operation $\langle \mathtt{R}, y\rangle$ is:

$$\sigma(y) = \left| \begin{array}{ll} \textbf{if } \alpha_2^1(y)[1] = y & \\ \textbf{then} & \left| \begin{array}{l} \textbf{if } 1 \le \alpha_2^1(y)[2] \\ \textbf{then } \left\langle \mathtt{S}_2, \alpha_2^1(y)\right\rangle \\ \textbf{else } \langle \mathtt{S}_1, y\rangle \end{array} \right. \\ \textbf{else } \langle \mathtt{S}_1, y\rangle & \end{array} \right.$$

with the parameter of the maximum defined as:

$$\alpha_2^1(y) = \max \{x \,|1 \le x[1] \le n, 1 \le x[2], x[1] = y\} \cap \{x \,|c_1(x)\}.$$

The first leaf of $\hat{\varsigma}_2^1(x,y)$ cannot be killed but the bottoms of the second and third leaves can. The exactness of this source means that no approximation has been done yet and is given in a parametric form with respect to the non-affine term P. If nothing is known about P, by giving to $\alpha_2^1(y)$ all possible values, we obtain the fuzzy source:

$$\mathsf{S}(y) = \left\{ \textbf{ if } x[1] = y \wedge 1 \le x[2] \textbf{ then } \langle \mathtt{S}_2, x\rangle \textbf{ else } \langle \mathtt{S}_1, y\rangle \,\middle|\, x \in \mathbb{Z}^2 \right\}.$$

Now let us consider the source of $\langle \mathtt{S}_2, y\rangle$. As previously, we consider first dependences at depth 1. In this case, the set $\mathsf{L}_2^1(y)$ is equal to

$$\{x \,|1 \le x[1] \le n, 1 \le x[2], x[1] = y[1], x[2] < y[2]\}.$$

The subset $\hat{\mathsf{Q}}_2^1(x,y)$ defining the partition of $\mathsf{L}_2^1(y)$ is:

$$\hat{\mathsf{Q}}_2^1(x,y) = \{z \,|1 \le z[1] \le n, 1 \le z[2], z[1] = x[1] = y[1], z[2] = x[2] < y[2]\}.$$

The environment is $1 \le y[1] \le n, 1 \le y[2], c_1(y,y)$ and an elementary source coming from a write of $\mathtt{S}_2$ is:

$$\hat{\varsigma}_2^1(x,y) = \left| \begin{array}{l} \textbf{if } x[1] = y[1] \\ \textbf{then} \left| \begin{array}{l} \textbf{if } 1 \le x[2] < y[2] \\ \textbf{then } \langle \mathtt{S}_2, x\rangle \\ \textbf{else } - \end{array} \right. \\ \textbf{else } - \end{array} \right.$$

Finally, the source of $\langle \mathtt{S}_2, y\rangle$ is:

$$\sigma(y) = \left| \begin{array}{l} \textbf{if } \alpha_2^1(y)[1] = y[1] \\ \textbf{then} \left| \begin{array}{l} \textbf{if } 1 \le \alpha_2^1(y)[2] < y[2] \\ \textbf{then } \langle \mathtt{S}_2, \alpha_2^1(y)\rangle \\ \textbf{else } \langle \mathtt{S}_1, y\rangle \end{array} \right. \\ \textbf{else } \langle \mathtt{S}_1, y\rangle \end{array} \right.$$

with:

$$\alpha_2^1(y) = \max\{x \,|1 \le x[1] \le, 1 \le x[2], x[1] = y[1], x[2] < y[2]\} \cap \{x \,|c_1(x)\}.$$

The source of `s` for $\langle \mathtt{S}_2, y\rangle$ is one of the previous executions of $\mathtt{S}_2$, for the same iteration of the `do` loop, or, if this instance does not exists, the previous instance of $\mathtt{S}_1$.

Note that we cannot conclude yet that the source comes from the previous iteration of the `while` if $y[2] > 1$. What is not taken into account in this computation is the fact that if an iteration of a `while` loop is executed, then all previous iterations, for the same value of surrounding loop indices, are executed too. A structural analysis will find such information (see Section 4.4.3).

4.3.5 Example E2: `if..then..else` construct

Let us compute a parametric source of $\mathtt{s}$ from $\langle \mathtt{R}, [] \rangle$ of Example E2. Consider first the dependence between an instance of $\mathtt{S}_1$ and $\langle \mathtt{R}, y \rangle$. The set of linear constraints of $\mathsf{Q}_1^0([])$ is:

$$\mathsf{L}_1^0([]) = \{x \,|1 \leq x \leq n\},$$

and it is partitioned into subsets:

$$\hat{\mathsf{Q}}_1^0(x, []) = \{z \,|1 \leq z \leq n, z = x\},$$

for all $x \in \mathsf{D}_1(y)$. The environment is $n \geq 1$ and an elementary source is:

$$\hat{\varsigma}_1^0(x, []) = \textbf{ if } 1 \leq x \leq n \textbf{ then } \langle \mathtt{S}_1, x \rangle \textbf{ else } -.$$

By symmetry,

$$\hat{\varsigma}_2^0(x, []) = \textbf{ if } 1 \leq x \leq n \textbf{ then } \langle \mathtt{S}_2, x \rangle \textbf{ else } -.$$

Finally, the parametric source is (we drop the notation of the zero-dimension vector):

$$\sigma = \left| \begin{array}{ll} \textbf{if} & 1 \leq \alpha_1^0 \leq n \\ \textbf{then} & \left| \begin{array}{ll} \textbf{if} & 1 \leq \alpha_2^0 \leq n \\ \textbf{then} & \left| \begin{array}{ll} \textbf{if} & \alpha_1^0 \geq \alpha_2^0 \\ \textbf{then} & \left| \begin{array}{l} \textbf{if } \alpha_1^0 \leq \alpha_2^0 \\ \textbf{then } \left\langle \mathtt{S}_2, \alpha_2^0 \right\rangle \\ \textbf{else } \left\langle \mathtt{S}_1, \alpha_1^0 \right\rangle \end{array} \right. \\ \textbf{else} & \left\langle \mathtt{S}_2, \alpha_2^0 \right\rangle \end{array} \right. \\ \textbf{else} & \left\langle \mathtt{S}_1, \alpha_1^0 \right\rangle \end{array} \right. \\ \textbf{else} & \left| \begin{array}{l} \textbf{if } 1 \leq \alpha_2^0 \leq n \\ \textbf{then } \left\langle \mathtt{S}_2, \alpha_2^0 \right\rangle \\ \textbf{else } - \end{array} \right. \end{array} \right.$$

We have several comments on this result:

- The loop indices of the possible sources are in $[1, n]$ but are not equal to n. This inaccuracy comes from the fact that we did not relate the loop indices for which the true branch of the conditional is executed and the loop indices for which the false branch is executed. Indeed, *at least* one of the branch of the conditional is executed for each iteration of the loop.

- The two parameters are compared and the case where they are equal is not excluded. According to the definition of the execution order $\prec$, the source is then the instance of the latest statement in textual order. In fact, the two branches of a conditional are not comparable according to the execution order, since *at most* one of the branches of the conditional is executed for a value of loop index. Therefore, the case where the two parameters are equal should not exist.

A structural analysis will give the necessary information to obtain a more precise source.

4.3.6 Example E3: Non-affine array subscript

Computing a parametric source for the read of $\mathtt{A}(y)$ in Example E3 is fairly simple, when no information is available concerning the non-affine terms. Consider the dependence between an instance of $\mathtt{S}_2$ and $\langle \mathtt{R}, y\rangle$. The set of linear constraints of $\mathsf{Q}_2^0(y)$ is:

$$\mathsf{L}_2^0(y) = \{x \,|1 \leq x \leq n\},$$

and is partitioned into the subsets

$$\hat{\mathsf{Q}}_2^0(x,y) = \{z \,|1 \leq z \leq n, z = x\},$$

for all $z \in \mathsf{D}_2(y)$. The environment is $1 \leq y \leq n$ and an elementary source is:

$$\hat{\varsigma}_2^0(x,y) = \textbf{ if } 1 \leq x \leq n \textbf{ then } \langle \mathtt{S}_2, x\rangle \textbf{ else } -.$$

A similar source is obtained from $\mathtt{S}_4$, and the parametric source is:

$$\sigma(y) = \left| \begin{array}{l} \textbf{if } 1 \leq \alpha_4^0(y) \leq n \\ \textbf{then } \left\langle \mathtt{S}_4, \alpha_4^0(y)\right\rangle \\ \textbf{else} \left| \begin{array}{l} \textbf{if } 1 \leq \alpha_2^0(y) \leq n \\ \textbf{then } \left\langle \mathtt{S}_2, \alpha_2^0(y)\right\rangle \\ \textbf{else } - \end{array}\right. \end{array}\right. .$$

Basically, this expression only says that the source either come from an instance of $\mathtt{S}_4$, from an instance of $\mathtt{S}_2$ or does not exist. An analysis of the variable `ii` may improve this result.

4.3.7 Example E4: Non-affine loop bounds

As for Example E3, the computation of the source of `A` in Example E4 leads to a very fuzzy result when no complementary analysis is used.

We study the dependence of depth 1 between an instance of $\mathtt{S}_3$ and $\langle \mathtt{R}, y\rangle$. As done previously, we explicitly represent the iteration vector of the implicit loops of statements $\mathtt{S}_3$ and `R`. The set of linear constraints is:

$$\mathsf{L}_3^1(y) = \{x \,|1 \leq x[1] \leq n, x[1] = y[1], x[2] = y[2]\}.$$

In the environment $1 \leq y[1] \leq n, 2 \leq y[2] \leq jmax(\langle \mathtt{R}, y\rangle)$, an elementary source is computed:

$$\hat{\varsigma}_3^1(x,y) = \textbf{ if } x = y \textbf{ then } \langle \mathtt{S}_3, y\rangle \textbf{ else } -.$$

Without any additional information, the dependence of depth 0 should be computed too, and the approximated source will be conservative but really inaccurate: the source may be an operation of statement $\mathtt{S}_3$ or may be $-$. A complementary analysis on the non-affine terms involved may improve the accuracy of the source.

4.4 Reducing Fuzziness

Our aim now is to find all interesting properties on the non-affine constraints. All these properties make up the set P, as defined in the previous section, and this is the first step of the computation of the approximate dataflow graph. Several techniques have been proposed to find properties on symbolic expressions. We first present a short review of the most used ones and then present two other methods in Sections 4.4.3 and 4.4.4. In order to make the other steps of the computation independent of the technique used to find the properties, we will consider general properties, as explained in Section 4.4.1.

4.4.1 General properties

Determining relations between symbolic expressions is undecidable in the general case. In practice, the techniques used to find relations between expressions restrain the kind of expressions analyzed and the kind of relations found and make conservative approximations. We present thereafter the best known methods. The application domains for which they have been developed range from dependence testing or array privatization to debugging. We mostly limit our discussion to their application domain and to the kind of relations exposed.

Relations between expressions of the program

The analysis described by Cousot and Halbwachs [33], based on an abstract interpretation framework, finds relations between the values of the variables of the program. It handles only affine expressions w.r.t. the variables of the program, the predicates of the conditionals are not taken into account and the use of a widening operator approximates the result whenever loops are treated. The relations between the values of variables are described with polyhedra. In the same framework, Granger [58] proposed a method to find congruence relations between the variables of the program. Masdupuy [87, 88] unified the two techniques (interval relations and congruence relations) with trapezoidal relations. Blume and Eigenmann [19] have proposed an extension of the method of Cousot and Halbwachs to polynomial expressions. However, their method manipulates intervals of values instead of polyhedra.

Another kind of analysis, relying on *Static Single Assignment*(SSA) form [37] focuses on the detection of equality relations among variables. The idea is to give different names to the different assignments to a variable. A ϕ-function is used to determine for each use the correct name of a variable when several possible control flows meet. Def-use chains are then embedded in the name of the variables. However, ϕ-functions ignore the *path conditions* guarding the execution of statements, and the comparison of expressions involving ϕ-functions seems therefore limited. *Gated Single Assignment*(GSA) [6] obviate this drawback by adding to the ϕ-functions the predicates of the meet. The value of

a variable is then represented as a symbolic expression using other variables, constants, ϕ-functions and predicates, which are other expressions. Tu and Padua [122, 124] described a demand-driven analysis that compares expressions, in GSA representation, using backward substitution. The relations exposed are inequalities. A relation between two expressions is established by their method only if the comparison of two expressions boils down to comparisons between variable names and trivial comparisons between path conditions. When the predicates are too complex to be handled, the value of variables are bounded. The main advantage of this method is that it does not rely on global forward substitution as most of the other techniques presented here. Hence only useful information is derived and more aggressive techniques can be implemented.

> Relations between the scalar variables of Example E4 page 101, can be found for instance by the demand-driven analysis proposed by Tu and Padua. They find that when statement `R` is executed, the values of $jlow$ and jup at an instance of statement S_3 verify the constraints: $jlow \leq 2$ and $jup \geq jmax - 1$. Such information allows them to conclude that all the elements accessed by an instance of `R` are defined by the instance of S_3 for the same iteration, hence privatization of the array `A` is possible. We will show that it is also allows the computation of the exact source of any element of `A` accessed by `R`.

Induction variable detection is a traditional method used in optimizers [1]. More complex techniques have been devised so as to detect induction variables with a non-constant increment [128, 67]. In [67], general induction expressions, which are not stored in any variable, can even be found. The form of the general induction variables or expressions detected is: $p(x) + ar^x$, where x is a loop counter , p is a polynomial function and a and r are constant w.r.t. the iteration vector. From such expression can then be deducted for instance monotony properties, constraints on the sign of induction variables or of the expressions in which they appear.

Properties on non-affine functions

Some techniques have been specifically designed to find properties on individual non-affine functions. Such is the case of the linearization method proposed by Maslov and Pugh [91] described in Section 3.4, page 90 and of the techniques presented by Dumay [47]. Dumay has proposed to derive from a system of non-affine constraints an approximate system of affine constraints. His method relies on a set of rules that associate for each kind of non-affine function (division, multiplication, exponentiation, ...) a new variable and some affine properties on this variable. The new variable then replaces the offending function in the expression in which it appears. The advantage of this approach is that the set of rules can be enlarged at will by the user, its drawback being that it targets more specifically non-affine expressions w.r.t the variables of the program rather than the linear expressions which are non-linear w.r.t. loop counters. Finally,

another method to collect properties on non-affine constraints is to ask the user to prove or disprove inequalities appearing among the dependence constraints or to insert some assertions in the program. (See [105] for example).

All the methods presented above find properties that are first order predicates. More precisely, the formalism used for the non-linear constraints leads us to consider two kind of properties. These properties are first order predicates and non-affine constraints are either represented by:

- Predicates, i.e. functions of boolean values;
- Or by inequalities in which non-affine functions appear.

The following two analyses gather relations between the non-affine constraints of the program in order to improve the output of the FADA. The first one, structural analysis, finds relations deducted from the structure of the program. The second method, iterative analysis, uses the results of fuzzy analysis in order to reduce the fuzziness of other fuzzy analyses. Both techniques represent non-affine constraints as predicates. But first of all, we give a characterization of the parameters of the maximum in the form of Presburger formulae.

4.4.2 Characterization of the parameters

The parameter of the maximum for the dependence between an instance of $\mathtt{S}_k$ and $\langle \mathtt{R}, y\rangle$ at depth p is defined by:

$$\alpha_k^p(y) = \max \mathsf{D}_k(y) \cap \mathsf{L}_k^p(y)_{|m_k},$$

when the intersection is not empty. Otherwise, $\alpha_k^p(y)$ is set to $-$. We translate this definition into a first order predicate: as $\alpha_k^p(y)$ is a maximum of a set when this set is not empty, this is equivalent to say that $\alpha_k^p(y)$ is an upper bound of this set and that $\alpha_k^p(y)$ belongs to the intersection or is equal to $-$. Formally, for all x,

$$l_k^p(x, y) \bigwedge_{i \in \mathsf{C}_k} c_i(x[1..e_i], y) \quad \Longrightarrow \quad x \ll \alpha_k^p(y), \tag{4.16}$$

where $l_k^p(x, y)$ is the characteristic function of $\mathsf{L}_k^p(y)_{|m_k}$,

$$(\alpha_k^p \neq -) \quad \Longrightarrow \quad l_k^p(\alpha_k^p(y), y) \bigwedge_{i \in \mathsf{C}_k} c_i(\alpha_k^p(y)[1..e_i], y). \tag{4.17}$$

Put into CNF, (4.16) and (4.17) are transformed into the clauses: for all x,

$$\neg l_k^p(x, y) \bigvee_{i \in \mathsf{C}_k} \neg c_i(x[1..e_i], y) \vee x \ll \alpha_k^p(y), \tag{4.18}$$

and for all $i \in \mathsf{C}_k$:

$$c_i(\alpha_k^p[1..e_i], y) \vee (\alpha_k^p = -) \tag{4.19}$$

and:

$$l_k^p(\alpha_k^p(y), y) \vee (\alpha_k^p(y) = -) \tag{4.20}$$

The properties (4.18) and (4.19) define relations between different candidate sources using the same non-affine constraints. (As for instance the candidate sources from the same statement for different dependence depths). (4.20) gives a property on $\alpha_k^p(y)$ when it is different from $-$. These properties will reduce the fuzziness of the source.

4.4.3 Structural analysis

In this section, we take into account the *structure* of the source program. That is, we exhibit the properties on non-linear constraints implied by some syntactical constructs. For the programs in the scope of FADA, the following information is available:

- The non-linear constraint governing the execution of the true branch of an `if..then..else..` construct is the negation of the non-linear constraint governing the other branch.
- If the body of a `while` loop is executed for an iteration, then it has been executed for all previous iterations of the loop, for the same iteration of surrounding loops. That means that the predicate of a `while` loop is true on a (possibly empty) interval of values of the loop index, beginning at 1 (the first value of the loop index).

We need to explicit the first property since the non-affine predicates of conditionals are represented in our formalism by two different non-affine constraints. In the formalism of our previous work [12], non-linear constraints were encoded in properties of their parameter domains. The relations of inclusion between the different parameter domains then had to be found by a recursive descent of the abstract syntax tree of the program. Such an analysis is no longer needed in our current formalism.

The structural analysis produces the following set of properties, derived from the structure of the program and from the environment.

- If c_i and c_j are two constraints generated by the non-linear predicate of an `if..then..else` construct, then add to the set of properties:

$$\forall x,\ c_i(x[1..e_i]) \Longleftrightarrow \neg c_j(x[1..e_j]). \tag{4.21}$$

- If c_i is the predicate of a `while` surrounded by n other loops, then:

$$\forall x_1, x_2,\; x_1[1..n] = x_2[1..n] \wedge x_2[n+1] \le x_1[n+1] \wedge c_i(x_1[1..e_i]) \Longrightarrow c_i(x_2[1..e_i]).$$

  We introduce the same property when c_i is one of the conditions ensuring that the loop counter of a `do` loop is between the bounds.

- For each non-linear constraint c_i verified by the read operation, add:

$$c_i(y[1..e_i], y). \qquad (4.22)$$

As hinted previously, the output of the analyses of Examples E1 and E2 can be improved with the relations produced by a structural analysis. We formulate here these relations for the two examples. How this information is integrated into the computation of the sources will be described in Sections 4.5 and 4.6.

`while` loop Applied to the `while` loop of Example E1, (4.22) is rewritten as:

$$\forall x, x',\; x[1] = x'[1] \wedge x[2] \le x'[2] \wedge c_1(x') \Longrightarrow c_1(x).$$

`if..then..else` construct Applied to Example E2, (4.21) is rewritten as:

$$\forall x,\; c_1(x) \Longleftrightarrow \neg c_2(x).$$

4.4.4 Iterative analysis

The key remark in this section is that two values of the same variable at two different steps of the execution are equal if they have the same source. Thanks to this remark, we will show that we may go one step further in dataflow analyses. That is, that the result of a first application of the FADA analysis may in turn help a second application in deriving a more precise result.

To see this, suppose that the same array occurs in the left hand-side of two statements, with the same variable as subscript. This variable is supposed not to depend linearly on induction variables. Making no assumptions on the values of variables, we may, however, try to prove that whatever the values of this variable, they are equal for some instances of the two statements. As hinted above, we may apply a dataflow analysis on the subscripting variables themselves, thus iterating the overall process of the analysis. Similarly, two constraints that are the same function but appear at different places in the program have the same value if the variables they use are the same and have the same values.

Therefore, the purpose of iterative analysis is to find relational properties (most of all, equality) between the non-linear constraints appearing in the existence predicates and in the conflicting access constraints of different statements.

This method may use the results of dataflow analysis on the variables of the non-linear constraints so as to find more accurate relations. As this dataflow analysis can be fuzzy, the method can then be applied once more and eventually the fuzziness will be reduced by successive analyses. This iterative analysis is only based on the DFG and does not examine the right-hand sides of the assignments. The equality of the sources of variable at two different steps of the execution is only a sufficient condition to prove the equality of the values taken by this variable. Deciding that different variables have the same value at different points in the program is another, more difficult problem and is not tackled in this thesis.

We first describe sufficient conditions ensuring the equality of two non-affine constraints for two operations. The steps of the analysis that checks these conditions are then detailed and we also study some possible variants of the conditions. Finally, we conclude by a description of the two different methods to perform an iterative analysis.

Equality of two non-affine constraints

To formalize the previous paragraph, suppose that we want to compute the source of $\mathtt{A}(g(y))$ at operation $\langle \mathtt{R}, y\rangle$ and that there appears in the computation of the source two non-affine constraints, c_i and c_j, represented as predicates. Our purpose is to decide whether the value of $c_i(x_i, y)$ is the same as the value of $c_j(x_j, y)$, given the operations $\langle \mathtt{T}_i, x_i\rangle$, $\langle \mathtt{T}_j, x_j\rangle$ and $\langle \mathtt{R}, y\rangle$, so as to possibly reduce the fuzziness of the source. So far, constraints have been defined as functions of iteration vectors. As a matter of fact, a constraint depends on variables that are functions of the iteration vector, as noticed in Section 4.3.1, just after (4.6). Let $\mathtt{t}_{ik}$, for $1 \leq k \leq v_i$ be the names of the variables used in c_i and $\mathtt{t}_{jk}$ be the names of the variables used in c_j. $t_{ik}(x, y)$ denotes the value taken by the variable $\mathtt{t}_{ik}$ at operation $\langle \mathtt{T}_i, x\rangle$ or $\langle \mathtt{R}, y\rangle$ depending on the statement in which it appears (similar notations for the values of $\mathtt{t}_{jk}$). c_i is therefore the composition of a function c_i^* with the functions t_{ik}:

$$c_i = c_i^* \circ (t_{i1}, \ldots, t_{iv_i}).$$

The following property defines a sufficient condition for which c_i equals c_j:

Property 4.1 $c_i(x_i, y) = c_j(x_j, y)$ *holds if:*

- c_i^* *and* c_j^* *define the same function (perhaps because they are syntactically equal).*
- *The exact (but possibly parametric) source of* $\mathtt{t}_{ik}$ *at operation* $\langle \mathtt{T}_i, x_i\rangle$ *and of* $\mathtt{t}_{jk}$ *at operation* $\langle \mathtt{T}_j, x_j\rangle$ *are the same.*

Proof As the sources of the variables are the same, they have the same value. The composition of the same functions (c_i^* and c_j^*) with variables of the same values entails that $c_i(x_i, y) = c_j(x_j, y)$.

Two non-affine expressions are in relation if two conditions are verified, according to Property 4.1. We detail thereafter how these conditions can be tested.

Same sources

This step consists in the construction of the conditions for which the sources of $\mathtt{t}_{ik}$ at operation $\langle \mathtt{T}_i, x_i\rangle$ and of $\mathtt{t}_{jk}$ at operation $\langle \mathtt{T}_j, x_j\rangle$ are the same. First, we can check syntactically that the names of the variables are the same. Then, consider a leaf $\langle \mathtt{U}_{ik}, u_{ik}(x_i)\rangle$ of $\sigma(\mathtt{t}_{ik}, \langle \mathtt{T}_i, x_i\rangle)$ governed by the conditions $s_{ik}(x_i)$ and a leaf $\langle \mathtt{U}_{jk}, u_{jk}(x_j)\rangle$ of $\sigma(\mathtt{t}_{jk}, \langle \mathtt{T}_j, x_j\rangle)$ governed by the conditions $s_{jk}(x_j)$. The two variables have the same source when the condition

$$s_{ik}(x_i) \wedge s_{jk}(x_j) \wedge (\mathtt{U}_{ik} = \mathtt{U}_{jk}) \wedge (u_{ik}(x_i) = u_{jk}(x_j))$$

is met. By doing so for each pair of leaves with the same statement, the condition for which the two variables have the same source is the disjunction of all conditions found for each pair. Let $src(x_i, x_j)$ be this condition. During the computation of the sources of the variables, we may find other non-affine constraints and compare them with another iterative analysis.

Comparing constraints c_i^* and c_j^*

Detecting that $c_i^*(x_i, y) = c_j^*(x_j, y)$ is a syntactic test. More complicated tests using for instance the normalization of both constraints have not been considered. If the equality is verified, then the iterative analysis produces the relation:

$$\forall x_i, x_j, src(x_i, x_j) \Longrightarrow (c_i(x_i, y) \Longleftrightarrow c_j(x_j, y)).$$

The computation of the source of the array element $\mathtt{A}(y)$ used by operation $\langle \mathtt{R}, y\rangle$ in Example E3 page 101 takes advantage of an iterative analysis.

First iterate The constraints $c_1^*(x_1, x_2) = (x_1 = x_2)$ and $c_2^*(x_1, x_2) = (x_1 = x_2)$ are equal and the variables used in c_1 and c_2 are `ii` and y in both cases. We will thus first apply a dataflow analysis to `ii` in order to find the conditions for which `ii` has the same value in $\mathtt{S}_1$ and in $\mathtt{S}_3$.

Second iterate For operation $\langle \mathtt{S}_1, x\rangle$, the exact source of `ii` is

$$\textbf{if } x \geq 2 \textbf{ then } \langle \mathtt{S}_2, x-1\rangle \textbf{ else } \langle \mathtt{S}_0, []\rangle.$$

For operation $\langle \mathtt{S}_3, x'\rangle$, the source is $\langle \mathtt{S}_2, x'\rangle$. The two sources are the same when:

$$(x' = x - 1) \wedge (x \geq 2).$$

Back to first iterate We will therefore make the computation of the source of $\langle \mathtt{R}, y\rangle$ with the property:

$$(x' = x-1) \wedge (x \geq 2) \Longrightarrow (c_1(x,y) \Longleftrightarrow c_2(x',y))$$

Equality is not the only relation that can be found between c_i^* and c_j^*. Let us now examine a more general case where constraints c_i^* and c_j^* are different but there exists some function e such that

$$c_i^* = e \circ c_j^*. \tag{4.23}$$

As c_i^* and c_j^* are functions of boolean values, the only non-trivial boolean function e is $\neg$. The method proceeds in the same manner for this case as for the equality relation.

Symmetrically, the constraints c_i^* and c_j^* may be related with the following equation:

$$c_i^* = c_j^* \circ e. \tag{4.24}$$

From a practical point of view, c_i^* and c_j^* have to be inequalities on affine functions in order to be able to detect this relation. All possible affine functions e verifying this equality are then found by Gaussian elimination.

So as to reuse previous results, our aim is to find a function f such that

$$e \circ (t_{j1}, \ldots, t_{jv_j})(x_j, y) = (t_{j1}, \ldots, t_{jv_j})(f(x_j), y).$$

That means that the value of some variables of the program for an operation (with iteration vector $f(x_j)$) are functions of the values of the same variables for another operation (with iteration vector x). Since this expression is the formal definition of a recurrence as given by Redon [111], this problem boils down to the detection of a recurrence on the variables $\mathtt{t}_{jk}, \mathtt{r}_{jk}$. If a recurrence is detected, then we have found two functions e_r and f_r verifying:

$$e_r \circ (t_{j1}, \ldots, t_{jv_j})(x_j, y) = (t_{j1}, \ldots, t_{jv_j})(f_r(x_j), y).$$

Indeed, the function e_r in the expression of the recurrence may not correspond to the function e verifying (4.24). If $e = e_r$ then the function f is equal to f_r. If $e \neq e_r$, it may be interesting to look for an integer n such that $e = e_r^n$. But this seems difficult when e_r is more complex than a translation, so we assume that there is no relation between c_i^* and c_j^* when $e \neq e_r$. Notice that detecting recurrences requires the computation of a dataflow graph, hence additional iterative analyses and recurrence detections.

We now have the following equality:

$$\begin{aligned} c_i^* \circ (t_{i1}, \ldots, t_{iv_i})(x_i, y) &= c_j^* \circ e \circ (t_{j1}, \ldots, t_{jv_j})(x_j, y) \\ &= c_j^* \circ (t_{j1}, \ldots, t_{jv_j})(f(x_j), y) \\ c_i(x_i, y) &= c_j(f(x_j), y). \end{aligned}$$

We can then check the relation between c_i and c_j in the same manner as before.

How to perform an iterative analysis

The process of finding the source of a variable to reduce the fuzziness of the computation of another source may not terminate. Indeed, this may happen in programs using for instance `A(A(x))`. Such a case can be detected by building a graph of the analyses. There is an edge from the analysis of `A` in statement `S` to the analysis of `B` in statement `T` iff `S` is a write into `B` and `A` is involved in a non-affine constraint of the parameter domain of the dependence between `S` and `T`. Analyses should be carried according to a topological sort of this oriented graph. Cycles in this graph indicate potentially non terminating analyses and should be avoided, that is, analyses are not allowed to iterate on a cycle. Whether the computation of a fixed point is possible is still uncertain.

More generally, iterative analysis can be computed in two manners:

- All the analyses of the graph of analyses are performed. This is the case considered above: the analyses are carried according to a linearization of the graph. Iterative analysis only reuses the results produced by preceding analyses and its cost is therefore limited.

- Only the analysis of a variable is needed. In that case, iterative analysis can be considered as a demand-driven analysis, and all the analyses preceding the desired analysis in the graph may be necessary.

4.5 Translating Properties

Once properties about non-linear constraints have been collected, they need to be converted into properties about parameters of the maximum. This is the second step of the computation of the fuzzy source, presented in Section 4.3.3. These new properties $\hat{\mathsf{P}}$, consequences of the properties P of non-affine constraints, will then reduce the set of possible sources. It is important to know when every parameter verifying $\hat{\mathsf{P}}$ is the parameter of the maximum associated to some constraints verifying P. We will then say that *no fuzziness is added*. As long as this is the case, gathering more information about non-linear constraints will improve the accuracy of the source. With such information, it may be possible to decide whether it is interesting to use a more complex method to find non-linear constraint relations. When some fuzziness is introduced during the translation of P into $\hat{\mathsf{P}}$, such decision is no longer possible.

This section proposes a method to derive properties on the parameters from properties on non-affine constraints represented either as symbolic predicates or as inequalities involving non-affine functions. The technique used in the latter case is more general than in the first and uses the results obtained by the first technique.

4.5.1 The resolution method

The idea is to use a subset of Robinson's resolution [113] in order to transform formulas with non-linear constraints into formulas in which they do not appear. We first give a short introduction to the notions of computer logic used in the sequel. See [118] for instance for a more complete introduction to the subject.

Any formula can be put into a universal conjunctive normal form, possibly resorting to Skolem functions (x denotes a vector of variables):

$$\forall x, \bigwedge_{i}^{n} r_i(x)$$

where $r_i(x)$, called a *clause*, is the disjunction of atomic formulae or negation of atomic formulae. A *literal* is defined as an atomic formula or the negation of an atomic formula. A *non-linear literal* is a non-linear constraint or its negation. Other literals are called *linear literals*. We use in the sequel the following general rule, known as *simple resolution* rule:

Rule 4.1 (simple resolution) *Given two clauses $r(x) \vee c(f(x))$ and $s(x') \vee \neg c(g(x'))$ where $c(f(x))$ and $\neg c(g(x'))$ are two literals, f and g two affine functions and x and x' universally quantified variables, we derive the clause:*

$$f(x) = g(x') \Longrightarrow s(x') \vee r(x).$$

The literal $(f(x) = g(x'))$ represents the *unification* condition between $c(g(x'))$ and $c(f(x))$. The resolution is a valid inference rule, that is, the derived clause $s(x') \vee r(x) \vee (f(x) \neq g(x'))$ is a logical consequence of $r(x) \vee c(f(x))$ and $s(x') \vee \neg c(g(x'))$.

The general method consist in applying repeatedly Rule 4.1 on the clauses of P or on the clause obtained by resolution so as to eliminate positive and negative non-linear literals. The clauses produced this way, with only linear literals, define $\hat{\mathsf{P}}$. The validity of resolution ensures that all these clauses are consequences of the clauses of P, therefore the real, unknown, source belongs to the set of sources (4.15).

This approach is exactly the same as the one used for the refutation of a system of clauses, in logical programming. The refutation consists in proving that a system of clauses implies a contradiction and one method to do it is to build the empty clause (the clause that is always false) by successive application of a resolution rule. Indeed, the completeness of the predicate calculus ensures that if a system of clauses is not satisfiable, then a refutation can be obtained by applying the simple resolution rule successively to the initial clauses or to the clauses obtained by derivation. In our case, the completeness of the predicate calculus will help us to prove the condition for which no fuzziness is added from P to $\hat{\mathsf{P}}$. Indeed, when replacing two clauses by one clause deducted from them, it is likely that some information is lost. So as to preserve accuracy, one

method consists in building all possible linear clauses that are consequences of the clauses of P. This is equivalent to the computation of all possible refutations. The following theorem shows that this method does not add fuzziness.

Theorem 4.1 *No fuzziness is added from* P *to* $\hat{\mathsf{P}}$ *if and only if all linear clauses derived by resolution from the clauses of* P *can be derived from the clauses of* $\hat{\mathsf{P}}$.

> **Proof Sufficient condition—the "if" part** All linear clauses derived by resolution from the clauses of P can be derived from the clauses of $\hat{\mathsf{P}}$. Suppose that for a value of the parameters verifying $\hat{\mathsf{P}}$, there does not exist any non-linear constraint verifying P. P is then not satisfiable. Thanks to the completeness of the predicate calculus, the empty clause can therefore be derived by resolution from P. By hypothesis, the empty clause can also be derived from $\hat{\mathsf{P}}$, thus would be a consequence of $\hat{\mathsf{P}}$ which is satisfiable. This cannot happen, hence we conclude that no fuzziness is added.
>
> **Necessary condition—the "only if" part** We show that when some linear clauses derived by resolution from P cannot be obtained by derivation of clauses of $\hat{\mathsf{P}}$, then some fuzziness is added. Consider a linear clause derived from P which cannot be derived from $\hat{\mathsf{P}}$ and which is not a tautology. This clause is not a consequence of the clauses of P, therefore there exists a value of the parameters such that they verify any clause of $\hat{\mathsf{P}}$ and not this distinguished clause. The empty clause can be derived from P and P is thus contradictory: There exists some parameters verifying $\hat{\mathsf{P}}$ which are not parameters of the maximum for any constraints verifying P. Some fuzziness is added, and this proves the theorem.

When this condition is satisfied, the set of sources (4.15) does not contain sources that are not accessible, given the properties on the non-linear constraints. (4.15) is then the exact set of sources associated to all possible constraints verifying P.

Testing the satisfiability of the set of clauses P is undecidable, since non-affine constraints are symbolic functions [45]. On the other hand, testing the satisfiability of the set of affine clauses $\hat{\mathsf{P}}$ is decidable. Thus building $\hat{\mathsf{P}}$ from P may not be possible in finite time. This problem can be easily illustrated. Consider the two clauses $\forall x, \neg c(x) \vee c(x+1)$ and $c(0)$. Repeatedly combining the derived clause with the first of the two clauses, we obtain the sequence: $\forall x_1,\ c(x_1+1) \vee (x_1 \neq 0)$, $\forall x_1, x_2,\ c(x_2+1) \vee (x_2 \neq x_1+1) \vee (x_1 \neq 0)$, etc. In other words, we derive all the clauses $c(x+1)$, for all $x \in \mathbb{N}$. On this simple example, it is clear that the two clauses are equivalent to the unique clause $\forall x, x \geq 0 \Longrightarrow c(x)$. But in general, it can be shown that such reformulation is equivalent to the problem of recurrence detection, which is undecidable.

Note that the undecidability problem can be removed for many kinds of relations on non-affine constraints. When the arguments of all non-affine constraints are universally quantified variables and if these variables are either equal or not related through affine constraints (meaning that the set of the values that universally variables can take must be a Cartesian product of intervals), then the problem of resolution boils down to a problem of resolution in propositional calculus. Hence, the translation of P into $\hat{\mathsf{P}}$ takes finite time in this case. Relations produced by structural analysis and by most symbolic analyses only relate the values of variables for the same value of the iteration vector, therefore the construction of $\hat{\mathsf{P}}$ takes finite although exponential time.

The use of theorem proving techniques may seem a priori too expensive for a compile-time technique. Type checking boils down, too, to theorem proving (thanks to the Curry-Howard isomorphism) and the many implementations of functional languages with complex type checking show that these techniques are not incompatible with reasonable compiling times. In a functional language such as ML, the same resolution methods with unification as the ones we use are applied. Although type checking do not resort to theorem proving on first order arithmetic predicates, a good adequation between the kind of constraints and the resolution method involved avoid a prohibitive computation cost (this is the case of structural properties and input linear resolution for instance, as shown in the following section). Moreover, if the cost of building the set $\hat{\mathsf{P}}$ with the minimal amount of fuzziness is too high, it is always possible to stop the derivation of clauses after a fixed amount of time. The result of FADA will still be conservative.

In the sequel, we present simplified resolution methods that do not build all derived clauses of P and therefore add some fuzziness. We then show that these techniques do not add fuzziness when the clauses of P have a particular form which happens to be the most frequently encountered one.

4.5.2 Boolean constraints

Let us apply the preceding results to the construction of $\hat{\mathsf{P}}$ when non-linear constraints are symbolic boolean functions. All predicates of P are put into universal conjunctive normal form. Most of the functions and parameters that appear in the following depend on y. For the sake of clarity, we will drop the notation of this argument since y does not have any active role in this section.

We present a simple algorithm that builds $\hat{\mathsf{P}}$ from P. As explained in the previous paragraph, this algorithm may introduce some fuzziness, but it seems to be sufficient for the kind of constraints we are able to find up to now.

The two following resolution rules are applied repeatedly on the clauses of P and it is assumed that the characterization of parameters of the maximum (4.18), (4.19) and (4.20) presented page 117 belong to P. The first rule uses (4.18) to eliminate non-linear constraints appearing as positive literals and the

second uses (4.19) to eliminate constraints appearing as negative literals.

Rule 4.2 *In a clause* $c_i(f_i(x)) \vee r(x)$*, the term* $c_i(f_i(x))$*, where* $f_i(x)$ *is an affine function and* x *a universally quantified variable, is replaced by the clauses: for all* x*, for all* x' *such that* $x'[1..e_i] = f_i(x)$*, for all* k *such that* $i \in \mathsf{C}_k$*, for all* p*,*

$$\neg l_k^p(x') \bigvee_{j \in \mathsf{C}_k - i} \neg c_j(x'[1..e_j]) \vee x' \ll \alpha_k^p, \tag{4.25}$$

and, for all x, x'

$$s(x') \vee (f_i(x) \neq g_i(x')), \tag{4.26}$$

where $\neg c_i(g_i(x')) \vee s(x')$ *is another clause of* P *different from the clauses characterizing parameters.*

If there does not exist any k such that $i \in \mathsf{C}_k$ then (4.25) is replaced by true. If there does not exist another clause of P, then true is derived instead of (4.26).

(4.25) is derived from (4.18) and $c_i(f_i(x))$.

Rule 4.3 *In a clause* $\neg c_i(g_i(x)) \vee r(x)$*, the term* $\neg c_i(g_i(x))$*, where* $g_i(x)$ *is an affine function, is replaced by the clauses: for all* k *such that* $i \in \mathsf{C}_k$*, for all* p*,*

$$(\alpha_k^p = -) \vee (g_i(x) \neq \alpha_k^p[1..e_i]), \tag{4.27}$$

and

$$s(x') \vee (f_i(x') \neq g_i(x)), \tag{4.28}$$

where $c_i(f_i) \vee s(x')$ *is another clause of* P *different from the clauses characterizing the parameters.*

If there does not exist any k such that $i \in \mathsf{C}_k$ then (4.27) are replaced by true. If there does not exist another clause of P, then true is derived instead of (4.28).

(4.27) is derived from (4.19) and $\neg c_i(g_i(x))$.

From these two rules, we build the following algorithm:

Input: Properties P

Output: Properties $\hat{\mathsf{P}}$

1. Apply Rules 4.2 and 4.3 repeatedly on the clauses of P. Application can be performed in any order. Each time two clauses of P are combined together to eliminate a literal, tag the resulting clause with this literal and with the tags of the two initial clauses so as to prevent cycling. Do not eliminate a literal in a clause that has been tagged by it.

2. Replace in the resulting set of clauses the parameters of the maximum α_k^p by free variables x_k^p.

3. Assign to $\hat{\mathsf{P}}$ the subset of linear clauses.

Lemma 4.1 *The algorithm terminates and $\hat{\mathsf{P}}$ does not depend on the order of applications of Rules 4.2 and 4.3.*

> **Proof** As hinted at the beginning of this section, Rule 4.2 substitutes positive non-linear literals by (4.25), making the number of positive non-linear literals in the resulting clauses strictly decreasing. Rule 4.3 substitutes negative literals by (4.27), which does not contain any non-linear literal. Moreover, clauses of P cannot be combined altogether indefinitely thanks to the conditions in the algorithm. Thus no cycling is possible and the algorithm terminates.
>
> Moreover, the order of application of the rules does not change $\hat{\mathsf{P}}$ because the non-linear literals are handled separately from each other and the substitution of a literal has no impact on the other literals of the clause.

The set of clauses produced by this algorithm defines $\hat{\mathsf{P}}(\ldots, x_k^p, \ldots)$. As the clauses defining $\hat{\mathsf{P}}$ are consequences of P, the source set (4.15) defined page 109 contains at least the sources corresponding to all possible non-linear constraints verifying P.

The resolution method of the previous algorithm derives new clauses only by combination of two clauses of P or by combination of a derived clause and of a clause of P. This technique is known as *Input Linear Resolution* (ILR) [86]. In general, ILR is not complete (it is not always possible to find a refutation even if the set of clauses is not satisfiable). Our algorithm is therefore unable to produce a set $\hat{\mathsf{P}}$ such that all linear clauses derived from clauses of P can be derived from the clauses of P. Thus we may introduce in general some fuzziness in $\hat{\mathsf{P}}$. However, we will see that no fuzziness is added for all the four examples presented in Section 4.2.2 page 100.

4.5.3 Constraints as inequalities on non-affine functions

We now assume that each non-linear constraint c_i used in the computation of a source function can be written as an inequality (or system of inequalities) as denoted by (4.6). In this section, we drop the notations of the restrictions on the vector sizes.

Our aim here is to propose an algorithm that builds a set $\hat{\mathsf{P}}$ whose clauses are consequences of the clauses of P. We assume that the clauses (4.18), (4.19)

and (4.20) characterizing the parameters α_k^p belong to P. In order to simplify the expressions manipulated, we will suppose that each non-linear constraint c_i involves only one non-linear rational function. The clauses (4.18), (4.19) are therefore written as:

$$\neg l_k^p(x) \bigvee_{i \in \mathsf{C}_k} (-a_i(x) - \beta_i n_i(x) - 1 \geq 0) \vee x \preceq \alpha_k^p, \tag{4.29}$$

and $\forall i \in \mathsf{C}_k, \forall x$,

$$(\alpha_k^p = -) \vee \big(a_i(\alpha_k^p) + \beta_i n_i(\alpha_k^p) \geq 0\big) \tag{4.30}$$

As in Section 4.5.1, $\hat{\mathsf{P}}$ is built as a consequence of the clauses of P and of the clauses characterizing the parameters of the maximum. The clauses of P may contain inequalities with non-linear terms, which are neither the inequalities appearing in (4.29), or in (4.30) nor their opposite. We therefore need some resolution rules that are more precise than in the case of general first order predicates. We first find these rules on a simplified example, and then generalize them.

Study of a simplified example

Suppose we have two clauses, $\forall x, r(x) \vee c(x)$ and $\forall x, r'(x) \vee c'(x)$ where

$$c(x) = (a(x) + \sum_k \beta_k n_k(x) \geq 0)$$

and

$$c'(x) = (a'(x) + \sum_k \beta'_k n_k(x) \geq 0)$$

with a and a' quasi-affine forms, n_k non-linear terms and β_k and β'_k integer coefficients. We want to find some formulae that are consequences of these clauses and that do not use any non-linear term. The conjunction of the two clauses is equivalent to the formula:

$$\forall x, x', (c(x) \wedge c'(x')) \vee (c(x) \wedge r'(x')) \vee (c'(x') \wedge r(x)) \vee (r(x) \wedge r'(x')).$$

Thus, $\forall x, (c(x) \wedge c'(x)) \vee r(x) \vee r'(x)$ is a consequence of the initial clauses. Given an x, the values of the $n_k(x)$s are unknown. According to Farkas theorem [95], the system $c(x) \wedge c'(x)$ with the unknowns $n_k(x)$ has a solution if and only if:

$$\forall \gamma, \gamma' \in \mathbb{N} \text{ such that } \forall k, \gamma\beta_k + \gamma'\beta'_k = 0, \text{ then } \gamma a(x) + \gamma' a(x') \geq 0$$

If we can find a vector $(\gamma_0, \gamma'_0) \geq 0$ verifying $\gamma_0\beta_k + \gamma'_0\beta'_k = 0$ for all k, then the conjunction $c(x) \wedge c'(x)$ is equivalent to $\gamma_0 a(x) + \gamma'_0 a(x') \geq 0$. Indeed, any

vector (γ, γ') verifying $\gamma\beta_k + \gamma'\beta'_k = 0$ is collinear to (γ_0, γ'_0). Finally, if a vector (γ_0, γ'_0) exists, then we consider the formula:

$$\forall x, (\gamma_0 a(x) + \gamma'_0 a(x') \geq 0) \vee r(x) \vee r'(x).$$

In the other cases , we consider the formula: true. In both cases, we obtain a consequence of the first two clauses.

General case

In general, $\hat{\mathsf{P}}$ is deducted from P as follows. Consider the clauses of P and a term

$$\forall x, e_j(x) = a'_j(x) + \sum_i \beta'_{ij} n_i(f_{ij}(x)) \geq 0 \tag{4.31}$$

of the jth clause. By anding and unifying these terms with (4.29), (4.20) and (4.30), a system of inequalities can be built in the same manner as in the previous example:

$$\begin{cases} \forall j, a'_j(\ldots) + \sum_i \beta'_{ij} n_i(\ldots) \geq 0 \\ \forall i, a_i(\ldots) + \beta_i n_i(\ldots) \geq 0 \text{ or } -a_i(\ldots) - \beta_i n_i(\ldots) - 1 \geq 0 \end{cases} \tag{4.32}$$

The universally quantified variables have possibly been renamed, the arguments of the functions have been unified and the conditions of the unification will be added to the final result. The notation ... signifies that the argument is the same as the one appearing in (4.31), (4.29), or (4.19), with a possible renaming of the quantified variables. The choice between an inequality or its opposite depends whether the construction of the system incorporates (4.29) or (4.30).

According to Farkas, the system (4.32) has a solution if and only if:

$$\forall \gamma'_j \in \mathbb{N}, \gamma_i \in \mathbb{Z} \text{ such that } \forall i, \sum_j \gamma'_j \beta'_{ij} + \gamma_i \beta_i = 0,$$

$$\text{then } \sum_j \gamma'_j a'_j(\ldots) + \sum_i (\gamma_i a_i(\ldots) - \gamma_i \delta_-(\gamma_i)) \geq 0,$$

with $\delta_-(x) = 1$ if $x < 0$, 0 otherwise.

If a base of vectors $(\ldots, \gamma'_{jk}, \ldots, \gamma_{ik}, \ldots)_k$, denoted $\mathrm{b}(\gamma_{ik}, \gamma'_{jk})$, verifying $\forall i, k, \sum_j \gamma'_{jk} \beta'_{ij} + \gamma_{ik} \beta_i = 0$ exists, then:

$$\bigwedge_k \left(\sum_j \gamma'_{jk} a'_j(\ldots) + \sum_i (\gamma_{ik} a_i(\ldots) - \gamma_i \delta_-(\gamma_i)) \geq 0 \right). \tag{4.33}$$

Note that this conjunction is a logical consequence of the system (4.32). Let u denotes this system of equations corresponding to the unification conditions. We replace the system (4.32) by the clauses: for all k,

$$\neg u \bigvee \left(\sum_j \gamma'_{jk} a'_j(\ldots) + \sum_i (\gamma_{ik} a_i(\ldots) - \gamma_i \delta_-(\gamma_i)) \geq 0 \right), \tag{4.34}$$

When $\gamma_{ik} > 0$, then (4.30) was used to obtain (4.34). When $\gamma_{ik} < 0$, then (4.29) was used instead. If $\gamma_{ik} = 0$ then $c_i(x)$ was of no use in (4.34) for the kth vector of the base. Likewise, when $\gamma'_{jk} = 0$, the jth clause was not used.

If there is no base $\mathrm{b}(\gamma_{ik}, \gamma'_{jk})$ then (4.32) is replaced by true. Likewise, if there is a non-linear term in a clause of P that do not appear in any clause (4.29) or (4.30), then it is replaced by true.

The following rule sums up the different steps of the transformation.

Rule 4.4 *Given a set of clauses* $(e_j(x_j) \geq 0) \vee s_j$, $\forall j$, *if the base* $\mathrm{b}(\gamma_{ik}, \gamma'_{jk})$ *associated to the* $e_j(x)$ *exists, then we build the predicates:* $\forall k$,

$$\neg u \bigvee \left(\sum_j \gamma'_{jk} a'_j(\ldots) + \sum_i (\gamma_{ik} a_i(\ldots) - \gamma_i \delta_-(\gamma_i)) \geq 0 \right) \bigvee_{j|\gamma'_{jk}>0} s_j \bigvee_{i|\gamma_{ik}>0} r_i^+ \bigvee_{i|\gamma_{ik}<0} r_i^-,$$

with u *the unification conditions,* r_i^+ *the conjunction of all the clauses* (4.30) *for all* k *and* p *such that* $i \in \mathsf{C}_k$, *minus the terms involving* c_i. *Symmetrically,* r_i^- *is the conjunction of the clauses* (4.18) *and* (4.20) *for all* k *and* p *such that* $i \in \mathsf{C}_k$, *minus the terms involving* c_i. *and with the quantified variables renamed appropriately.*

The algorithm we propose relies on the same principle as the previous one: a variant of ILR, limited to one combination of clause by literal so as to ensure termination.

Input: Properties P

Output: Properties $\hat{\mathsf{P}}$

1. Apply Rule 4.4 repeatedly on the clauses of P. Application can be performed in any order. Each time some clauses of P are combined together to eliminate a term, tag the resulting clause appropriately so as to prevent cycling.

2. Replace in the resulting set of clauses the parameters of the maximum α_k^p by free variables x_k^p.

3. Assign to $\hat{\mathsf{P}}$ the subset of linear clauses.

It can easily be shown that this set of clauses do not depend on the way the steps are applied and is obtained after a finite number of applications. The clauses of $\hat{\mathsf{P}}$ are consequences of the clauses of P.

4.5.4 Simplifications

Both algorithms proposed to build $\hat{\mathsf{P}}$ from P have at worst exponential time and exponential space requirements w.r.t the number of literals in P. Thus it seems important to be able to do some simplifications whenever possible. The two algorithms are based on ILR and combine derived clauses with the clauses of P. Two simplifications can therefore be examined:

- When the empty clause is derived, we stop combining clauses of P with it. But the empty clause should not be produced, since P is supposed to be satisfiable.
- When a derived clause is a tautology, we stop combining clauses with it. Indeed, all derived clauses would then be tautologies too.

Some tautologies are easily detected: Any clause can be written as $r \Longrightarrow s$, where r is the conjunction of all linear positive literals. Note that all the unification conditions are then in r. This clause is a tautology if r is the empty clause, that is, if the polyhedron defined by r is empty. Testing the emptiness can be achieved by simply comparing parallel constraints induced by unification conditions. This simplification saves time. Another kind of simplification, saving space, consists in eliminating all redundant inequalities in r.

4.5.5 A particular case: Exact results

There is one interesting particular case: When $y[1..m_k] \in \mathsf{L}_k^p(y)_{|m_k} \cap \mathsf{D}_k(y)$, that is, when $l_k^p(y[1..m_k], y) = \text{true}$ and $\bigwedge_{i \in \mathsf{C}_k} c_i(y, y)$, by combination with (4.18), the inequality $y[1..m_k] \leq \alpha_k^p$ is verified. This implies that $\alpha_k^p \neq -$, and with (4.20) we conclude that $\alpha_k^p = y[1..m_k]$. The source set is therefore a singleton if α_k^p is the only parameter that remains in the expression of the source.

For instance, our analysis combined to a structural analysis computes the exact source of `A` for statement `R` in the following example:

```
  do x1 = 1 while (P(x1))
   do x2 = 1, n
S   A(x2) = ..
   enddo
   do x2 = 1, n
R   .. = A(x2)
   enddo
  enddo
```

Indeed, here the non-linear constraint is $c_1(x1) = (P(x1) = \text{true})$. A structural analysis proves that if y is the iteration vector of the read, $c_1(y[1])$. Moreover, y verifies all the linear constraints that must check the iteration vector of the source for the dependence at depth 1 between an instance of S and $\langle \mathtt{R}, y\rangle$. The source is therefore exact and equal to $\langle \mathtt{S}, y\rangle$.

4.5.6 Conclusion

Translating properties on non-affine constraints into properties on parameters is a difficult task when the first properties are general first order predicates. We have shown in this section that the translation boils down to the application of theorem proving algorithms. When the translation preserves the preciseness of the properties, the approximation of the source can be optimal w.r.t. the information available on non-affine constraints. In spite of the fact that optimality is out of reach in the general case, a large class of common properties on non-affine constraints can be handled with no loss of information.

In the following section, we show how to use the deducted properties on parameters so as to build a fuzzy source. In section 4.8.3, we show that taking into account properties on non-affine properties is difficult and it boils down to the methods presented in this section, whether or not parameters are used.

4.6 Building Source Sets

The parameters of the maximum verify the properties $\hat{\mathsf{P}}$ that have been derived from the properties P of the non-affine constraints. The last and third step of the computation of the fuzzy source, as described in Section 4.3.3, consists in using these affine properties so as to simplify the expression of the fuzzy source. The expression of the source set may be transformed in two ways: first, by integrating in the expression of the source the clauses of $\hat{\mathsf{P}}$, then by removing the parameters of the maximum from the expression of the predicates of the source quast.

4.6.1 Context quasts

The clauses of $\hat{\mathsf{P}}$ may still contain universally quantified variables. Hence the expression of the source set may not be easy to manipulate. One solution consists in transforming this set of clauses into a quast, called *context quast*, without loss of information, and then in combining this quast with the parametric quast of the source.

A context quast is a quast whose only purpose is to enrich the context of the normal quast with which it is combined. Its leaves are either $-$, meaning that the context added is only the conjunction of the conditionals from the root to this leaf, or a polyhedron. In this latter case, the polyhedron is added to the current context of the quast computation. Two context quasts can also

be combined, the rules are the same as for normal quasts, excepted for the combination of two leaves different from $-$: the result is then the intersection of the two polyhedra. To sum up, two context quasts are combined according to Rules 3.1, 3.2 and:

Rule 4.5 *If τ_1 and τ_2 are two polyhedra then:*

$$\max(\tau_1, \tau_2) = \tau_1 \cap \tau_2.$$

Simplifications are achieved by application of Rules 3.4, 3.5 and:

Rule 4.6 *Let $\tau =$ **if** c **then** τ_1 **else** τ_2 is a subtree of a context quast and τ_1 is the leaf of a context quast, p its context and q the current environment. If $q \wedge p \wedge c \wedge \tau_1$ is unsatisfiable then replace τ with τ_2.*

One context quast is built for each clause of $\hat{\mathsf{P}}$. Without loss of generality, an affine clause of $\hat{\mathsf{P}}$ can be written as:

$$\forall x, r(x) \Longrightarrow s(x) \ll 0,$$

where r is a polyhedron and e an affine function. This implication is equivalent to the inequality:

$$\max\{e(x)\,|s(x)\} \ll 0.$$

Computing the minimum of an objective function defined over a polyhedron is a classical problem of integer programming. PIP [50] for instance produces a quast as result. This quast is the context quast corresponding to the initial clause.

The application of this transformation on all the clauses of $\hat{\mathsf{P}}$ produces a number of context quasts, which, in turn, give a source quast when combined with the quast $\max_{1\leq k\leq m} \max_{0\leq p\leq d_{\mathsf{S}_k \mathtt{R}}} \hat{\varsigma}_k^p(x_k^p, y)$. Let $\tau(x)$ denote this quast. The new expression of the source set is thus:

$$\mathsf{S}(y) = \left\{\tau(\ldots, x_k^p, \ldots)\,\middle|\, x_k^p \in \mathbb{Z}^{m_k}\right\}.$$

4.6.2 Removing parameters

Consider a leaf of the quast $\tau(x)$ defined above, in which some parameters appear. This leaf represents the set of sources obtained by giving all possible values to these parameters. The set of possible values is obtained by "anding" all predicates in the unique path from the root of the quast to the leaf in question.

Rule 4.7 *Let $\tau(x)$ be a leaf parameterized by x governed by n predicates $p_1(x)$, ..., $p_n(x)$ in the unique path from the root to the leaf. Then $\tau(x)$ is transformed into $\{\tau(x)\,|\bigwedge_{i=1}^n p_i(x)\}$.*

After a systematic application of this rule, any leaf in which parameters occur is transformed into a *set* in which the parameters are bound by the predicates governing the leaf. Leaves which do not depend on parameters become singletons.

Now consider the quast: **if** $c(x)$ **then** τ_1 **else** τ_2. Thanks to Rule 4.7, τ_1 and τ_2 are sets of sources. Since the exact value of x is unknown, we cannot predict the outcome of the test. The best we can do is to take the union $\{\tau_1\} \cup \{\tau_2\}$ as an approximation :

Rule 4.8 *A quast* **if** $C(x)$ **then** τ_1 **else** τ_2 *is transformed into* $\{\tau_1\} \cup \{\tau_2\}$.

For each of the examples considered up to now, we recall the clauses of P and compute the set of linear clauses $\hat{\mathsf{P}}$ according to the preceding algorithms. We present the derivations that lead to tautologies only for the first example.

4.6.3 Example E1: `while` loop

For Example E1, the following clauses belong to the set P associated to the dependences of sink $\langle \mathsf{S}_2, y\rangle$:

$$\forall x, x', (x[1] \neq x'[1]) \vee (x[2] > x'[2]) \vee \neg c_1(x') \vee c_1(x), \tag{4.35}$$

$$c_1(y), \tag{4.36}$$

$$\forall x, \neg l_2^0(x, y) \vee \neg c_1(x) \vee x \ll \alpha_2^0(y), \tag{4.37}$$

$$c_1(\alpha_2^0(y)) \vee (\alpha_2^0(y) = -), \tag{4.38}$$

$$l_2^0(\alpha_2^0(y), y) \vee (\alpha_2^0(y) = -), \tag{4.39}$$

$$\forall x, \neg l_2^1(x, y) \vee \neg c_1(x) \vee x \ll \alpha_2^1(y), \tag{4.40}$$

$$c_1(\alpha_2^1(y)) \vee (\alpha_2^1(y) = -), \tag{4.41}$$

$$l_2^1(\alpha_2^1(y), y) \vee (\alpha_2^1(y) = -). \tag{4.42}$$

The first equation accounts for the property of the while, the second for the fact that the read is inside the while, the three next clauses characterize the parameter $\alpha_2^0(y)$ and the three others characterize $\alpha_2^1(y)$.

First, we enumerate the combinations of clauses that produce tautologies: (4.36) combined with (4.37) leads to $\neg l_2^0(y, y) \vee y \ll \alpha_2^0(y)$ is a tautology since $l_2^0(y, y)$ is always false, due to the condition on dependence depth. Same remark for (4.36) with (4.40). (4.37) combined with (4.38) produces a tautology since $\alpha_2^0(y) \ll \alpha_2^0(y)$ is always true. Same remark for (4.40) with (4.41).

Moreover, note that the lazy method saves us the computation of the dependence of depth 0, meaning that this dependence is always killed. Hence $\alpha_2^0(y)$ will not appear in the result, which signifies that any combination whose resulting clause involves $\alpha_2^0(y)$ is not important for the accuracy of the approximate source. We have not considered, up to now, an algorithm that would perform this kind of optimization automatically before the computation of the parametric source quast. The simplification of the source quast during its construction is however likely to eliminate the nodes using $\alpha_2^0(y)$. For simplicity reasons, we will not consider the combinations introducing $\alpha_2^0(y)$.

By combining (4.35) with (4.36), we obtain:

$$\forall x, x[1] \neq y[1] \vee x[2] > y[2] \vee c_1(x).$$

Then, combined with (4.40):

$$\forall x, x[1] \neq y[1] \vee x[2] > y[2] \vee \neg l_2^1(x,y) \vee x \ll \alpha_2^1.$$

$l_2^1(x,y)$ implies $x[1] = y[1] \wedge x[2] \leq y[2]$, therefore the previous clause can be simplified into:

$$\forall x, l_2^1(x,y) \Longrightarrow x \ll \alpha_2^1. \tag{4.43}$$

This linear clause and (4.42) thus belong to $\hat{\mathsf{P}}$.

The context quast associated to (4.42) is:

$$\textbf{if } \alpha_2^1(y) \neq - \textbf{ then } \alpha_2^1(y)[1] = y[1] \wedge \alpha_2^1(y)[2] < y[2] \textbf{ else } -.$$

According to Section 4.6.1, the context quast corresponding to (4.43) is the quast:

$$\textbf{if } 1 < y[2] \textbf{ then } (y[1], y[2]-1) \ll \alpha_2^1(y) \textbf{ else } -_c.$$

Combined together, the two context quasts produce:

$$\textbf{if } 1 < y \textbf{ then } \alpha_2^1(y) = (y[1], y[2]-1) \textbf{ else } -.$$

The exact parametric source is therefore:

$$\sigma(y) = \textbf{ if } 1 < y \textbf{ then } \langle \texttt{S}_2, (y[1], y[2]-1)\rangle \textbf{ else } \langle \texttt{S}_1, y\rangle.$$

As no parameter of the maximum appears in this source, there is no fuzziness at all.

Concerning the source of sink $\langle \texttt{R}, y\rangle$, fuzziness is not reduced by a structural analysis. Hence, the exact parametric source keeps the same expression as in Section 4.3.4. Removing parameters, we obtain the source set:

$$\mathsf{S}(y) = \{\langle \texttt{S}_1, y\rangle\} \cup \left\{\langle \texttt{S}_2, x\rangle \,\middle|\, x \in \mathbb{Z}^2, x[1] = y, x[2] > 1\right\}$$

4.6.4 Example E2: `if..then..else` construct

Consider in Example E2 the dependences of sink $\langle \texttt{R}, y\rangle$ for `s`. The clauses of P are:

$$\forall x, c_1(x) \vee c_2(x), \tag{4.44}$$

$$\forall x, \neg c_1(x) \vee \neg c_2(x), \tag{4.45}$$

$$\forall x, \neg l_1^0(x,y) \vee \neg c_1(x) \vee x \ll \alpha_1^0(y), \tag{4.46}$$

$$c_1(\alpha_1^0(y)) \vee (\alpha_1^0(y) = -), \tag{4.47}$$

$$l_1^0(\alpha_1^0(y), y) \vee (\alpha_1^0(y) = -), \tag{4.48}$$

$$\forall x, \neg l_2^0(x,y) \vee \neg c_2(x) \vee x \ll \alpha_2^0(y), \tag{4.49}$$

$$c_1(\alpha_2^0(y)) \vee (\alpha_2^0(y) = -), \tag{4.50}$$

$$l_2^0(\alpha_2^0(y), y) \vee (\alpha_2^0(y) = -). \tag{4.51}$$

The two first clauses come from the structural analysis, the other ones being characterizations of $\alpha_1^0(y)$ and $\alpha_2^0(y)$.

Combining (4.44) with (4.46) and then with (4.49) produces the linear clause:

$$\forall x, \neg l_1^0(x,y) \vee (x \ll \alpha_1^0) \vee \neg l_2^0(x,y) \vee (x \ll \alpha_2^0).$$

As $l_1^0(x,y) = l_2^0(x,y) = \{x \,|1 \le x \le n\}$, this constraint is equivalent to:

$$\forall x, l_1^0(x,y) \wedge (x > \alpha_1^0) \Longrightarrow (x \le \alpha_2^0). \tag{4.52}$$

Combining now (4.45) with (4.47) and then with (4.50) yields:

$$\forall x, (x \ne \alpha_1^0) \vee (\alpha_1^0 = -) \vee (x \ne \alpha_2^0) \vee (\alpha_2^0 = -). \tag{4.53}$$

$\hat{\mathsf{P}}$ thus contains the clauses (4.52), (4.53), (4.48) and (4.51). The corresponding context quasts are, respectively:

$$\begin{gathered}
\textbf{if } 1 \le n \textbf{ then if } \alpha_1^0 < n \textbf{ then } n \le \alpha_2^0 \textbf{ else } - \textbf{ else } -,\\
\textbf{if } \alpha_1^0 > - \textbf{ then if } \alpha_1^0 < \alpha_2^0 \textbf{ then } - \textbf{ else } \alpha_1^0 > \alpha_2^0 \textbf{ else } -,\\
\textbf{if } \alpha_1^0 > - \textbf{ then } 1 \le \alpha_1^0 \le n \textbf{ else } -,\\
\textbf{if } \alpha_2^0 > - \textbf{ then } 1 \le \alpha_2^0 \le n \textbf{ else } -.
\end{gathered}$$

Taking the maximum of the context quasts, after simplifications in the environment of $\langle \mathtt{R}, [] \rangle$, $1 \le n$:

$$\left| \begin{array}{ll}
\textbf{if } \alpha_1^0 > - & \\
\textbf{then} & \left| \begin{array}{ll}
\textbf{if } \alpha_2^0 > - & \\
\textbf{then} & \left| \begin{array}{l}
\textbf{if } \alpha_1^0 < n \\
\textbf{then } 1 \le \alpha_1^0 < n = \alpha_2^0 \\
\textbf{else } 1 \le \alpha_2^0 < n = \alpha_1^0
\end{array} \right. \\
\textbf{else } 1 \le \alpha_1^0 = n &
\end{array} \right. \\
\textbf{else } 1 \le \alpha_2^0 = n &
\end{array} \right.$$

The context quasts have been combined in the order (4.48), (4.51), (4.52) and (4.53). Finally, in this context, the parametric source becomes:

$$\left| \begin{array}{ll}
\textbf{if } \alpha_1^0 > - & \\
\textbf{then} & \left| \begin{array}{ll}
\textbf{if } \alpha_2^0 > - & \\
\textbf{then} & \left| \begin{array}{l}
\textbf{if } \alpha_1^0 < n \\
\textbf{then } \langle \mathtt{S}_2, n \rangle \\
\textbf{else } \langle \mathtt{S}_1, n \rangle
\end{array} \right. \\
\textbf{else } \langle \mathtt{S}_1, n \rangle &
\end{array} \right. \\
\textbf{else } \langle \mathtt{S}_2, n \rangle &
\end{array} \right.$$

Note that no $-$ appears in this source, which shows that `s` has been correctly initialized, and that the source is either the last instance of $\mathtt{S}_1$ or the last instance of $\mathtt{S}_2$. Removing parameters, we get the fuzzy source:

$$\mathsf{S}([]) = \{\langle \mathtt{S}_1, n \rangle, \langle \mathtt{S}_2, n \rangle\}.$$

4.6.5 Example E3: Non-affine subscript

The clauses of P for Example E3 are:

$$\forall x, x', (x' \neq x-1) \vee (x < 2) \vee \neg c_1(x,y) \vee c_2(x',y), \tag{4.54}$$
$$\forall x, x', (x' \neq x-1) \vee (x < 2) \vee c_1(x,y) \vee \neg c_2(x',y), \tag{4.55}$$
$$\forall x, \neg l_2^0(x,y) \vee \neg c_1(x,y) \vee x \leq \alpha_2^0(y), \tag{4.56}$$
$$c_1(\alpha_2^0(y),y) \vee (\alpha_2^0(y) = -), \tag{4.57}$$
$$l_2^0(\alpha_2^0(y),y) \vee (\alpha_2^0(y) = -), \tag{4.58}$$
$$\forall x, \neg l_4^0(x,y) \vee \neg c_2(x,y) \vee x \leq \alpha_4^0(y), \tag{4.59}$$
$$c_2(\alpha_4^0(y),y) \vee (\alpha_4^0(y) = -), \tag{4.60}$$
$$l_4^0(\alpha_4^0(y),y) \vee (\alpha_4^0(y) = -). \tag{4.61}$$

The first two clauses are obtained by iterative analysis, the other characterize α_0^2 and α_0^4.

Combining (4.54) with (4.57) and then with (4.59) produces:

$$\forall x, x', (x' \neq x-1) \vee (x < 2) \vee (x \neq \alpha_2^0(y)) \vee (\alpha_2^0(y) = -) \vee \neg l_4^0(x',y) \vee (x' \leq \alpha_4^0(y)),$$

that is,

$$(\alpha_2^0(y) \geq 2) \wedge l_4^0(\alpha_2^0(y)-1,y) \Longrightarrow \alpha_2^0(y)-1 \leq \alpha_4^0(y). \tag{4.62}$$

By symmetry, the combination of (4.55), (4.56) and (4.60) leads to:

$$\forall x, x', (x' \neq x-1) \vee (x < 2) \vee (x' \neq \alpha_4^0(y)) \vee (\alpha_4^0(y) = -) \vee \neg l_2^0(x,y) \vee x \leq \alpha_4^0(y),$$

that is,

$$(\alpha_4^0(y) \geq 1) \wedge l_2^0(\alpha_4^0(y)+1,y) \Longrightarrow \alpha_4^0+1 \leq \alpha_2^0(y). \tag{4.63}$$

$l_1^0(x,y) = l_2^0(x,y) = (1 \leq x \leq n)$ and the environment is $(1 \leq y \leq n)$. The context quasts corresponding to (4.62), (4.63), (4.58) and (4.61) are respectively, after simplification in the environment:

$$\textbf{if } 2 \leq \alpha_2^0(y) \textbf{ then if } \alpha_2^0(y) \leq n+1 \textbf{ then } \alpha_2^0(y)-1 \leq \alpha_4^0(y) \textbf{ else } - \textbf{ else } -$$
$$\textbf{if } 1 \leq \alpha_4^0(y) \textbf{ then if } \alpha_4^0(y) \leq n-1 \textbf{ then } \alpha_4^0(y)+1 \leq \alpha_2^0(y) \textbf{ else } - \textbf{ else } -$$
$$\textbf{if } \alpha_2^0 \neq - \textbf{ then } 1 \leq \alpha_2^0(y) \leq n \textbf{ else } -$$
$$\textbf{if } \alpha_4^0 \neq - \textbf{ then } 1 \leq \alpha_4^0(y) \leq n \textbf{ else } -.$$

Combined altogether, the complete context quast is:

$$
\left|\begin{array}{ll}
\multicolumn{2}{l}{\textbf{if } \alpha_2^0(y) \neq -} \\
\textbf{then} & \left|\begin{array}{ll}
\multicolumn{2}{l}{\textbf{if } \alpha_4^0(y) \neq -}\\
\textbf{then} & \left|\begin{array}{ll}
\multicolumn{2}{l}{\textbf{if } 2 \leq \alpha_2^0(y)}\\
\textbf{then} & \left|\begin{array}{l}\textbf{if } \alpha_4^0(y) \leq n-1 \\ \textbf{then } \alpha_2^0(y)-1=\alpha_4^0(y) \\ \textbf{else } \alpha_2^0(y)-1 \leq \alpha_4^0(y) = n\end{array}\right.\\
\multicolumn{2}{l}{\textbf{else } \alpha_4^0(y)=n}
\end{array}\right.\\
\multicolumn{2}{l}{\textbf{else } \alpha_2^0(y)=1}
\end{array}\right.\\
\textbf{else} & \left|\begin{array}{l}\textbf{if } \alpha_4^0(y)\neq - \\ \textbf{then } \alpha_4^0(y)=n \\ \textbf{else } -\end{array}\right.
\end{array}\right.
$$

The expression of the exact source is then:

$$
\sigma(y) = \left|\begin{array}{ll}
\multicolumn{2}{l}{\textbf{if } \alpha_2^0(y) \neq -} \\
\textbf{then} & \left|\begin{array}{ll}
\multicolumn{2}{l}{\textbf{if } \alpha_4^0(y) \neq -}\\
\textbf{then} & \left|\begin{array}{ll}
\multicolumn{2}{l}{\textbf{if } 2 \leq \alpha_2^0(y)}\\
\textbf{then} & \left|\begin{array}{l}\textbf{if } \alpha_4^0(y) \leq n-1 \\ \textbf{then } \left\langle \mathtt{S}_2, \alpha_2^0(y)\right\rangle \\ \textbf{else } \langle \mathtt{S}_4, n\rangle\end{array}\right.\\
\multicolumn{2}{l}{\textbf{else } \langle \mathtt{S}_4, n\rangle}
\end{array}\right.\\
\multicolumn{2}{l}{\textbf{else } \langle \mathtt{S}_2, 1\rangle}
\end{array}\right.\\
\textbf{else} & \left|\begin{array}{l}\textbf{if } \alpha_4^0(y)\neq - \\ \textbf{then } \langle \mathtt{S}_4, n\rangle \\ \textbf{else } -\end{array}\right.
\end{array}\right.
$$

and the source set is:

$$\mathsf{S}(y) = \left\{\left\langle \mathtt{S}_2, x_2^0\right\rangle \middle| 1 \leq x_2^0 \leq n \right\} \cup \{\langle \mathtt{S}_4, n\rangle\}$$

4.6.6 Example E4: Non-affine loop bounds

The set P for the sink $\langle \mathtt{R}, y\rangle$ contains the following clauses:

$$(jup - jmax + 1 \geq 0), \tag{4.64}$$

$$(1 - jlow \geq 0), \tag{4.65}$$

$$(jmax - 1 - y[2] \geq 0), \tag{4.66}$$

$$\forall x, \neg l_3^1(x, y) \vee (jlow - x[2] - 1 \geq 0) \vee (x[2] - jup - 1 \geq 0) \vee x \ll \alpha_3^1(y), \tag{4.67}$$

$$l_3^1(\alpha_3^1(y), y) \vee (\alpha_3^1(y) = -). \tag{4.68}$$

The first two clauses stem from a symbolic analysis, the third clause comes from the context and the others are characterization of $\alpha_3^1(y)$. The variables $jup, jmax$ and $jlow$ are considered as constants since their value do not change in the loop. This is a quick simplification of the problem, as a matter of fact,

the variables depend on the operations which access them and another relations then says that the values taken by jup (resp. $jmax$, $jlow$) are the same for all operations of the loop.

Combining (4.64) with (4.67) gives:

$$\forall x, \neg l_3^1(x,y) \vee (jlow - x[2] - 1 \geq 0) \vee (x[2] - jmax \geq 0) \vee x \ll \alpha_3^1(y),$$

which, in turn, combined with (4.65), provides:

$$\forall x, \neg l_3^1(x,y) \vee (-x[2] \geq 0) \vee (x[2] - jmax \geq 0) \vee x \ll \alpha_3^1(y).$$

Finally, combining this clause with (4.66) yields:

$$\forall x, \neg l_3^1(x,y) \vee (-x[2] \geq 0) \vee (x[2] - y[2] - 1 \geq 0) \vee x \ll \alpha_3^1(y),$$

that is,

$$\forall x, l_3^1(x,y) \wedge (1 \leq x[2] \leq y[2]) \Longrightarrow x \ll \alpha_3^1(y).$$

As $l_3^1(x,y) = (x = y) \wedge (1 \leq x[1] \leq n)$, the context quast associated to this clause is:

$$y \ll \alpha_3^1(y).$$

The context quast associated to (4.68) is:

$$\textbf{if } \alpha_3^1(y) \neq - \textbf{ then } \alpha_3^1(y) = y \textbf{ else } -.$$

The combination of the two context quasts gives, after simplification:

$$\alpha_3^1(y) = y.$$

Hence, the source of $\langle \texttt{R}, y\rangle$ is exact and equal to $\langle \texttt{S}_3, y\rangle$.

4.7 Implementation

We have partially implemented the methods described in this chapter. Our prototype, Caravan, uses the formalism of a previous version of FADA. Moreover, we only integrate the properties found by a structural analysis on the `if..then..else`.

4.7.1 Overview of Caravan

Caravan is a prototype programmed in Objective Caml [96], which performs approximate array dataflow analysis on the subset of Fortran programs described in 4.1. The prototype has around 15000 lines of Caml code and is made up of different modules. The most important ones are, so far:

Parser A subset of Fortran 77 is recognized by our parser. The program model of our approximate analysis is included inside this subset. This parser is

made of two parts. The first one, written in C, parses a subset of Fortran 77 and of C and produces a common intermediate representation for both languages [44]. This representation is then parsed by the second part, written in Caml. The first part of the parser is a improved version of the parser used by PAF.

Formal calculus module This module contains basic tools to manipulate quasi-affine systems of inequalities and an interface with PIP so as to compute maxima, minima or testing the emptiness of polyhedra. This part has been written by P. Boulet.

Dependence testing module Dependence testing is considered as a prepass to array dataflow analysis. We use the PIP test.

Structural analysis Structural properties are found with this module.

Translation module This module translates the properties on non-affine constraints into properties on parameters of the maximum. We detail in the following the way it works.

FADA module This module implements the computation of exact parametric sources w.r.t. parameter domains. It is assumed that the set of clauses $\hat{\mathsf{P}}$ on parameter domains is given as a context quast. This module does not translate P into $\hat{\mathsf{P}}$.

We use the framework described in [12] for the approximate analysis. Therefore, the properties on non-affine constraints are written as relationships between parameter domains. So far, only the relations coming from the characterization of the parameters and from the structural analysis are taken into account. Moreover, for the structural analysis, the properties of the `while` loops are not expressed.

The different modules which are always called during an analysis are: the parser, dependence testing, FADA modules, in that order. If the usage of the structural analysis module is required by the user, then the translation module is also called.

4.7.2 Translation module

We present in this section a more detailed description of the way the properties on parameter domains are handled and of their translation into properties on parameters of the maximum.

In the version of FADA which is implemented, relations on non-affine constraints can only be relations between parameter domains. We represent parameter domains by their non-affine constraints. Intersections, unions of sets, images of sets by affine functions and relations between sets are built with type

constructors. Any relation of inclusion between two sets can be written as a system of inclusion relations:

$$\bigcap_i \mathsf{E}_i \subseteq \bigcup_j \mathsf{F}_j$$

where E_i and F_j are either polyhedra or parameter domains or image of parameter domains by an affine function. Any relation on parameter domains is first normalized into a system of inclusion relations of this kind.

Then, each inclusion relation is translated into properties on parameters of the maximum thanks to the rules given in [12]. These properties are implications, and are encoded as a context quast. A simplification of the combined context quasts is performed after each combination. The result is then used by the FADA module.

4.7.3 Example

Our prototype produces for Example E1 the text of the program, with the results of the analysis in the comments. Each statement is annotated by a statement number and the sources of the variables it reads. In the following program, we only considered the sources of `s`.

```
C STATEMENT NUMBER: 1
        do i = 1,n
C STATEMENT NUMBER: 2
         s = 10
C STATEMENT NUMBER: 3
         do while (z.ge.0)
C STATEMENT NUMBER: 4
C SOURCE OF s:
C  Parameters :
C  Quast :
C  if _alpha11 >= i then
C    if i >= _alpha11 then
C      if _alpha12 >= 1 then
C        if _while0-1-_alpha12 >= 0 then
C          [i; _alpha12], Statement4
C        else [i], Statement2
C      else [i], Statement2
C    else [i], Statement2
C  else [i], Statement2
          s = 2*s
          z = z-1
         enddo
C STATEMENT NUMBER: 5
```

```
C SOURCE OF s:
C  Parameters :
C  Quast :
C  if _alpha7 >= i then
C    if i >= _alpha7 then
C      if _alpha8 >= 1 then
C        [i; _alpha8], Statement4
C      else [i], Statement2
C    else [i], Statement2
C  else [i], Statement2
         z = s
        enddo
       end
```

Fuzzy parameters are called `_alpha`n, and an operation $\langle \mathtt{n}, x\rangle$ is written as `[x], Statementn`. The variable `_while0` denotes the loop counter of the while.

The sources we get correspond to the exact parametric sources described in Section 4.3.4 page 110.

4.8 Related Work

Compile-time flow-sensitive analyses can be categorized into one of the two approaches:

- The analysis computes approximate flow dependences between operations accessing the same array elements, and operations are explicitly manipulated. This fine-grain kind of analysis is the approach taken in this chapter, following the method described by Collard [29], as well as the approach taken by Duesterwald et al. [46] and by Pugh and Wonnacott [129]. These analyses are conservative, thus the exact dependence between two operations implies the existence of the approximate dependence. Hence the approximation consists in computing a set of possible dependences instead of only one dependence. With the exception of the work of Duesterwald et al. which is an extension of the classical scalar dataflow framework to arrays structures, all other works are extensions of the pairwise methods [23, 52].

- The analysis operates on summaries of the array elements from which the flow between blocks of operations can be deduced between blocks of operations. Note that unlike the previous techniques, no approximate DFG is explicitly computed. This approach includes most of the extensions of the classical scalar dataflow frameworks to array structures. Methods based on reaching sets [64, 114, 59] still conserve the statements which last defined array elements, which is not the case of the methods comput-

ing only *array regions* [120, 34, 35, 68, 69, 122, 65] , that is, summaries of the array elements used or defined in a code fragment.

In general, the second approach has a wider domain of application than the first one but some of the methods that expose fine-grain parallelism necessitate the results of the first approach (such as scheduling).

4.8.1 Analyzing individual array references

Let us examine the three methods that propose to approximate the DFG by a set of possible dependences between operations accessing the same individual array elements:

Duesterwald, Gupta and Soffa [46]: Their method extends *reaching definition* analysis [1] for scalars to array structures. The computation of the reaching definitions for scalars is a commonly used technique to determine the flow of data by means of dataflow equations. Each statement is annotated with the last defining statement for each variable. Although operations are not explicitly manipulated, the reaching definitions define the flow at the operation level through the concept of the "last defining statement". The method proposed in [46] computes an over- and under-approximation of the dependence distance between any read operation accessing an array element and the last defining statement accessing the same element. The approximation operates on a lattice of integer intervals, not on a polyhedron, hence the approximation can be very rough in some cases. Moreover, they do not handle conditional dependences, array subscripts must be affine w.r.t. the innermost loop index and their method targets mostly monodimensional arrays.

Pugh and Wonnacott [105, 106, 129]: They extended their framework of dependence relations, presented in Section 3.4, so as to cope with non-affine functions w.r.t. loop indices. The approach presented is very similar to ours. In their first work [105], over- and under-approximations of dependence relations were computed by either setting non-affine terms to true or to false. They also proposed some more precise bounds, when dealing with equalities between non-affine functions. In their more recent works [106, 129], they use SSA form to find relations between non-affine functions and could use any of the methods presented in Section 4.4. Whether the accuracy of the dependence relations depends only on the amount of information gathered on non-affine constraints is however uncertain. We compare in greater details their method and ours in Section 4.8.3.

Collard and previous versions of FADA [29, 30, 12]: Collard [29] first introduced the term of FADA to name an extension of the analysis proposed by Feautrier to dynamic control programs. The first version of FADA extended the static control program model to `while` loops and

`if..then` constructs. Since then, we have proposed in [30, 12] and in this chapter several improvements of the original method, changing the formalism so as to take into account a larger class of non-affine constraints, arising for example from array subscripts, and a larger class of properties on non-affine constraints. In the same time, we have also found the conditions for which no information is lost during this process of integration, as far as the source is concerned. We present thereafter the limitations of the previous versions of FADA.

In [29], what is called *hidden variables* represents the non-affine constraints and the parameters of the maximum are not defined. Non-affine subscripts are not handled and the structural property of the `while` loops is the only property on non-affine constraints that can be treated. Hence, this first FADA produced for Example E1 described in Section 4.2.2 the same source set as the one we computed in this chapter, but produced very fuzzy results for the other examples.

In [30], the formalism is the same as in the previous paper, and the FADA is able to take advantage of the fact that the two branches of the conditionals cannot be executed in the same time. Yet the source set for Example E2 is still more fuzzy than what we obtained in Section 4.6.3 since the analysis do not use the fact that at least one of the branches is executed when the `if..then..else` is executed.

In [12], we introduced the notion of parameters of the maximum and therefore were able to present an algorithm that translates properties on non-affine constraints into properties on parameters of the maximum so as to integrate them in the computation of the source. The formalism is the same as the one described in this chapter, but the properties on non-affine constraints are restricted to properties on parameter domains. All properties resulting from a structural analysis are translated without adding fuzziness. Therefore the behavior of `if..then..else` constructs can be modeled correctly. However, general relations taking the form of Presburger formulae such as the ones provided by symbolic analysis cannot be used. The analysis of the source of array `A` in Example E4 is therefore very fuzzy. We proved that any property on parameter domains can be taken into account in this framework without adding fuzziness and obtained the same accuracy for the analyses of Examples E1, E2 and E3. The integration of the properties in the computation of the source set is performed by an *ad hoc* method, less general than the resolution method presented in this thesis and specifically designed to handle only relations between parameter domains.

4.8.2 Analyses using summaries

These methods make coarser approximations and can abstract dependences as dependence levels. This is sufficient for privatization. We first detail the techniques computing reaching sets, and then array region analyses.

Several extensions to the reaching definition method have been proposed for array structures [64, 114, 59], which only specify the last defining statement and do not keep track of the precise source operation, as in the original method for scalars. For each array structure and for each statement, sets of array elements are associated to the last statements defining them. The reaching information is summarized after each loop. However, when several statements define the same array elements, the conditions for which one definition or the other is the reaching definition are not in general determined.

```
   do i = 1, 15
S1  if .. then A(i) = ..
   enddo
   do i = 10, 20
S2  if .. then A(i) = ..
   enddo
```

Figure 4.4: Reaching definitions and conditional definitions

For instance, for the example of Figure 4.4, statement $\mathtt{S}_1$ conditionally defines $\mathtt{A}(1:15)$ and statement $\mathtt{S}_2$ conditionally defines $\mathtt{A}(10:20)$. In the framework defined in [59], the sets of data reaching a statement are over- and under-approximated: the representation of the conditional reaching definition after the two statements is: $(\mathtt{A}, (\{1:9\}, \{10,15\}, \{16,20\}), (\{\mathtt{S}_1\}, \{\mathtt{S}_1, \mathtt{S}_2\}, \{\mathtt{S}_2\}))$[3]. Its means that for all i s.t. $1 \leq i \leq 9$, $\mathtt{A}(i)$ may be defined by a previous execution of statement $\mathtt{S}_1$, for all $i, 10 \leq i \leq 15$, $\mathtt{A}(i)$ may be defined by a previous execution of $\mathtt{S}_1$ or $\mathtt{S}_2$ and the elements of index between 16 and 20 may be defined by $\mathtt{S}_2$. The representation of the unconditional reaching definition is $(\mathtt{A}, (), ())$, meaning that no array element is defined with certainty.

As for array regions, their purpose was first to keep track of the effects of procedure calls on array elements [120, 121, 24]. Array regions accessed by an individual array reference are locally approximated by some area descriptor. Then, the summarized information locally determined is propagated along the control flow graph. The complexity of the computation depends on the choice of the area descriptor and on the complexity of the meet operator, summarizing the effects of several control flows on array structures. Several kinds of regions can be computed for structured pieces of code [69, 122, 34, 65] and we describe four of them, using the terminology of Creusillet [34, 35]: (S represents a possibly

[3] As a matter of fact, the authors choose not to represent the statements in conditional reaching sets, for efficiency reasons.

compound statement)

- WRITE(S): the set of array elements defined by S.
- READ(S): the set of array elements read by S.
- OUT(S): the set of array elements that are live after S and will be used. Some variants define only the region of the elements live after S.
- IN(S): also called upward exposed reads, this is the set of elements that are read by S and are not (yet) defined by S. Intuitively, IN(S) is the set of elements that are imported by S.

In fact, only over- and under-approximations of these regions are computed (called respectively May and Must regions). By intersecting IN and WRITE regions of different statements, one can find whether the source of some array elements comes from the statement of the WRITE region. Array privatization techniques based on array region analyses rely on this fact. Copy-out regions can then be found thanks to the OUT regions.

There are many representations of array regions. Regions only describe parts of the data space, thus the descriptor used do not need, in general, to be affine w.r.t. the iteration counters. There exists one exception: A representation, called *Atom Image* [84, 71], takes into account the exact effects of statements. But the summary consists in the list of the regions of each statement and the exactness is only ensured if some heavy linearity constraints similar to those of static control programs are imposed. More compact representations do not have these restraints. However, some representations introduce inaccuracy because they are not closed by union and intersection (such is the case for convex sets or bounded $\mathbb{Z}$-modules) or do not allow the use of control information, such as `if` conditions.

4.8.3 Detailed comparison with Wonnacott's work

Pugh and Wonnacott [107, 129] have extended their framework of dependence relations to more general control flow and non-linear array subscripts. As a matter of fact, their techniques apply on programs verifying the same conditions as ours, described in Section 4.1. We first present how they propose to handle non-affine constraints and then compare their method with ours. We will in particular highlight the differences that come from the representation, and the differences that arise from the method.

Non-affine constraints in dependence relations

They proposed three degrees of approximation, from a very fuzzy analysis to an analysis able to take into account some symbolic information. They use the same definition of non-affine constraints as in our framework.

The first idea is to ignore the non-affine constraints. This mode is called "panic mode", and as we have already noticed in Section 4.2.3, this is not in general a conservative approximation, since some dependences may be forgotten during the computation. They suggest to tag the clauses that have been approximated in this way so as to determine when the approximation is conservative. In fact, this approximation is safe when non-affine constraints are removed from a formula that is negated.

The second approximation consists in replacing any non-affine constraint by the term *unknown*. Therefore, the formula $1 \leq i \wedge i \leq f(..)$ becomes $1 \leq i \wedge \mathit{unknown}$. Pugh and Wonnacott define the behavior of *unknown* w.r.t. the boolean operations so that this approximation is always conservative. Basically, this approach is equivalent to our maximal fuzziness, when no property is found on the parameters of the maximum.

The third method is the most interesting: One non-linear expression e is represented in this framework by the expression $f_e(x)$, where f_e is an *uninterpreted function symbol* and x is an iteration vector. As in our formalism, these functions describe non-linear predicates of a conditional or of a `while` loop, non-linear `do` loop bounds or non-linear array subscripts. All functions are supposed to have integer values, so as to be handled by Presburger arithmetic.

```
do i=1, n S A(m*i) = .. R .. = A(i) enddo
```

In this example for instance, the flow (memory-based) dependence relation between S and R is:

$$\mathsf{M}_{\mathtt{S}\to\mathtt{R}} = \left\{ i' \to i \,\middle|\, 1 \leq i' \leq i \leq n \wedge f_{m*i}(i') = i \right\}.$$

Pugh and Wonnacott make the following restriction on the usage of non-linear functions: They are allowed in the expression of a relation if they depend only on constants and on the iteration vectors of the pair of operations in relation. (In the above example, this is the case since i is the iteration vector of the write.) Note that when non-linear functions only depend on the iteration vector of the read, this corresponds to our *exact case* 4.5.5 described page 131. When they only depend on the iteration vector of the write, this corresponds to another exact case. This is one of the advantages of representing relations instead of functions as in our framework.

Value-based dependence relations represent the pairs of write-read operations that are in direct dependence (see Section 3.4.1 for a detailed description). These relations are deducted from memory-based dependence relations by composition and subtraction. For example, the value-based dependence from statement $\mathtt{S}_i$ to R, when $\mathtt{S}_j$, for all j, are write statements of possible sources, is

$$\mathsf{V}_{\mathtt{S}_i\to\mathtt{R}} = \mathsf{M}_{\mathtt{S}_i\to\mathtt{R}} - \bigcup_j \mathsf{M}_{\mathtt{S}_i\to\mathtt{S}_j} \circ \mathsf{M}_{\mathtt{S}_j\to\mathtt{R}}. \tag{4.69}$$

During the computation of a value-based dependence, some functions may appear with arguments different from the iteration vectors of the operations in relation, such as quantified variables. These arguments are deemed to be eliminated from the final result. Since Presburger arithmetic with uninterpreted function symbols is undecidable [45], an approximation is necessary. If the arguments of a function f_e appearing in the expression of a value-based dependence are not completely determined by the value of the iteration vectors of the pair of read-write operations, then:

- $f_e(x)$ is replaced by an existentially quantified variable $\alpha_{f_e(x)}$ in all inequalities in which it appears. The purpose of $\alpha_{f_e(x)}$ is to capture some cases where there is no solution to the initial system of inequalities.
- The term *unknown* is added to the definition of the relation, to show that an approximation has been done. Note that this approximation cannot be refined in any way afterwards.

Note that this approximation takes place in every term $\mathsf{M}_{\mathtt{S}_i \to \mathtt{S}_j} \circ \mathsf{M}_{\mathtt{S}_j \to \mathtt{R}}$ which must be subtracted to $\mathsf{M}_{\mathtt{S}_i \to \mathtt{R}}$.

Pugh and Wonnacott propose several methods to find relations between non-affine functions so as to improve the accuracy of the dependence relations:

- Ask the programmer to confirm a relation that would disprove a dependence. It is indeed easy to retrieve the expression of the real non-linear constraints from the uninterpreted function symbols.
- Use an SSA analysis to prove the equality between two functions used in different points of the program
- Use another symbolic analysis. They suggest to use the output of the analysis of Tu and Padua [124] in order to compute the sources in Example E4.

Basically, any relation on non-affine constraints that is written as a Presburger formula can be used by their analysis. Hence FADA and their method can resort to the same symbolic analyses so as to improve their output.

Comparing with our framework

The complementary analyses that find relations on non-affine constraints can be used in both frameworks, but how this information is used and the impact it has on the accuracy of the resulting dependences seems to differ between dependence relations and FADA.

When a non-linear constraint appears in the computation of a value-based dependence and when the conditions for an exact case are not satisfied, then a rough approximation is automatically performed, introducing "*unknown*" which do not allow any further refinement. This entails two consequences:

Dependence relations cannot be computed entirely in a parametric way with uninterpreted function symbols, by opposition to the computation of the source function with parameters of the maximum. Moreover, the accuracy of the result does not depend only on the relations that can be found between non-affine constraints.

Let us study this last difference. A relation between non-affine constraints can be taken into account if none of the uninterpreted function symbols has been eliminated yet and replaced by *unknown*. Therefore, in the framework described in [129], the relations that can really improve the accuracy of the result must involve non-affine functions that:

- Either appear in the same memory-based dependence relation.
- Or depend on the same iteration vector.

This seems to be a major drawback since even structural properties cannot sometimes be taken into account. Such is the case of Example E2, since both write statements are governed by a non-affine predicate in a loop that is not shared with the read. In order to determine if more distant writes than $\mathtt{S}_1$ and $\mathtt{S}_2$ could define the value read in $\langle \mathtt{R}, [] \rangle$, they compute the partial cover of these two writes. The partial cover represents the set of iteration vectors of the read which corresponds to sinks of yet unknown source (upward exposed sinks). For Example E2, the iteration domain of R has only one element, the vector [], and the partial cover of R with respect to $\mathsf{M}_{\mathtt{S}_1 \to \mathtt{R}}$ and $\mathsf{M}_{\mathtt{S}_2 \to \mathtt{R}}$ is:

$$\{[]\,|n \geq 1\} - \mathsf{RANGE}(\mathsf{M}_{\mathtt{S}_1 \to \mathtt{R}}) - \mathsf{RANGE}(\mathsf{M}_{\mathtt{S}_1 \to \mathtt{R}}),$$

where

$$\mathsf{RANGE}(\mathsf{M}_{\mathtt{S}_1 \to \mathtt{R}}) = \{[]\,|\exists x, 1 \leq x \leq n \wedge f_P(x) \geq 0\}\,,$$

and

$$\mathsf{RANGE}(\mathsf{M}_{\mathtt{S}_2 \to \mathtt{R}}) = \{[]\,|\exists x, 1 \leq x \leq n \wedge f_P(x) < 0)\}\,.$$

Note that the original predicates have been replaced by inequalities with uninterpreted function symbols. The partial cover can be written as:

$$\{[]\,|(n \geq 1) \wedge \neg(\exists x, 1 \leq x \leq n \wedge f_P(x) \geq 0) \wedge \neg(\exists x, 1 \leq x \leq n \wedge f_P(x) < 0)\}\,.$$

In the formula coming from $\mathtt{S}_1$, f_P depends on an existentially quantified variable that must be eliminated, hence $\exists x, 1 \leq x \leq n \wedge f_P(x) \geq 0$ is transformed into $\exists x, \exists \alpha_{f_P(x)}, 1 \leq x \leq n \wedge \alpha_{f_P(x)} \geq 0 \wedge \mathit{unknown}$. Likewise, $\exists x, 1 \leq x \leq n \wedge f_P(x) < 0$ is transformed into $\exists x, \exists \alpha_{f_P(x)}, 1 \leq x \leq n \wedge \alpha_{f_P(x)} < 0 \wedge \mathit{unknown}$. Finally, the set obtained is:

$$\{[]\,|\mathit{unknown}\}\,.$$

Thus this set is not empty, there is still a possibility that a statement preceding $\mathtt{S}_1$ and $\mathtt{S}_2$ is the source of s. This fuzziness can be avoided in our framework by using a simple structural analysis.

Wonnacott[4] proposes to change the way structural properties of conditionals are handled. For Example E2, instead of subtracting the ranges of $\mathsf{M}_{\mathtt{S}_1 \to \mathtt{R}}$ and $\mathsf{M}_{\mathtt{S}_2 \to \mathtt{R}}$ independently of each other, the union of the ranges is first computed, and then this new set is subtracted to $\{[]\,|n \geq 1\}$. The union of the two ranges, $\mathsf{RANGE}(\mathsf{M}_{\mathtt{S}_1 \to \mathtt{R}}) \cup \mathsf{RANGE}(\mathsf{M}_{\mathtt{S}_2 \to \mathtt{R}})$, is

$$\{[]\,|\exists x_1, x_2,\ \ (1 \leq x_1 \leq n \wedge f_P(x_1) \geq 0) \vee (1 \leq x_2 \leq n \wedge f_P(x_2) < 0)\ \ \},$$

that is,

$$\left\{ [] \left| \exists x_1, x_2, \begin{array}{l} (x_1 = x_2 \wedge 1 \leq x_1 \leq n) \\ \vee \left(x_1 \neq x_2 \wedge \left(\begin{array}{l} (1 \leq x_1 \leq n \wedge f_P(x_1) \geq 0) \\ \vee (1 \leq x_2 \leq n \wedge f_P(x_2) < 0) \end{array} \right) \right) \end{array} \right. \right\}.$$

This time, the term $x_1 = x_2 \wedge 1 \leq x_1 \leq n$ ensures that the partial cover is empty. As a matter of fact, it can be easily seen that the technique consisting in eliminating $f_P(x_1) \geq 0$ and $f_P(x_2) < 0$ when $x_1 = x_2$ is the application of a resolution rule. So the method of resolution we presented in Section 4.5 to eliminate non-affine constraints of P is also valid in the framework of dependence relations, provided that we make the following modifications to the initial algorithm computing value-based dependences (and partial covers):

- Consider the set $\bigcup_j \mathsf{M}_{\mathtt{S}_i \to \mathtt{S}_j} \circ \mathsf{M}_{\mathtt{S}_j \to \mathtt{R}}$ in the expression (4.69) and the set of clauses P defining relations on non-affine constraints. Apply a resolution methods on the clauses defining these two sets so as to eliminate non-affine functions.

- Compute the difference and remove any remaining non-affine function from the set by the techniques described in [129].

The condition of Theorem 4.1 on the fuzziness of the result applies on dependence relations and the original limited method of Pugh and Wonnacott can be considered as a trade-off between efficiency and accuracy.

Finally, setting aside the differences that are due to the representation, the fundamental difference between the method based on dependence relations and FADA is that dependence relations cannot be computed entirely in a parametric way with uninterpreted function symbols, whereas sources can be expressed exactly w.r.t. the parameters of the maximum. Exact but parametric sources can then be used by some applications, such as scheduling, and the approximation can take place after this application so as to obtain better results.

4.8.4 Detailed comparison with Creusillet's work

Fuzzy array dataflow analysis is more complex than some analyses that do not compute explicitly a DFG, such as array regions. Is this difference in

[4] Personal communication

complexity justified by a difference in the degree of parallelism that can be exposed? Consider for instance array privatization. For static control programs, we have seen that FADA boils down to an exact array dataflow analysis. Thus the exact DFG is found and our analysis allows privatization techniques to expose at least as much parallelism as can be exposed from an array region analysis, which is in general approximate. For non-static control programs in the scope of FADA, the two approaches are approximate. It is no longer clear that FADA outperforms a region analysis in terms of exposed parallelism. We will show that this is still true, and that this is due to the fact that FADA is able to take advantage of any symbolic information about non-affine constraints.

We will make the comparison of FADA with the framework presented by Creusillet [34], as an example of an array region framework. She clearly separates the limitations induced by the choice of a particular region representation from the approximation problems that must be faced by any representation. An array element can be privatized in a loop only if none of its possible sources for an iteration comes from a previous iteration of the loop. Hence, we will compare the accuracy with which both formalisms describe WRITE and IN regions, or loop-carried flow dependences. In order to make the comparison easier, we will introduce a formalism from which array regions and dataflow dependences can be derived. Note that this formalism, called *extended array regions* is introduced only for comparison purposes and is not considered as a new way of computing dependences or array regions. Moreover, we will resort to a dependence relation representation of direct dependences instead of a source representation. As shown in the previous section, we can still use resolution rules so as to integrate non-affine relations in the analysis.

In the first part of this comparison, a semantic analysis framework that allows the construction of array regions and dependence relations is exposed. It is largely inspired by the formalism introduced by Creusillet and the reader is referred to [34] for a more precise description of usual semantic analysis frameworks. Then we present the preliminary analyses required by the computation of the extended array regions. Finally, we give the rules to build these regions and compare the accuracy of FADA with array regions through this formalism.

Semantical domains and functions

The idea is to associate to each statement (or block of statements) of the program and to each possible execution of this statement, a set of array elements and the operations that have accessed these elements, for each array structure. This idea is not new, Li and Yew [84] and then Hind et al. [71] have already proposed this kind of representation for regions, but their framework is limited to affine constraints with respect to loop indices. So as to cope with non-affine constraints, we will describe the sets of array elements with first order arithmetic predicates, with any function symbol. Our aim is slightly different from the goal of Li and Yew, since we only want to build a representation from which

the expression of WRITE and IN exact array regions and exact dependence relations can be derived. We will not compute the extended array regions.

The sets of array elements and operations from which we deduce array regions and dataflow dependences are the result of a semantic analysis. We restrict this analysis to the language $\mathcal{L}$ described by the grammar of Figure 4.8.4. `iter` represents a loop counter, `id` a non-loop index variable and `f`

$$\begin{array}{lcl} P & \rightarrow & B \\ S & \rightarrow & \mathtt{id}(E_1) = E_2 \\ & & \mid \mathtt{do}\ x = E_1,\ E_2\ \mathtt{step}\ E3;\ B_1;\ \mathtt{enddo} \\ & & \mid \mathtt{do}\ x = 1\ \mathtt{while}\ E;\ B_1;\ \mathtt{endwhile} \\ & & \mid \mathtt{if}\ E\ \mathtt{then}\ B_1\ \mathtt{else}\ B_2 \\ B & \rightarrow & B_1;\ S \\ & & \mid\ ; \\ E & \rightarrow & \mathtt{f}(\mathtt{id}_1(E_1), \ldots, \mathtt{id}_n(E_n)) \\ & & \mid \mathtt{iter} \end{array}$$

Figure 4.5: Grammar of $\mathcal{L}$

stands for a function (affine or not). Traditionally, semantic functions associate to statements (or blocks of statements) and to the current state of the memory at an execution point, an element of a set. The current state of the memory at an execution point is completely determined by the current operation. Moreover, we only need, for our semantic functions, to represent the current operation by its iteration vector. Hence, all semantic functions we present in the following share a common type:

$$\begin{array}{rcl} \mathsf{R} : \mathcal{L} & \rightarrow & (\mathbb{Z}^p \longrightarrow \mathsf{E}) \\ l & \rightarrow & \mathsf{R}[\![l]\!] \end{array}$$

with E a set, called semantical set, and p the number of loops surrounding the considered statement. Note that the value of p may vary from a statement to another. This apparent problem in the definition of R can be addressed by considering that p is set to the maximal number of nested loops in the program. We will only consider sets E that are complete lattices. The structure of lattice is then extended to the semantic functions; If $\subseteq_{\mathsf{E}}$, $\cup_{\mathsf{E}}$ and $\cap_{\mathsf{E}}$ are respectively the partial order, least upper and lower bounds operators of E, then $\subseteq_{\mathsf{R}}$, $\cup_{\mathsf{R}}$ and $\cap_{\mathsf{R}}$ are defined as:

$$\begin{array}{rcl} \mathsf{R}_1[\![l_1]\!] \subseteq_{\mathsf{R}} \mathsf{R}_2[\![l_1]\!] & \Longleftrightarrow & \forall x \in \mathbb{Z}^p,\ \mathsf{R}_1[\![l_1]\!](x) \subseteq_{\mathsf{E}} \mathsf{R}_2[\![l_2]\!](x), \\ \forall x \in \mathbb{Z}^p,\ (\mathsf{R}_1[\![l_1]\!] \cup_{\mathsf{R}} \mathsf{R}_2[\![l_2]\!])(x) & = & \mathsf{R}_1[\![l_1]\!](x) \cup_{\mathsf{E}} \mathsf{R}_2[\![l_2]\!](x), \\ \forall x \in \mathbb{Z}^p,\ (\mathsf{R}_1[\![l_1]\!] \cup_{\mathsf{R}} \mathsf{R}_2[\![l_2]\!])(x) & = & \mathsf{R}_1[\![l_1]\!](x) \cup_{\mathsf{E}} \mathsf{R}_2[\![l_2]\!](x). \end{array}$$

We use two types of semantic functions: the first one transforms an iteration vector into a set of integer vectors (E is $\wp(\mathbb{Z}^m)$ for a value of m). The second kind of semantic functions associates to each statement and iteration vector a set of array elements and operations, for each array structure. Let Ω denote the set of operations and $(\mathtt{A}_1, \ldots, \mathtt{A}_m)$ the list of array structures in the program. The semantic domain is then:

$$\mathsf{E} = \prod_{i=1}^{m} \wp\Big(\mathbb{Z}^{d_i} \times \Omega\Big).$$

The ith set in this Cartesian product is associated to the array $\mathtt{A}_i$ of dimension d_i. For each array structure, we build a set formed by pairs of array elements and operations.

For the first kind of semantic functions, $\wp(\mathbb{Z}^m)$ is the usual lattice, ordered by the inclusion relation. For the second one, we define for all i an order $\sqsubseteq$ on $\wp\big(\mathbb{Z}^{d_i} \times \Omega\big)$, built from the orders $\subseteq$ and $\preceq$. For all sets $\mathsf{A}_1 = \{x, \tau \,|r_1(x, \tau)\}$ and $\mathsf{A}_2 = \{x, \tau \,|r_2(x, \tau)\}$ of $\wp\big(\mathbb{Z}^{d_i} \times \Omega\big)$, with r_1 and r_2 two predicates,

$$\begin{aligned} \mathsf{A}_1 \sqsubseteq \mathsf{A}_2 \iff & (\{x \,|\exists \tau, (x, \tau) \in \mathsf{A}_1\} \subseteq \{x \,|\exists \tau, (x, \tau) \in \mathsf{A}_2\}) \\ & \wedge ((x, \tau_1) \in \mathsf{A}_1 \wedge (x, \tau_2) \in \mathsf{A}_2 \Longrightarrow \tau_1 \preceq \tau_2). \end{aligned}$$

Intuitively, a set is smaller than the other if the array elements of the first belong to the second one and any operation associated to an array element of the first set is executed before all operations associated to the same element of the second set. One can easily verify that this order defines a complete lattice. The corresponding operators $\sqcup$ (union), $\sqcap$ (intersection) and $\boxminus$ (minus) are defined by:

$$\begin{aligned} \{x, \tau \,|r_1(x, \tau)\} \sqcup \{x, \tau \,|r_2(x, \tau)\} &= \left\{ x, \tau \,\middle|\, \exists \tau_1, \exists \tau_2, \begin{array}{c} \left(\begin{array}{c} (\tau_1 \neq - \wedge r_1(x, \tau_1)) \\ \vee(\tau_2 \neq - \wedge r_2(x, \tau_2)) \end{array} \right) \\ \wedge \tau = \max(\tau_1, \tau_2) \end{array} \right\}, \\ \{x, \tau \,|r_1(x, \tau)\} \sqcap \{x, \tau \,|r_2(x, \tau)\} &= \left\{ x, \tau \,\middle|\, \exists \tau_1, \exists \tau_2, \begin{array}{c} r_1(x, \tau_1) \wedge r_2(x, \tau_2) \\ \wedge \tau = \min(\tau_1, \tau_2) \end{array} \right\}, \\ \{x, \tau \,|r_1(x, \tau)\} \boxminus \{x, \tau \,|r_2(x, \tau)\} &= \{x, \tau \,|r_1(x, \tau) \wedge \neg(\exists \zeta \text{ s.t. } r_2(x, \zeta) \wedge \tau \preceq \zeta)\}. \end{aligned}$$

Intuitively, the union of two sets is the set of all array elements belonging to one of the sets, associated to the latest operations that are associated to these elements. The subtraction of two sets is the set of array elements that do not appear in the subtracted set with a more recent operation. Finally, using Church's λ-calculus [8], we define the following operator, denoted $\max_{\preceq_q}$:

$$\begin{aligned} & \max_{\preceq_q}(\lambda x'. \{z, \langle \mathsf{S}, x\rangle \,|\, r(x, x', z)\}) = \\ & \lambda x'. \{z, \langle \mathsf{S}, x\rangle \,|\, r(x, x', z) \wedge \neg(\exists x'' \text{ s.t. } x[p] < x''[p] \wedge r(x'', x', z))\}. \end{aligned}$$

Preliminary analyses

Some preliminary analyses have to be performed in order to achieve the computation of our extended regions. Many program expressions are needed in the definition of these regions, such as loop bounds, array subscripts or conditionals. The symbolic evaluation of the syntactic expressions w.r.t. loop counters is performed by a semantic function, denoted V. The value of an expression E, given the iteration vector x, is written as $\mathsf{V}[\![E]\!]x$.

Moreover, we define some other functions as grammar attributes. If S represents a possibly compound statement, $S.p$ is the depth of S, $S.l$ is the label associated to S. $S.iter(x)$ is the monodimentional iteration domain of the innermost loop surrounding S (`do` or `while` loop), with respect to the iteration indices of outer loops.

The collection of properties of non-affine constraints is another preliminary analysis. In the framework of Creusillet, there are three kinds of properties, formalized as semantic functions:

Preconditions : They associate to statements the set of predicates governing the execution of this statement.

Transformers : Transformers are semantic functions that abstract the transformation on the values of variables, resulting from the execution of a statement. Reverse transformers are also considered, and they abstract the inverse effects.

Continuation conditions : They are the conditions for which the program does not stop after the execution of a statement. We will not handle continuation conditions, since our program model does not include the instruction `stop`.

All these functions find affine properties on the variables, approximating the real properties of the program. These properties are relations on the variables of the program, and the scope of the algorithm used to build them is similar to the domain of the method described by Cousot and Halbwachs [33].

Extended array regions

We describe in this paragraph the rules to build the extended array regions. We build two semantic functions, represented as grammar attributes. $S.i$ (resp. $S.w$) corresponds to the function associating to the statement S and to an iteration vector the set of array elements that are read (resp. written) by S and the operations that last read (resp. last wrote) them. We define one attribute by array structure.

So as to give the rules to build $S.i$ and $S.w$, we define some other useful functions: $\mathsf{LOOP}[\![S]\!]x$ is the iteration domain of S, when the indices of all loops

but the innermost are given and equal to x. $\mathsf{COND}[\![E]\!]$ is the set of iteration vectors x surrounding the boolean expression E such that $\mathsf{V}[\![E]\!]x$ is true.

$$\begin{aligned}\mathsf{LOOP}[\![S]\!] &= \lambda x'.\left\{x \,\middle|\, x[1..S.p-1]=x', x[S.p]\in S.iter(x')\right\},\\ \mathsf{COND}[\![E]\!] &= \lambda x'.\left\{x \,\middle|\, \mathsf{V}[\![E]\!]x, x=x'\right\}\end{aligned}$$

We present in detail the rules concerning the assignment, the sequence and the `do` loop. All the rules are presented in Table 4.1.

Assignment : $S \to \mathtt{id}(E_1) = E_2$. The elements read by S are the elements read by E_1 and E_2. Hence, $S.i = E_1.i \sqcup E_2.i$. The array elements defined by S for operation $\langle S.l, x\rangle$ are the elements $\mathtt{id}(z)$ with $z = E_1(x)$. Therefore,

$$S.w = \lambda x.\left\{z, \langle S.l, x'\rangle \,\middle|\, \mathsf{V}[\![E_1]\!]x' = z, x' = x\right\}.$$

Sequence : $B \to B_1;\ S$. The elements written by B are the elements defined by S and the elements defined by B_1 that have not been redefined by S. Hence, $B.w = B_1.w \sqcup S.w$. Note that, with this definition and according to the definition of $\sqcup$, if $B_1.w$ defines some elements that are redefined by $S.w$, the definitions of $B_1.w$ are killed. The elements imported by B are the elements imported by S that have not been defined by B_1 and the elements imported by B_1. Therefore, $B.i = (S.i \boxminus B_1.w) \sqcup B_1.i$.

Loop : $S \to$ `do` $x = E_1,\ E_2$ `step` $E3;\ B_1;$ `enddo`. The elements imported by the loop are those read by E_1, E_2, E_3 and the elements imported by B_1 for all iterations ($B_1.i{\circ}\mathsf{LOOP}[\![B_1]\!]$) that have not been redefined by an operation of B_1 in a following iteration of the loop. Therefore,

$$S.i = E_1.i \sqcup E_2.i \sqcup E_3.i \sqcup (B_1.i{\circ}\mathsf{LOOP}[\![B_1]\!] \boxminus B_1.w{\circ}\mathsf{LOOP}[\![B_1]\!]).$$

As for the elements defined by the loop, they are the elements defined by B_1, for all iterations ($B_1.w{\circ}\mathsf{LOOP}[\![B_1]\!]$). We only consider the latest definition of these elements: $S.w = \max_{\prec_{d_{B_1.l}}}(B_1.w{\circ}\mathsf{LOOP}[\![B_1]\!])$. This operator corresponds in Creusillet's framework to a *least fixed point*.

These rules correspond to the semantic functions that defined respectively in [34] exact IN and WRITE regions. Hence, projecting out the operations in the representation of the attributes $.i$ and $.w$ produces respectively IN and WRITE exact regions. We use inherited attributes to build dependence relations. The attribute $.inf$ defines the function that associates to a (compound) statement and to an iteration vector the set of array elements associated to the operations of the preceding iterations that last defined these elements. In order to integrate the sequencing order and the different dependence depths, we introduce the function:

$$\mathsf{PREC}[\![S]\!] = \lambda x'.\left\{x \,\middle|\, x[1..S.p-1] = x'[1..S.p-1], x[S.p] < x'[S.p]\right\}.$$

$S \to \mathtt{id}(E_1) = E_2$	$S.w = \lambda x. \{z, \langle S.l, x' \rangle \,\vert \mathsf{V}[\![E_1]\!]x' = z, x' = x\}$ $S.i = E_1.i \sqcup E_2.i$
$S \to \mathtt{do}\ x = E_1,\ E_2\ \mathtt{step}\ E3;$ B_1 $\mathtt{enddo}$	$S.w = \max_{\prec_{d_{B_1.l}}}(B_1.w \circ \mathsf{LOOP}[\![B_1]\!])$ $S.i = E_1.i \sqcup E_2.i \sqcup E_3.i$ $\sqcup\,(B_1.i \circ \mathsf{LOOP}[\![B_1]\!] \boxminus B_1.w \circ \mathsf{LOOP}[\![B_1]\!]$
$S \to \mathtt{do}\ x = 1\ \mathtt{while}\ E;$ B_1 $\mathtt{endwhile}$	$S.w = \max_{\prec_{d_{B_1.l}}}(B_1.w \circ \mathsf{LOOP}[\![B_1]\!])$ $S.i = (E.i \sqcup B_1.i) \circ \mathsf{LOOP}[\![B_1]\!] \boxminus B_1.w \circ \mathsf{LOOP}[\![B_1]\!]$
$S \to \mathtt{if}\ E\ \mathtt{then}\ B_1\ \mathtt{else}\ B_2$	$S.w = B_1.w \circ \mathsf{COND}(E) \sqcup B_2.w \circ \mathsf{COND}(\neg E)$ $S.i = E.i \sqcup B_1.i \circ \mathsf{COND}(E) \sqcup B_2.i \circ \mathsf{COND}(\neg E)$
$B \to B_1;\ S$	$B.w = B_1.w \sqcup S.w$ $B.i = (S.i \boxminus B_1.w) \sqcup B_1.i$
$B \to ;$	$B.w = \emptyset$ $B.r = \emptyset$
$E \to \mathtt{f}(\ldots, \mathtt{id}(E), \ldots)$	$E.r = \lambda x. \{z, \langle E.l, x' \rangle \,\vert \mathsf{V}[\![E]\!]x' = z, x' = x\}$
$E \to \mathtt{iter}$	$E.r = \emptyset$

Table 4.1: Extended regions.

The attribute $.inf$ is build with the rules described in Table 4.2. Dependence relations for statement S are then deducted from $S.inf$ and $S.i$. If $S.inf = \lambda x'. \{z, \langle \mathtt{S}, x \rangle \,|r_{inf}(x, x', z)\}$ and $S.i = \lambda x'. \{z, \langle \mathtt{R}, x \rangle \,|r_i(x, x', z)\}$, then we build the dependence relation:

$$\left\{ \langle \mathtt{S}, x \rangle \to \langle \mathtt{R}, x' \rangle \,\middle|\, \exists z, \exists x'', r_i(x', x'', z) \wedge r_{inf}(x, x'', z) \right\}.$$

Note that with this method, we can compute dependence relations between compound statements.

Comparing approximations

An array is privatizable in a loop if none of the reads depends on writes performed in a previous iteration of this loop. In terms of regions, an array is privatizable if the intersection of an IN region with a WRITE region for a previous iteration is always empty. In terms of dependence relations, the relation between a read and a write of a previous iteration is empty.

In the framework of extended regions, an array structure is privatizable for a loop `do ..; S; enddo` if:

$$S.inf \sqcap S.i = \emptyset.$$

If $S.inf = \lambda x. \{z, \tau \,|r_{inf}(z, x, \tau)\}$ and $S.i = \lambda x. \{z, \tau \,|r_i(z, x, \tau)\}$ then

$$S.inf \sqcap S.i = \lambda x. \{z, \tau \,|\exists \tau_i, r_{inf}(z, x, \tau) \wedge r_i(z, x, \tau_i)\}.$$

Projecting out the operation component of this set (τ), we obtain the conditions on array regions for the array to be privatizable. Projecting out the array

$S \rightarrow$ `do` $x = E_1,\ E_2$ `step` $E3;$ B_1 `enddo`	$B_1.inf = S.inf \sqcup S.w \circ \mathsf{PREC}[\![B_1]\!]$ $E_k.inf = S.inf, 1 \le k \le 3$
$S \rightarrow$ `do` $x = 1$ `while` $E;$ B_1 `endwhile`	$B_1.inf = S.inf \sqcup S.w \circ \mathsf{PREC}[\![B_1]\!]$ $E.inf = B_1.inf$
$S \rightarrow$ `if` E `then` B_1 `else` B_2	$B_1.inf = S.inf$ $B_2.inf = S.inf$
$B \rightarrow B_1;\ S$	$S.inf = B.inf \sqcup B_1.w$ $B_1.inf = B.inf$
$P \rightarrow B$	$B.inf = \emptyset$

Table 4.2: Extended regions for preceding iterations.

element component of this set (z), we obtain similar conditions on dependence relations. Note however that the array element can be easily retrieved when the operation and array structure are known. Thus $S.inf \sqcap S.i$ can be deduced from the dependence relation.

Now, we test the emptiness of dependence relations defined with non-affine constraints by testing the emptiness of an approximated relation. Likewise, array regions are not computed exactly but over- and under-approximated. To improve the quality of the approximation, both approaches use the relations found by the analyses of preconditions and transformers (continuation conditions are of no use in our program model). These relations are affine relations on the variables of the program, therefore the resolution method applies (see Section 4.8.3 for the adaptation of this method to dependence relations). When the conditions of Theorem 4.1 are met, we are able to calculate a dependence relation that is as accurate as possible w.r.t. the properties of non-affine constraints. In other words, since $S.inf \sqcap S.i$ can be deduced from the dependence relation, under the conditions of Theorem 4.1, the best affine approximation of $S.inf \sqcap S.i$ can therefore be computed. This implies that the affine approximation of the projection of $S.inf \sqcap S.i$ corresponding to array regions do not find more privatizable arrays than dependence relations.

To conclude, we have shown, thanks to the formalism of extended regions, that FADA or a similar analysis enables privatization techniques to expose in some cases at least as much parallelism as region-based techniques, such as Creusillet's. This is the case when properties on non-affine constraints can be translated into properties on affine constraints without adding fuzziness. Preconditions and transformers, used in Creusillet's framework, are affine relations on the variables that belong to this class of properties, since only variables of the same iteration are compared. (See Section 4.5.1 for a discussion about the kind of properties that can be handled without adding fuzziness.)

4.9 Conclusion

The purpose of this chapter was to present an approximate array dataflow analysis that can handle any constraint that is not affine with respect to loop indices. These constraints are introduced in programs by non-affine array subscripts, `while` loops, non-affine predicates of conditionals or non-affine `do` loop bounds. We have proposed an analysis that is an extension of the polyhedral method described by Feautrier: Not only is the analysis exact on static control programs but also, as the approximation is performed in the last steps of the analysis, the techniques of source computation are the same as the ones used for static control programs. Moreover, in order to improve its accuracy, our analysis is able to take into account a large class of properties characterizing non-affine constraints, properties that can be produced by the many existing symbolic analyses. In addition to these existing techniques, we proposed an original method, called iterative analysis, that finds relations between values of non-affine constraints by examining, possibly iteratively, the sources of the variables they use. Finally, we have found the condition for which the approximation achieved is the best one can obtain with respect to the properties on non-affine constraints. We have implemented a previous version of our analysis so as to validate our theoretical framework. Figure 4.6 sums up the different steps of the computation of an approximate source.

A detailed comparison has been performed with the method proposed by Wonnacott, and beyond the differences due to the representations, we have proved that our method can apply in a framework of dependence relations. Moreover, we have shown that the condition on the accuracy of the result is also valid in this formalism.

As for region analyses, we have introduced a representation from which regions and sources can be derived. Although not directly computable, the summaries obtained in this representation have allowed us to prove that no more parallelism could be extracted with array regions than with FADA, as long as no fuzziness is added by our analysis. Whether this intermediate representation has some interest for an implementation of a FADA that could degrade into array region analysis is uncertain and left for future work.

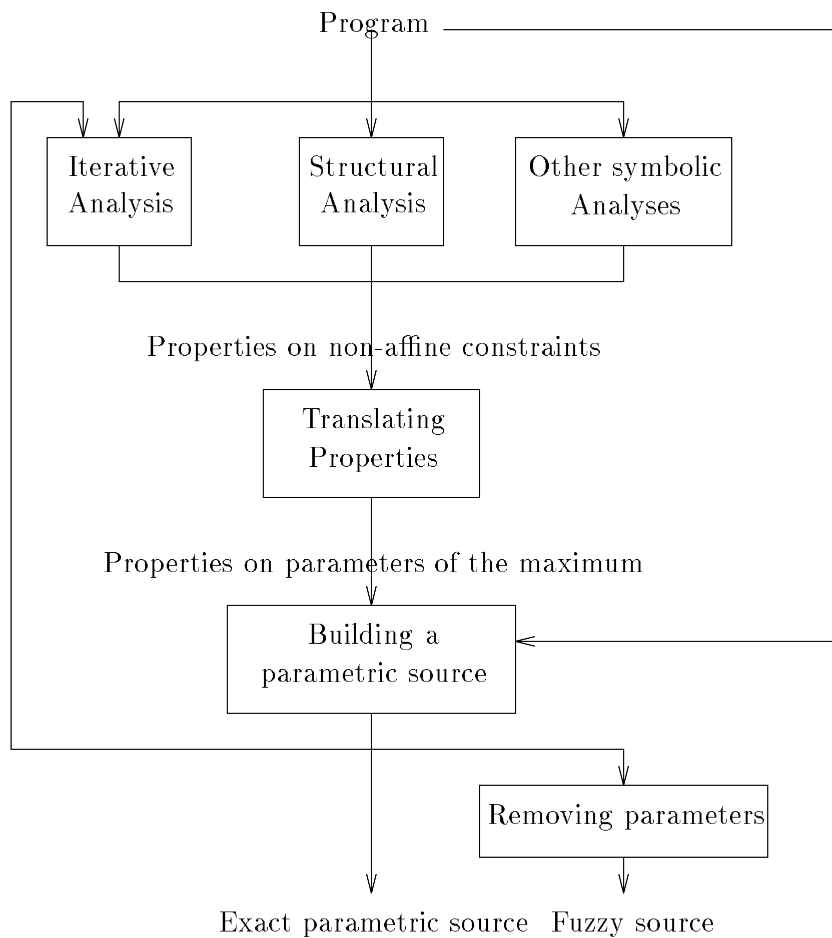


Figure 4.6: Steps of the computation of an approximate source.

Chapter 5

Applications

We provide in the previous chapter a general method to compute the DFG of a program even when non-affine constraints are involved in the computation. We propose to perform the approximation in the last steps of the analysis, by removing parameters. In general, making an approximation in the latest stages of a computation provides the best results. In fact, we believe that in some cases, the approximation should not take place at the dependence computation step but at the application step. Indeed, the DFG is computed for each data structure and the properties on non-affine constraints only concern one data structure at a time. Applications needing some knowledge about the flows of different data structures at the same time could take advantage of the relations between the non-affine constraints involved in the computation of different flows. Such is the case for instance of the computation of a schedule or of a recurrence detection. In the same manner as before, these relations may be established by complementary techniques such as symbolic analysis and then transformed into properties on the parameters of the maximum. Note however that the nature of the accuracy entailed by such technique has changed, since we do no longer improve the output of the dataflow analysis but of the applications using the output of the dataflow analysis. Concerning scheduling, we show in this chapter that more information about non-affine constraints may save for instance a speculative execution, but cannot improve the latency. Keeping the parameters through the application is not always necessary to improve the accuracy of the application and we show that static memory expansion only needs an approximate dataflow graph as input.

Many usual applications of exact array dependence analysis can be used without any change or with only minor modifications with an approximate analysis. Brandes [23] proposed several applications of array dependence analysis in the compilation field. We first present program checking (see Section 5.1), a technique that checks if all the variables of a program (or of a fragment of program) are defined before being used. Once again, we use the characterization of the parameters of the maximum so as to find predicates that must be

verified so as to ensure a correct initialization of the variables. Then, we will describe the slight adaptations of traditional code transformations in Section 5.2 and of recurrence detection in Section 5.3. We propose an expansion of array structures, which is able to cope with an approximate DFG, in Section 5.4. Finally, we conclude by the adaptation of scheduling techniques.

5.1 Program Checking

In a correct program, all variables are initialized before they are used. Detecting uninitialized variables is a very simple application of array dataflow analysis. Indeed, a variable is initialized if and only if $\perp$ is not among the possible sources found by a dataflow analysis. This simple fact can help the programmer to check the correctness of his program and to validate some properties of non linear constraints.

Let us consider a quast obtained by a dataflow analysis of a given variable, after removing the parameters due to fuzziness (if any). If a leaf of this quast is $\perp$, it means that the variable is not correctly initialized. Indeed, the conjunction of the predicates on the path of the quast from the root to $\perp$ represents the condition on y, the iteration vector of the read, for which there is no initialization. This condition is verified by at least one value of y since otherwise the leaf would have been eliminated in the simplification process.

Consider now a direct dependence from statement R to statement S at depth p. The following results can be extended to a source built from any number of statements at any depth. When $\perp$ is one of the possible sources, it means that some of these non-linear constraints may not yield a correct initialization of the variable. We give in the sequel a characterization of all possible non-linear constraints for which the program is correct and then check whether the actual constraints comply to this characterization.

Let us consider the expression of the quast with the parameter of the maximum $\alpha^p_{\mathtt{SR}}(y)$ (before removing the parameters). Let us sort the predicates from the root to a $\perp$ leaf into $r_i(y), 0 \leq i \leq m$, depending only on y, and $s_j(y, \alpha^p_{\mathtt{SR}}(y)), 0 \leq j \leq n$, depending also on a parameter of the maximum. For the initialization of the variable to be correct, the $\perp$ leaf should never be reached, i.e. the following condition must be fulfilled:

$$\forall y \in \mathsf{I}(\mathtt{R}), \left(\bigwedge_{0 \leq i \leq m} r_i(y) \right) \wedge \left(\bigwedge_{0 \leq j \leq n} s_j(y, \alpha^p_{\mathtt{SR}}(y)) \right) = \text{false},$$

which is equivalent to: for all $y \in \mathsf{I}(\mathtt{R})$ verifying $\wedge_{0 \leq i \leq m} r_i(y)$ then:

$$\bigvee_{0 \leq j \leq n} \neg s_j(y, \alpha^p_{\mathtt{RS}}(y)).$$

Thanks to the definition of $\alpha^p_{\mathtt{RS}}(y)$, this is equivalent to the following condition:

$$\forall y \in \mathsf{I}(\mathtt{R}) \text{ s.t. } \bigwedge_{0\le i\le m} r_i(y), \exists x \text{ s.t. } \left(\bigvee_{0\le j\le m} \neg s_j(x,y)\right) \wedge \left(\bigwedge_{h\in \mathsf{C}_{\mathtt{SR}}} c_h(x,y)\right) \quad (5.1)$$

This characterization of the non-linear constraints is a necessary and sufficient condition for the program to initialize correctly all used variables.

The following piece of code illustrates this property:

```
   do x = 1, n
S     A(f(x)) = ..
   enddo
   do y = 1, n
R     .. = A(y)
   enddo
```

The non-affine constraint is $c_1(x,y) = (f(x) = y)$. Without any complementary information about f, the fuzzy source of $\langle \mathtt{R}, y\rangle$ is formed by $-$ and $\{\langle \mathtt{S}, x\rangle \,|1 \le x \le n\}$. Therefore we will give a property on the non-linear constraint in order to have a correct initialization of `A`. The quast with the parameter of the maximum is:

$$\textbf{if } 1 \le \alpha^0_{\mathtt{SR}}(y) \le n \textbf{ then } \langle \mathtt{S}, \alpha^0_{\mathtt{SR}}(y)\rangle \textbf{ else } -.$$

Hence the direct application of the characterization given by (5.1) for $r(y) = (1 \le y \le n)$ (the environment) and $s(x,y) = \neg(1 \le x \le n)$ leads to:

$$\forall y \text{ s.t. } 1 \le y \le n, \exists x \text{ s.t. } 1 \le x \le n, f(x) = y.$$

In other words, `A` is initialized iff f is a permutation on $1..n$.

Checking this condition can be left to the programmer or submitted to an assertion generator in the manner of Floyd [56]. For obvious psychological reasons, such a verifier will be used only if it finds no more $-$ than necessary. Hence our insistence in the previous chapter on not adding fuzziness.

Program checking can also be performed on smaller pieces of code such as subroutines bodies, so as to compare the variables and the conditions on which they are initialized outside the code with what the programmer expected.

5.2 Traditional Code Transformations

Classical reordering transformations can take advantage of the accuracy of an array dataflow analysis, fuzzy or not, which is more precise than the usual use-def chain technique. This latter method is not able to cope efficiently with arrays. Among the transformations which may use an array dataflow analysis, we can cite:

- dead code elimination, which is often combined with copy propagation,

- code motion coming from common subexpression and loop invariant detections.

Brandes [23] described these optimizations in the case of exact direct dependences. The adaptation of the standard algorithms to fuzzy array dataflow analysis is straightforward. Notice however that as the analysis is performed on operations instead of statements, the optimizations can precisely remove some of the operations or move them outside the loops. Loop peeling may then be used to particularize some loop iterations.

The following example illustrates such a case:

original code	after optimization
`      z = 0`	`z = 0`
`      do i = 1, n`	`do i = 1, n`
`S1       A(z) = i`	`   A(z) = i`
`S2       z = ..`	`   z=..`
`S3       A(z) = 0`	`enddo`
`      enddo`	`A(z) = 0`
`      do j = 1, n`	`do j = 1, n`
`S4       ..  = A(j)`	`   ..  = A(j)`
`      enddo`	`enddo`

Indeed, a fuzzy iterative analysis finds that the source of `A(j)` in operation $\langle \mathtt{S}_4, j\rangle$ is $\{-, \langle \mathtt{S}_3, n\rangle, \langle \mathtt{S}_1, \alpha\rangle \,|1 \leq \alpha \leq n\}$. Operations $\langle \mathtt{S}_3, i\rangle$, for $i = 1..n-1$ can be removed safely, and $\langle \mathtt{S}_3, n\rangle$ is placed outside the loop.

5.3 Detection of Recurrences

Redon defines in [111] a method to detect and normalize recurrences in a program. The advantages of such a method are obvious for software engineering and parallelizing compilers: the normalization gives the programmer a way to retrieve the semantic of the operations performed, whatever the manner they are implemented. Besides, the detection in itself enables significant improvement on the schedule, placement and code produced by a parallelizing compiler which can use libraries with optimized versions of the recurrences, if they exist.

The detection of the recurrences is based on a *system of linear recurrence equations* (SLRE), obtained from the dataflow graphs of the variables of the program. Obviously, the detection can use the result produced by a structural or iterative fuzzy dataflow analysis whenever it is exact. Structural analysis enables the detection of recurrences that are nested in while loops and branches of conditionnals. Iterative analysis enables the use of variables that are not loop counters as the indices of the recurrence, as long as there is enough information about these variables for the array sources to be exact.

We give in the following the conditions for which the detection of recurrences works with non exact sources. The reader will find in [111] the details

about the general detection. Each equation of an SLRE gives the expression of a variable of the program in function of the other variables. This expression is a conditional function depending on iteration vectors and parameters of fuzziness, when the variable has a fuzzy source. Each possible kind of value given by this conditional are called clauses. The method then builds a system graph whose edges are the couples of equations such that the variable defined in the first equation is used in a clause of the second one. The oriented cycles in this graph are then normalized so that a pattern-matching algorithm applied on each equation can decide whether or not it is a recurrence. Dependences that do not appear in the cycles of the system graph do not interfere with the algorithm. Consequently, variables with fuzzy sources must not be a recurrence accumulator, unless the only fuzzy sources are for the initial value of the accumulator.

The following example cannot be handled by Redon, because the subscripts of `M` and `V2` are non-affine, and the bounds of j are non-affine. This code represents a matrix-vector multiplication where the matrix is in sparse-column format and has blocks of non-null lines.

```
  do i = 1, n
   do j = ib(i-1), ib(i)-1
S1  V1(j) = 0
    do k = 1, n
S2   V1(j) = V1(j) + M(j,X(k))*V2(X(k))
    enddo
   enddo
  enddo
```

A fuzzy analysis of the source of `V1` for operation $\langle \mathtt{S}_2, i, j, k\rangle$ is exact when the structural analysis is taken into account. The source is then:

$$\textbf{if } k > 1 \textbf{ then } \langle \mathtt{S}_2, i, j, k-1\rangle \textbf{ else } \langle \mathtt{S}_1, i, j\rangle\,.$$

Indeed, when `V1` is read by the operation $\langle \mathtt{S}_2, i, j, k\rangle$, the constraint $\mathtt{ib}(i-1) \le j \le \mathtt{ib}(i)-1$ is verified. This environment constraint is added to the context of the computation and it entails that the write operation $\langle \mathtt{S}_1, i, j\rangle$ is executed. This operation kills all previous fuzzy sources. Note that the sources of `M` and `V2` at operation $\langle \mathtt{S}_2, i, j, k\rangle$ are fuzzy but that does not prevent the detection of reccurence.

Putting the variable `V1` into SA form, $\mathtt{V1}(j)$ is renamed $\mathtt{V11}(i,j)$ for statement $\mathtt{S}_1$ and $\mathtt{V12}(i,j,k)$ for statement $\mathtt{S}_2$. Using the notations of [110], the SLRE associated to `V1` is:

$$\begin{array}{lcl}
\mathtt{V11} & : & \{i, j \,|\, 1 \le i \le n, \mathtt{ib}(i-1) \le j \le \mathtt{ib}(i)-1\,\} \\
\mathtt{V12} & : & \{i, j, k \,|\, 1 \le i, k \le n, \mathtt{ib}(i-1) \le j \le \mathtt{ib}(i)-1\,\}
\end{array}$$

$$\begin{array}{lcl}
\mathtt{V11}(i,j) & = & 0 \\
\mathtt{V12}(i,j,k) & = & \text{case} \\
 & & \{i,j,k \,|\, k>1\,\} : \mathtt{V12}(i,j,k-1) + \mathtt{M}(j,\mathtt{X}(k)) * \mathtt{V2}(\mathtt{X}(k)) \\
 & & \{i,j,k \,|\, k=1\,\} : \mathtt{V11}(i,j) + \mathtt{M}(j,\mathtt{X}(k)) * \mathtt{V2}(\mathtt{X}(k)) \\
 & & \text{esac.}
\end{array}$$

After normalization of this SLRE and introduction a scan operator, we obtain:

$$
\begin{array}{l}
\texttt{V12}(i,j,k) = \\
\text{case} \\
\quad \{i,j,k\,|k>1\} \ : \text{Scan}(\quad \{i',j',k'|1 \le i',k' \le n, \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{ib}(i'-1) \le j' \le \texttt{ib}(i')-1\}, \\
\qquad\qquad\qquad\qquad\qquad [0\ 0\ 1], +, \\
\qquad\qquad\qquad\qquad\qquad \texttt{M}(j,\texttt{X}(k)) * \texttt{V2}(\texttt{X}(k)), \\
\qquad\qquad\qquad\qquad\qquad \texttt{M}(j,\texttt{X}(k)) * \texttt{V2}(\texttt{X}(k)) \\
\text{esac}
\end{array}
$$

In other words, the sum represented by the scan is:

$$\sum_{k=1}^{n} \texttt{M}(j,\texttt{X}(k)) * \texttt{V2}(\texttt{X}(k)).$$

5.4 Maximal Static Expansion

Expansion of data structures is a well-known general technique to break spurious data dependences that hamper program parallelization. In parallel processing, expanding a datum also allows to place one copy of the datum on each processor, enhancing parallelism. This technique is known as array privatization and is extremely important to parallelizing and vectorizing compilers [83, 17, 99]. A similar technique is register or variable renaming.

In the extreme case, each memory cell is written at most once and the program is said to be in single assignment form. Unfortunately, when the control flow cannot be predicted at compile-time, some run-time computation is needed to preserve the original data flow: In the *static single assignment* framework, ϕ-functions may be needed to "merge" multiple reaching definitions, i.e. possible data definitions due to several incoming control paths [36, 37]. Such ϕ-functions may be an overhead at run-time, especially for non-scalar data structures or when replicated data are distributed across processors. We are thus looking for a *static expansion*, i.e. an expansion of data structures that does not need a ϕ-function. (Notice that according to our definition, an expansion in the *static single assignment* framework may *not* be static.) Our goal is to automatically find the static expansion which expands all data structures as much as possible, i.e. the *maximal static expansion*. Maximal static expansion may be considered as a trade-off between parallelism and memory usage.

We present an algebraic framework in which to derive the maximal static expansion. The input of this framework is the (perhaps inaccurate) output of a value-based dependence analysis and our method is "optimal" with respect to the precision of this analysis. Our framework is valid for array structures and for any program on which a value-based analysis can be performed. In particular, this framework is valid for programs fulfilling the requirements of

Section 4.1. The results presented in this section are joint work with Cohen and Collard [10].

Section 5.4.1 studies motivating examples showing what we want to achieve. Section 5.4.3 formally states what (maximal) static expansion is, and Section 5.4.4 presents a general framework to solve this problem. This framework is applied in Section 5.4.5 to derive an algorithm for maximal static expansion. Section 5.4.6 applies this algorithm to the motivating examples, before our conclusion.

5.4.1 Motivating examples

First Example: Dynamic Control Flow

We first study the code of Example E1, presented in Section 4.2.2:

```
    do x1 = 1 to n
S1    s = ..
      do x2 = 1 while ..
S2      s = .. s ..
      enddo
R     .. = .. s ..
    enddo
```

Let us try to expand scalar s. One way is to convert the program into SA form, making $\texttt{S}_1$ write into $\texttt{s}'(x_1)$ and $\texttt{S}_2$ into $\texttt{s}''(x_1, x_2)$: Then, each memory cell is assigned to at most once, complying with the definition of SA. However, what should right-hand sides look like now? A brute-force application of the source functions of s found in 4.6.3 yields the program in Figure 5.1. While the right-hand side of $\texttt{S}_2$ only depends on w, the right-hand side of R depends on the control flow, thus needing a function similar to a ϕ-function in the SSA framework (even if, in this introductory example, the ϕ-function would be very simple) [61].

```
  do x1 = 1, n
S1  s'(x1) = ..
    do x2 = 1 while ..
S2    s''(x1,x2) = ..
        if (x2>1) then s''(x1,x2-1) else s'(x1) ..
    enddo
R   ..  = ..  φ(⟨S1, x1⟩ , {⟨S2, x1, x2⟩ |x2 ≥ 1})
  enddo
```

Figure 5.1: Single Assignment for Example E1.

Our aim is to expand s as much as possible in this program *but* without having to insert ϕ-functions.

A possible static expansion is to uniformly expand s into s(x_1) and avoid output dependencies between distinct iterations of the do loop. The resulting *maximal static expansion* of this example is given by Figure 5.2. It has the same degree of parallelism and is simpler than the program in single-assignment.

```
   real s(1:n)

   do x1 = 1, n
S1  s(x1) = ..
    do x2 = 1 while ..
S2   s(x1) = .. s(x1) ..
    enddo
R   .. = .. s(x1) ..
   enddo
```

Figure 5.2: Expanded version of Example E1.

Notice that it should be easy to adapt the array privatization techniques by Maydan et al. [93] to handle the program in Figure 5.2; This would tell us that s can be privatized along x_1. However, we want to do more than privatization along loops, as illustrated in the following examples.

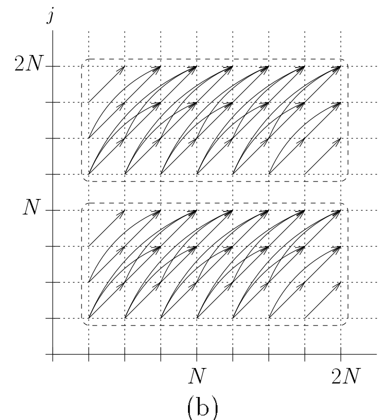
Second Example: Array Expansion

Let us give a more complex example; We would like to expand array A in the program in Figure 5.3.

Since T always executes when j equals N, a value read by $\langle$S$, i, j\rangle$, $j > N$ is never defined by an instance $\langle$S$, i', j'\rangle$ of S with $j' \leq N$. Figure 5.3 describes the dataflow relations between S instances: An arrow from (i', j') to (i, j) means that instance (i', j') defines a value that *may* reach (i, j).

```
  real A(4*N-1)
  do i=1, 2*N
    do j=1, 2*N
      if .. then
S       A(i-j+2*N) = .. A(i-j+2*N)
      endif
      if j = N then
T       A(i+N) = ..
      endif
    enddo
  enddo
```

(a)



(b)

Figure 5.3: Second example of expansion.

Formally, the source of one instance of statement S is:

$$\sigma(\mathtt{A}, \langle \mathtt{S}, i, j\rangle) = \left| \begin{array}{l} \textbf{if } j \leq N \\ \textbf{then } \left\{ \langle \mathtt{S}, i', j'\rangle \;\middle|\; \begin{array}{l} 1 \leq i' \leq 2N \\ \wedge 1 \leq j' < j \wedge i' - j' = i - j \end{array} \right\} \\ \textbf{else } \left\{ \langle \mathtt{S}, i', j'\rangle \;\middle|\; \begin{array}{l} 1 \leq i' \leq 2N \\ \wedge N < j' < j \wedge i' - j' = i - j \end{array} \right\} \\ \qquad \cup \{\langle \mathtt{T}, i', N\rangle \,|1 \leq i' < i \wedge i' = i - j + N\} \end{array} \right. \tag{5.2}$$

Because sources are non-singleton sets, converting this program to (dynamic) SA form would require run-time computation of the memory location read by S.

However, we notice that the iteration domain of S may be split into *disjoint subsets* by grouping together operations involved in the same data flow. These subsets build a partition of the iteration domain. Each subset may have its own memory cell, a cell that will not be written nor read by operations outside the subset. The partition is given in Figure 5.4.a.

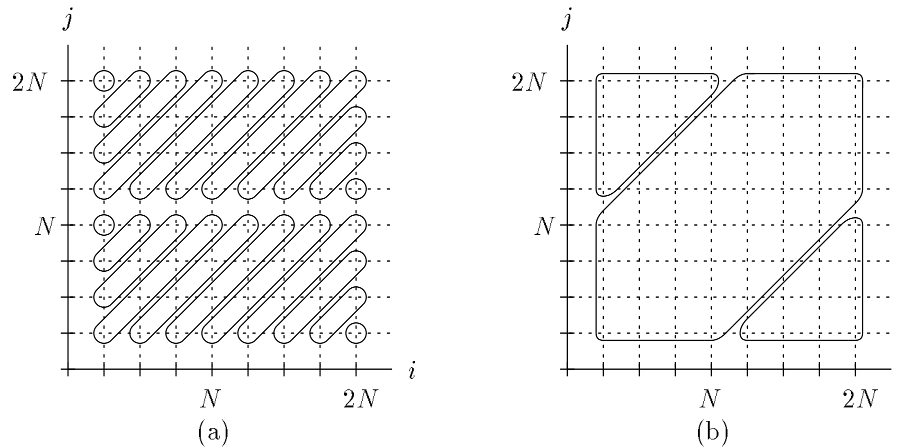


Figure 5.4: Partition of the iteration domain($N = 4$).

Using this property, we can duplicate only those elements of A that are used twice. These are all the array elements $\mathtt{A}(c)$, $1 + N \leq c \leq 3N - 1$. They are accessed by operations in the large central set in Figure 5.4.b. Let us label with 1 the subsets in the lower half of this area, and with 2 the subsets in the top half. We add one dimension to array A, subscripted with 1 and 2 in statements $\mathtt{S}_2$ and $\mathtt{S}_3$ in Figure 5.5, respectively. Elements $\mathtt{A}(c), 1 \leq c \leq N$ are accessed by operations in the upper left triangle in Figure 5.4.b and have only one subset each (one subset in the corresponding diagonal in Figure 5.4.a, which we label with 1. The same labeling holds for sets corresponding to operations in the lower right triangle.

The maximal static expansion is shown in Figure 5.5. Notice that this program has the same degree of parallelism as the corresponding single-assignment program, without the run-time overhead and memory spare.

```
  real A(4*N,2)
  do i=1, 2*N
    do j=1, 2*N

C     expansion of statement S

      if -2*N+1 <= i-j <= -N then
        if .. then
          A(i-j+2*N,1) = .. A(i-j+2*N,1) ..
        endif
      elsif -N+1 <= i-j <= N-1 then
        if j <= N then
          if .. then
            A(i-j+2*N,1) = .. A(i-j+2*N,1) ..
          endif
        elsif .. then
          A(i-j+2*N,2) = .. A(i-j+2*N,2) ..
        endif
      elsif .. then
        A(i-j+2*N,1) = .. A(i-j+2*N,1) ..
      endif

C     expansion of statement T

      if j = N then A(i+N,2) = .. endif
    enddo
  enddo
```

Figure 5.5: Maximal static expansion for the second example.

Third Example: Non-Affine Array Subscripts

Consider the program in Figure 5.6.a, where *foo* and *bar* are arbitrary subscripting functions[1]. Since all array elements are assigned by `T`, the value read by `R` at the ith iteration must have been produced by `S` or `T` at the same iteration. The data-flow graph is similar to the first example:

$$\sigma(\mathtt{A}, \langle \mathtt{R}, i\rangle) = \left\{ \langle \mathtt{S}, i\rangle \right\} \cup \left\{ \langle \mathtt{T}, i, j\rangle \,|\, 1 \leq j \leq N \right\}. \tag{5.3}$$

The maximal static expansion adds a new dimension to `A` subscripted by i. It is sufficient to make the first loop parallel.

[1] `A`$(foo(i))$ stands for an array subscript between 1 and N, "too complex" to be analyzed at compile-time.

```
  real A(N)
  do i=1, N
    do j=1, N
T     A(j) = ..
    enddo
S   A(foo(i)) = ..
R   .. = .. A(bar(i))
  enddo
```

Source program.
(a)

```
  real A(N,N)
  do i=1, N
    do j=1, N
T     A(j,i) = ..
    enddo
S   A(foo(i),i) = ..
R   .. = .. A(bar(i),i)
  enddo
```

Expanded version.
(b)

Figure 5.6: Third example of expansion.

These examples show the need for an automatic static expansion technique. We present in the following section a formal definition of expansion and a general framework for maximal static expansion. We then describe an expansion algorithm for arrays that yields the expanded programs shown above. Notice that it is easy to recognize the original programs in their expanded counterparts, which is a practical property of our algorithm.

5.4.2 Related work

If the input program is built of nested `do` loops with affine bounds and accesses arrays with affine subscripts, one can find a static expansion which is also in single-assignment form [49].

In the case of programs with general control and unrestricted arrays subscripts, array data-flow analyses are approximate: Several writes may be the unique definition of a given value, but the analysis cannot tell. [61] describes how to obtain a single-assignment program to the price of dynamic restoration of data flow.

Many studies are related to array privatization. As hinted above, Maydan et al. [93] proposed an algorithm to privatize arrays, based on value-based dependence analysis. However, their method only applies to static control programs. Tu and Padua [123] proposed a privatization technique for a very large class of programs. But it resorts to dynamic restoration of data flow. Another accurate approach using array regions has been described by Creusillet [34]. Her method avoids the cost of a dynamic restoration and copies back the privatized elements into the original arrays.

However, privatization only detects parallelism along the enclosing loops; It is thus less powerful than general array expansion techniques. Indeed, the example in Section 5.4.1 shows that our method may not only expand along diagonals in the iteration space but may also do some "blocking" along these diagonals.

5.4.3 Static expansion

Let Ω be the set of all operations in the program, f the function mapping operations to memory cells they write into, and $\mathsf{W} \subseteq \Omega$ be the set of all writes. Let f' be the expansion, that is the *new* function, after program transformation, mapping operations to the memory cells they write into.

Let us consider two operations τ and ζ belonging to the same set of possible sources of some read ν. If they both write in the same memory cell $f(\tau) = f(\zeta)$ and if we assign two distinct memory cells to τ and ζ ($f'(\tau) \neq f'(\zeta)$), then a ϕ-function is needed to restore the data flow since we do not know which of the two cells has the value needed by τ. Note that the fact that τ and ζ belong to the same source set does not implies that $f(\tau) = f(\zeta)$. Such cases may occur when dealing with non-affine array subscripts.

Definition 5.1 (Static expansion) *A static expansion is a mapping f' from operations to memory cells such that*

$$\forall \tau, \zeta \ : \ (\exists \nu \ : \ \tau \in \sigma(\nu) \wedge \zeta \in \sigma(\nu) \wedge f(\tau) = f(\zeta)) \Longleftrightarrow f'(\tau) = f'(\zeta).$$

Because the sources of a read are mapped to the same memory cell by f', static expansion preserves the original dataflow graph.

Notice also that, according to this definition, even a constant function on W is a static expansion. Because we are interested in maximizing the memory expansion, the range of a "good" static expansion should be as large as possible. Such an expansion would be constant on sets as small as possible:

Definition 5.2 (Maximal static expansion) *A static expansion f' is maximal on the set of operations W if, for any static expansion f'',*

$$\forall \tau, \zeta \in \mathsf{W} \ : \ f'(\tau) = f'(\zeta) \Longrightarrow f''(\tau) = f''(\zeta).$$

Intuitively, if f' is maximal, then no f'' can do better and map two writes to two different memory cells when f' does not.

We need to characterize the sets of operations on which a maximal static expansion f' is constant. These sets correspond to the equivalence classes of the relation $\{\tau, \zeta \in \mathsf{W} \,|\, f'(\tau) = f'(\zeta)\}$, and the set of theses classes is denoted by W/f'. The number of memory cells after maximal static expansion is thus equal to the cardinal of W/f'.

However, this hardly gives us an expansion scheme, because this result does not tell us how much each individual memory cell should be expanded. The purpose of Section 5.4.4 is to give a similar result for each memory cell c used in the original program. This result appears in Theorem 5.1. This theorem is then used to give a practical expansion scheme.

5.4.4 Expansion scheme

Let us define the relation:

$$\tau \mathrm{R} \zeta \iff \exists \nu \ : \ \tau \in \sigma(\nu) \wedge \zeta \in \sigma(\nu). \tag{5.4}$$

σ is itself a relation on $\Omega \times \Omega$ and the reciprocal relation is denoted by σ^{-1}. Therefore, $\tau \mathrm{R} \zeta \iff \tau \sigma \circ \sigma^{-1} \zeta$, i.e., $\mathrm{R} = \sigma \circ \sigma^{-1}$. Relation R is obviously symmetric. Definition 5.1 requires that a static expansion f' verifies $f'(\tau) = f'(\zeta)$ when $f(\tau) = f(\zeta)$ and $\tau \mathrm{R} \zeta$. Given τ, ζ and ν in W, if $f(\tau) = f(\zeta) = f(\nu)$, $\tau \mathrm{R} \zeta$ and $\zeta \mathrm{R} \nu$ then $f'(\tau) = f'(\zeta) = f'(\nu)$. Therefore, given $\tau \in \mathsf{W}$, f' is constant on the set of all $\zeta \in \mathsf{W}$ such that $f(\tau) = f(\zeta)$ and $\tau \mathrm{R}^* \zeta$, R^* being the transitive closure of R. We may give an equivalent definition of a static expansion:

Definition 5.3 *A static expansion is a mapping f' from operations to memory cells such that*

$$\forall \tau, \zeta : \ \tau \mathrm{R}^* \zeta \wedge f(\tau) = f(\zeta) \implies f'(\tau) = f'(\zeta).$$

We now characterize any maximal static expansion in terms of R^* and f:

Lemma 5.1 *f' is a maximal static expansion if and only if*

$$\forall \tau, \zeta \in \mathsf{W} : \ \tau \mathrm{R}^* \zeta \wedge f(\tau) = f(\zeta) \iff f'(\tau) = f'(\zeta). \tag{5.5}$$

Proof Sufficient condition—the "if" part

Let f' be a mapping verifying $\forall \tau, \zeta \in \mathsf{W} \ : \ f'(\tau) = f'(\zeta) \iff \tau \mathrm{R}^* \zeta \wedge f(\tau) = f(\zeta)$. By definition, f' is a static expansion.

Let us show that f' is maximal. Suppose that for $\tau, \zeta \in \mathsf{W}$: $f'(\tau) = f'(\zeta)$. (5.5) implies $\tau \mathrm{R}^* \zeta$ and $f(\tau) = f(\zeta)$. Thus, from Definition 5.3, any static expansion f'' satisfies $f''(\tau) = f''(\zeta)$. Therefore, $f'(\tau) = f'(\zeta) \iff f''(\tau) = f''(\zeta)$, so f' complies with Definition 5.2.

Necessary condition—the "only if" part

Let f' be a maximal static expansion. Because f' is a static expansion, we only have to prove that $\forall \tau, \zeta \in \mathsf{W} \ : \ f'(\tau) = f'(\zeta) \iff \tau \mathrm{R}^* \zeta \wedge f(\tau) = f(\zeta)$.

On the one hand, $f'(\tau) = f'(\zeta) \iff f(\tau) = f(\zeta)$ because f is a static expansion. On the other hand, assume $f'(\tau) = f'(\zeta)$ and $\neg \tau \mathrm{R}^* \zeta$. We show that it contradicts the maximality of f': Let $f''(\nu) = f'(\nu)$ when $\neg \tau \mathrm{R}^* \nu$, and $f''(\nu) = c$ when $\tau \mathrm{R}^* \nu$, with $c \neq f'(\tau)$. f'' is a static expansion: By construction, $f''(\tau') = f''(\zeta')$ for any τ' and ζ' such as $\tau' \mathrm{R}^* \zeta'$. The contradiction comes from the fact that $f''(\tau) \neq f''(\zeta)$.

Let us define $\mathsf{M} = f(\mathsf{W})$ the set of all memory cells accessed by write operations, and for $c \in \mathsf{M}$, $\mathsf{W}(c) = \{\tau \in \mathsf{W} \,|\, f(\tau) = c\}$ the set of operations writing into c. Given $c \in \mathsf{M}$, the previous lemma entails that a static expansion f' is maximal iff

$$\forall \tau, \zeta \in \mathsf{W}(c) \ : \ f'(\tau) = f'(\zeta) \Longleftrightarrow \tau \mathrm{R}^* \zeta.$$

Therefore, classes of R^* in $\mathsf{W}(c)$ are exactly the sets we are looking for:

Theorem 5.1 *The sets on which a maximal static expansion f' is constant are described by:*

$$\mathsf{W}/f' = \bigcup_{c \in \mathsf{M}} \mathsf{W}(c)/\mathrm{R}^* \tag{5.6}$$

The equivalence classes defined in the Theorem gives the partitions intuitively found in Section 5.4.1, and the expansion factor of each individual memory cell c is $\mathrm{card}(\mathsf{W}(c)/\mathrm{R}^*)$. Consider for instance `A(0)` in Figure 5.4.a. The instances of `S` that belong to $\mathsf{W}(\mathtt{A}(0))$ are on the main diagonal: $\{(i,j)|1 \le i,j \le 2N \ \wedge \ i = j = 0\}$. R^* partitions these operations in exactly the two subsets depicted in the figure. To generate the transformed code, one has to remember which equivalent class an operation belongs to: Let φ be the function mapping an operation τ to a representative of its equivalence class. One may label each element of $\mathsf{W}(c)/\mathrm{R}^*$, or equivalently, label each element of $\varphi(\mathsf{W}(c))$. Such a labeling scheme is obviously arbitrary, but all programs transformed using our method are equivalent up to a permutation of these labels. We denote by $n(\tau)$ the label we choose for the elements of $\varphi(\mathsf{W}(f(\tau)))$. Then, $f' = (f, n)$.

Our expansion scheme depends on the calculation of the transitive closure of R and of $\mathsf{W}(c)$. We would like to stress the fact that the expansion produced is static and maximal with respect to the results yielded by these calculations, whatever their accuracy:

- The exact transitive closure may be too complicated and may therefore be over-approximated. The expansion factor of a memory cell c is then lower than $\mathrm{card}(\mathsf{W}(c)/\mathrm{R}^*)$. However, the expansion remains static and is maximal with respect to the transitive closure given to the algorithm.

- The sets $\mathsf{W}(c)$ may not be known precisely at compile-time. (For instance, when data structures are arrays with non-affine subscripts.) One may use some approximation $\widetilde{\mathsf{W}(c)}$ instead, such that $\mathsf{W}(c) \subseteq \widetilde{\mathsf{W}(c)}$, and expand c into as many cells as elements in $\widetilde{\mathsf{W}(c)}/\mathrm{R}^*$.

  An operation τ may then belong to two distinct classes of $\widetilde{\mathsf{W}(c)}/\mathrm{R}^*$ and $\widetilde{\mathsf{W}(c')}/\mathrm{R}^*$, $c \neq c'$, that is, have several class representatives and be associated to different class labels. To avoid this, n could be defined as a

function of both τ and c. But constructing n is not easy then and left for future work.

To avoid this pitfall, we enforce the same labels for all classes including τ. To do this, we first label all classes of W/R^*, which in turn gives labels to the classes of all $\widetilde{\mathsf{W}(c)}/\mathrm{R}^*$. The drawback of this method is that some memory cells not used during program execution may be allocated. The reasons are that we cannot know statically which cells will be referred to, and that the set of numbers labeling the classes of a given $\widetilde{\mathsf{W}(c)}/\mathrm{R}^*$ may not be dense.

The maximal static expansion scheme given above works for any imperative programs. However, we give below an algorithm to construct expanded codes for loops nests and arrays only.

Before giving the algorithm, we would like to focus on two important points:

- The algebraic view given in this section considered each memory cell c in turn. Obviously, since the number of memory cells brought into play in a program is often unknown or parameterized, a naive application of this view would not practical. Our method gives a solution parameterized by the identity of the cell c, so its complexity does not grow with $\mathrm{card}(\mathsf{M})$.
- The definitions given in Section 5.4.3 and the expansion scheme are valid for any class of imperative programs. The only restrictions and limitations are those of the dataflow analysis and of the algorithm to compute transitive closures.

  In the sequel, since we use our own array dataflow analysis to apply the maximal static expansion framework, we inherit its syntactical restrictions: Data-structures are scalars and arrays; Pointers are not allowed. Loops and conditionals are unrestricted.

5.4.5 Algorithm

Using a dataflow analysis such as the one we presented in Chapter 4, the dataflow graph is described by systems of affine inequalities over iteration variables and structure parameters. Our algorithm then reduces to well known transformations on affine integer polyhedra, most of them being implemented in Omega [102]. We present below the expansion algorithm for all accesses to a given array `A`.

Input: The dataflow graph as an affine relation σ between reads and their reaching definitions (the sources).

Output: The target expanded code.

1. Compute $\mathrm{R} = \sigma \circ \sigma^{-1}$. (This boils down to eliminating ν in (5.4).)

2. If R is not transitive, compute R^* with Omega's transitive closure operator. Because the transitive closure of an affine relation is not necessarily affine, the result may be an upper-approximation. See [76] for details. This approximation is a conservative one, but may hide an interesting possible static expansion. Using Omega, R^* is described as a mapping from τ to $\widehat{\tau}$ ($\widehat{\tau}$ being the *class* of τ for relation R^*: $\widehat{\tau} = \{\zeta \in \mathsf{W} \,|\, \tau \mathrm{R}^* \zeta\}$).

3. In each class $\widehat{\tau}$, pick a single, arbitrary element. This chosen element is now considered as the representative $\varphi(\tau)$. How do we pick this element? As long as the element we pick is unique, any method is fine. We choose the minimum according to lexicographical order (which is a case of overkill).

4. Are all subscript functions affine?

   Yes Let us consider $c = \mathtt{A}(x)$. $\mathsf{W}(\mathtt{A}(x))$ is the union of

   $$\{\langle \mathsf{S}, i\rangle \,|\, i \in \mathsf{I}(\mathsf{S}) \wedge f_{\mathsf{S}}(\langle \mathsf{S}, i\rangle) = x\}$$

   over all statements S writing into A with subscript f_{S}.

   Compute $\varphi(\mathsf{W}(\mathtt{A}(x)))$, which is a set of representatives of the classes of $\mathsf{W}(\mathtt{A}(x))/\mathrm{R}^*$. Give a number to each element in the set of representatives.

   No Compute $\varphi(\mathsf{W})$. Give a number to each element in the set of representatives.

   If an element in the set of representatives is itself a parameterized affine set of operations, labeling boils down to scanning exactly once all the integer points in the set, which can be done using classical techniques [5, 31].

   In both cases, τ has a single representative and is therefore mapped to a unique label $n(\tau)$.

5. Code generation is then straightforward: Any reference $\mathtt{A}(f_{\mathsf{S}}(\tau))$ in the left hand side of S is transformed into $\mathtt{A}(f_{\mathsf{S}}(\tau), n(\tau))$. For any reference in the right hand side, one has to find the label of the source of the read. That is, any read $\mathtt{A}(g_{\mathsf{S}}(\tau))$ is transformed into $\mathtt{A}(g_{\mathsf{S}}(\tau), n(\sigma(\tau)))$. (Recall that $\sigma(\tau)$ is a set which is mapped, by construction of n, to a *single* label $n(\sigma(\tau))$.)

   When n is a conditional whose predicate is affine w.r.t. loop counters, then the conditional can be taken out of A's subscript.

6. The size declaration `A(...)` of `A` is transformed into the following declaration: `A`$(\ldots, \max_{\mathtt{S}} \max_{x\in \mathsf{I}(\mathtt{S})} n(\langle \mathtt{S}, x\rangle))^2$.

Computing the Lexicographical Minimum Let us call $\widehat{\tau}$ the equivalence class of τ for relation R^*. The lexicographical minimum of $\widehat{u}$ is:

$$\min(\widehat{\tau}) = \zeta \text{ s.t. } \tau \mathrm{R}^* \zeta \wedge \ (\nexists \nu, \tau \mathrm{R}^* \nu \wedge \nu \prec \zeta)$$

This definition may be simplified by writing $\prec$ as a relation between operations:

$$\prec = \{(\tau, \zeta) \,|\tau \prec \zeta\} .$$

Thus,

$$\min(\widehat{\tau}) = \big(\mathrm{R}^* \setminus (\prec \circ \mathrm{R}^*)\big)(\tau) \tag{5.7}$$

Complexity For each array in the source program, the algorithm proceeds as follows:

- Compute the reciprocal relation σ^{-1} of σ. This is different from computing the inverse of a function and barely consists in a swap of the two arguments of σ.
- Composing two relations σ and σ' boils down to eliminating y in $x\sigma y \wedge y\sigma' z$.
- Computing the exact transitive closure of R is quite expensive. Kelly et al. [76] do not give a formal bound on the complexity of their algorithm, but their implementation in the Omega toolset proved to be efficient if not concise. Notice again that the exact transitive closure is *not* necessary for our expansion scheme to be correct.

  Moreover, R happens to be often transitive in practice. Of course, this can be checked before triggering the computation of the closure: One just has to check that the difference $(\mathrm{R}\circ\mathrm{R}) \setminus \mathrm{R}$ is empty.
- In the algorithm above, φ is a lexicographic minimum. This is uselessly expensive since the expansion scheme just needs a way to pick one element per equivalence class. Computing the lexicographical minimum is expensive but was easy to implement in our first prototype.
- Finally, numbering classes is costly only when we have to scan a polyhedral set of representatives in dimension greater than 1. In practice, we only had intervals in the examples we tried.

---

[2]Because arrays usually have to be declared rectangular, $\mathtt{A}_{n(\tau)}(f(\tau))$ may be a better renaming. Consider for instance the expanded version of Example 2: Expanding `A` into `A1` and `A2` would require $6N-2$ array elements instead of $8N-2$ in Figure 5.5

Implementation The maximal static expansion has been implemented in C++ by Cohen and Collard on top of the Omega library and of Caravan. Figure 5.7 summarizes the computation times for the three examples (on a Sun SPARCstation 5). These results do not include the computation times for dataflow analysis and code generation.

	1st example	2nd example	3rd example
transitive closure (check)	100	100	110
picking the representatives (function φ)	110	160	110
other	130	150	70
total	340	410	290

Figure 5.7: Computation times, in milliseconds.

Moreover, computing the class representatives is relatively fast; It validates our choice to compute function φ (mapping operations to their representatives) using a lexicographic minimum. The intuition behind these results is that the computation time mainly depends on the number of affine constraints in the dataflow analysis relation.

Our only concern so far would be to find a way to approximate the expressions of transitive closures when they become large.

5.4.6 Examples

This section applies our algorithm of the previous section to the motivating examples, using the Omega Calculator [102] as a tool to manipulate affine relations.

First Example

Let us consider the source program of Example E1, presented in Section 4.2.2. Using the Omega Calculator text-based interface, we describe a step-by-step execution of the expansion algorithm. We have to code operations as integer-valued vectors. An operation $\langle \mathtt{S}_l, x\rangle$ is denoted by vector $[x, .., l]$, where $[..]$ possibly pads the vector with zeroes. We number $\mathtt{S}_1, \mathtt{S}_2, \mathtt{R}$ with 1, 2, 3 in this order, so $\langle \mathtt{S}_1, x_1\rangle$, $\langle \mathtt{S}_2, x_1, x_2\rangle$ and $\langle \mathtt{R}, x_1\rangle$ are written $[x_1, 0, 1], [x_1, x_2, 2]$ and $[x_1, 0, 3]$ respectively. The execution order, $\prec$, then becomes the lexicographical order. Note that in general, transforming the execution order into a lexicographical order is always possible but requires a more complex transformation on the iteration vector. See the work of Kodukula and Pingali [77] for a general way

to encode the execution order as a lexicographic order. However, our transformation is correct for the examples we consider.

From the source functions of `s`, we construct the source relation `S`:

```
# S := {[x1,0,2]->[x1,0,1] : 1<=x1<=N}
#      union {[x1,x2,2]->[x1,x2-1,2] : 1<=x1<=N && 2<=x2}
#      union {[x1,0,3]->[x1,0,1] : 1<=x1<=N}
#      union {[x1,0,3]->[x1,x2,2] : 1<=x1<=N && 1<=x2};
```

Step 1. Computing R is straightforward:

```
# S' := inverse S;
# R := S(S');
# R;

{[x1,0,1]->[x1,0,1] : 1<=x1<=N} union
{[x1,x2,2]->[x1,0,1] : 1<=x1<=N && 1<=x2} union
{[x1,0,1]->[x1,x2',2] : 1<=x1<=N && 1<=x2'} union
{[x1,x2,2]->[x1,x2',2] : 1<=x1<=N && 1<=x2' && 1<=x2}
```

Step 2. R is already transitive, no closure computation is thus necessary.

Step 3. Let us choose $\varphi(\tau)$ as the first executed operation in class $\widehat{\tau}$ (the least operation according to the sequential order): $\varphi(\tau) = \min\{\zeta \,|\zeta \mathrm{R}^* \tau\}$.

To compute the lexicographical minimum, let us rewrite its definition using the Omega Calculator syntax. We thus describe φ by a relation of the form:

$$\begin{aligned}\varphi([x_1,x_2,l]) \quad = \quad & [x_1',x_2',l'] \text{ s.t.} \\ & [x_1,x_2,l],[x_1',x_2',l'] \in \mathsf{W}(\mathtt{s}) \\ & \wedge [x_1,x_2,l]\mathrm{R}^*[x_1',x_2',l'] \\ & \wedge\ (\nexists [x_1'',x_2'',l''] \in \mathsf{W}(\mathtt{s})\ : \\ & \qquad [x_1,x_2,l]\mathrm{R}^*[x_1'',x_2'',l''] \\ & \qquad \wedge [x_1',x_2',l'] \ll [x_1'',x_2'',l''])\end{aligned}$$

Since $[x_1,x_2,l] \in \mathsf{W}(\mathtt{s})$ is always verified in this example, we may simplify this expression in using (5.7):

$$\varphi([x_1,x_2,l]) = \big(\mathrm{R}^* \setminus (\ll \circ \mathrm{R}^*)\big)([x_1,x_2,l]). \tag{5.8}$$

The result of this computation is:

$$\begin{aligned}\forall x_1,x_2, 1 \le x_1 \le N, x_2 \ge 1 \quad : \quad & \varphi(\langle \mathsf{S}_1, x_1\rangle) = \langle \mathsf{S}_1, x_1\rangle\,, \\ & \varphi(\langle \mathsf{S}_2, x_1, x_2\rangle) = \langle \mathsf{S}_1, x_1\rangle\,.\end{aligned}$$

Step 4. Since we have only one memory cell, $\mathsf{W}(\mathtt{s}) = \mathsf{W}$. Computing $\varphi(\mathsf{W}(\mathtt{s}))$ yields N operations of the form $\langle \mathtt{S}_1, x_1 \rangle$. Maximal static expansion of accesses to variable `s` requires N memory cells. x_1 is an obvious label:

$$\forall x_1, x_2,\ 1 \le x_1 \le N, x_2 \ge 1\ :\ n(\langle \mathtt{S}, x_1, x_2 \rangle) = n(\langle \mathtt{S}_1, x_1 \rangle) = x_1. \tag{5.9}$$

Step 5. All left-hand side references to `s` are transformed into $\mathtt{s}(x_1)$; All references to `s` in the right hand side are transformed into $\mathtt{s}(x_1)$ too since their sources are instances of $\mathtt{S}_1$ or $\mathtt{S}_2$ for the same x_1. The expanded code is thus exactly the one found intuitively in Figure 5.2.

Step 6. The size declaration of the new array is $\mathtt{s}(1..N)$.

Second Example

We now consider the source program in Figure 5.3. Operations $\langle \mathtt{S}, i, j \rangle$ and $\langle \mathtt{T}, i, N \rangle$ are denoted by $[i, j, 1]$ and $[i, N, 2]$ respectively. From (5.2), the source relation `S` is defined as:

```
# S := {[i,j,1]->[i', j',1] : 1<=i,i'<=2N
#       && 1<=j'<j<=N && i'-j'=i-j}
#      union {[i,j,1]->[i',N,2] : 1<=i,i'<=2N
#             && N<j<=2N && i'=i-j+N}
#      union {[i,j,1]->[i',j',1] : 1<=i,i'<=2N
#             && N<j'<j<=2N && i'-j'=i-j};
```

Step 1. As in the first example, we compute relation R using Omega:

```
# S' := inverse S;
# R := S(S');
# R;

{[i,j,1]->[i',j-i+i',1] : 1<=i<=2N-1 && 1<=j<N
 && 1<=i'<=2N-1 && i<j+i' && j+i'<N+i} union
{[i,j,1]->[i',j-i+i',1] : N<j<=2N-1 && 1<=i<=2N-1
 && 1<=i'<=2N-1 && N+i<j+i' && j+i'<2N+i} union
{[i,N,2]->[i',N-i+i',1] : 1<=i<i'<=2N-1
 && i'<N+i} union
{[i,j,1]->[N+i-j,N,2] : N<j<=2N-1 && i<=2N-1
 && j<N+i} union
{[i,N,2]->[i,N,2] : 1<=i<=2N-1}
```

Step 2. We compute R^*. Figure 5.4.a shows the equivalence classes of R^*.

Step 3. We compute $\varphi(\tau)$ as a relation similar to (5.8), using Omega. The result follows:

$$\begin{array}{l}
\forall i,j,\ 1 \le i \le 2N, 1 \le j \le N, j-i \ge 0 \\
\qquad \varphi(\langle \mathtt{S}, i, j\rangle) = \langle \mathtt{S}, 1, j-i+1\rangle \\
\forall i,j,\ 1 \le i \le 2N, 1 \le j \le N, j-i < 0 \\
\qquad \varphi(\langle \mathtt{S}, i, j\rangle) = \langle \mathtt{S}, i-j+1, 1\rangle \\
\forall i,j,\ 1 \le i \le 2N, N+1 \le j \le 2N, j-i \ge N \\
\qquad \varphi(\langle \mathtt{S}, i, j\rangle) = \langle \mathtt{S}, 1, j-i+1\rangle \\
\forall i,j,\ 1 \le i \le 2N, N+1 \le j \le 2N, j-i < N \\
\qquad \varphi(\langle \mathtt{S}, i, j\rangle) = \langle \mathtt{T}, i-j+N, N\rangle \\
\forall i,j,\ 1 \le i \le 2N \\
\qquad \varphi(\langle \mathtt{T}, i, N\rangle) = \langle \mathtt{T}, i, N\rangle
\end{array}$$

Step 4. $\mathsf{W}(c) = \{\langle \mathtt{S}, i, j\rangle \,|\, i-j+2N = c\} \cup \{\langle \mathtt{T}, c-N, N\rangle\}$.
Let us compute the representatives for $\mathsf{W}(c)$:

$$\begin{array}{rcl}
1 \le c \le N & : & \varphi(\mathsf{W}(c)) = \langle \mathtt{S}, 1, 1-c\rangle \\
N+1 \le c \le 3N-1 & : & \varphi(\mathsf{W}(c)) = \{\, \langle \mathtt{S}, c+1, 1\rangle, \\
 & & \qquad\qquad\quad \langle \mathtt{T}, c+N, N\rangle \,\} \\
3N \le c \le 4N-1 & : & \varphi(\mathsf{W}(c)) = \langle \mathtt{S}, c+1, 1\rangle
\end{array}$$

This result shows three intervals of constant cardinality of $\mathsf{W}(c)/\mathrm{R}^*$; They are described in Figure 5.4.b. A labeling can be found mechanically. If $i-j+2N \le N$ or $i-j+2N \ge 3N$, there is only one representative in $\varphi(\mathsf{W}(i-j+2N))$, thus $n(\langle \mathtt{S}, i, j\rangle) = 1$. If $N+1 \le i-j+2N \le 3N-1$, there are two representatives; Thus we define $n(\langle \mathtt{S}, i, j\rangle) = 1$ if $j \le n$, $n(\langle \mathtt{S}, i, j\rangle) = 2$ if $j > N$, and $n(\langle \mathtt{T}, i, N\rangle) = 2$.

Step 5. The static expansion code appears in Figure 5.5. As hinted in Section 5.4.5, conditionals in n have been taken out of array subscripts.

Step 6. Array `A` is allocated as `A`$(1..4*N-1, 1..2)$.

Third Example: Non-Affine Array Subscripts

We come back to the program in Figure 5.6.a. Operations $\langle \mathtt{T}, i, j\rangle$, $\langle \mathtt{S}, i\rangle$ and $\langle \mathtt{R}, i\rangle$ are written $[i,j,1], [i,0,2]$ and $[i,0,3]$. From (5.3), we build the source relation as follows:

```
# S := {[i,0,3]->[i,j,1] : 1<=i,j<=N}
#      union {[i,0,3]->[i,0,2] : 1<=i<=N};
```

Step 1.

```
# S' := inverse S;
# R := S(S');
# R;

{[i,j,1]->[i,j',1] : 1<=i<=N && 1<=j<=N
 && 1<=j'<=N} union
{[i,0,2]->[i,j',1] : 1<=i<=N && 1<=j'<=N} union
{[i,j,1]->[i,0,2] : 1<=i<=N && 1<=j<=N} union
{[i,0,2]->[i,0,2] : 1<=i<=N}
```

Step 2. R is already transitive: $\mathrm{R} = \mathrm{R}^*$.

Step 3. We compute $\varphi(\tau)$ as a relation similar to (5.8).

$$\forall i,\ 1 \le i \le N\ :\ \varphi(\langle \mathtt{S}, i\rangle) = \langle \mathtt{T}, i, 1\rangle$$
$$\forall i, j,\ 1 \le i \le N, 1 \le j \le N\ :\ \varphi(\langle \mathtt{T}, i, j\rangle) = \langle \mathtt{T}, i, 1\rangle$$

Note that every $\langle \mathtt{T}, i, j\rangle$ operation is in relation with $\langle \mathtt{T}, i, 1\rangle$.

Step 4. Since some subscripts are not affine, we cannot compute at compile-time the exact sets $\mathsf{W}(\mathtt{A}(x))$ of operations writing in some cell $\mathtt{A}(x)$. Therefore, we compute $\varphi(\mathsf{W})$:

$$\varphi(\mathsf{W}) = \{\, \langle \mathtt{T}, i, 1\rangle \,\}.$$

We can use i to label these representatives; Thus the resulting n function is:

$$n(\langle \mathtt{S}, i\rangle) = n(\langle \mathtt{T}, i, j\rangle) = i.$$

Step 5. Using this labeling, all left hand side references to $\mathtt{A}(...)$ become $\mathtt{A}(..., i)$ in the expanded code. Since the source of $\langle \mathtt{R}, i\rangle$ is an instance of `S` or `T` at the same iteration i, the right hand side of `R` is expanded the same way. Expanding the code thus leads to the intuitive result given at Figure 5.6.b.

Step 6. The size declaration of `A` is now $\mathtt{A}(1..N, 1..N)$.

5.4.7 Conclusion

Expanding data structures is a classical optimization to cut memory-based dependences. However, the generated code has to ensure that all reads refer to the correct memory cell. When control flow is dynamic, the main drawback of such methods is therefore that some run-time computation has to be done to decide the identity of the correct memory cell.

We presented a new and general expansion framework: A cell can be expanded at most as many times as there are classes of independent (as far as dataflow is concerned) writes. A practical algorithm was given and applied to real-life loop nests accessing arrays.

Interestingly enough, the framework does not require any precision level of the dataflow analysis, nor does it require the closure computation to be exact. Conservative approximate results are fine as well, the only drawback being a probable loss in static expansion. However, we cannot do any better, and in accordance to our definition, the static expansion we derive is still maximal. When the dataflow analysis and/or the transitive closure tool give poor results, our expansion scheme does not fail but degrades gracefully.

Future work will study the application of the framework to a wider class of problems. We also intend to enhance the algorithm so as to handle pointer-based data structures and recursive programs.

5.5 Scheduling

A schedule assigns an execution date to each operation of the program. In general, this date is logical and all operations are supposed to take unit time [51]. Operations which are to be executed at the same time can be executed in parallel. Basic scheduling algorithms are presented in [53, 54]. As we will explain here, there is no difficulty in extending these algorithms to the case where the dataflow analysis has given fuzzy results.

Dependent operations must be executed sequentially and other operations can be executed concurrently. If anti- and output-dependences can be removed (by array expansion or array privatization, for instance), then the maximum degree of parallelism can be exposed by taking into account only value-based dependences: an operation can be executed only after all its sources.

A schedule θ must then verify in the case of an exact analysis:

$$\theta(\langle \mathtt{R}, y\rangle) \geq \theta(\sigma(\mathtt{A}, \langle \mathtt{R}, y\rangle)) + 1,$$

for any variable A read in $\langle \mathtt{R}, y\rangle$. In the case of a fuzzy analysis, all the possible sources of a read must be executed before the read:

$$\forall \tau \in \mathsf{S}(\mathtt{A}, \langle \mathtt{R}, y\rangle) \; : \; \theta(\langle \mathtt{R}, y\rangle) \geq \theta(\tau) + 1. \qquad (5.10)$$

In the result of the corresponding FADA, τ depends on a fuzzy parameter x_τ and occurs in a leaf of a quast governed by an affine predicate $r_\tau(y, x_\tau)$: $\tau = \zeta_\tau(y, x_\tau)$ where ζ is affine. We may refine (5.10) into:

$$\forall x_\tau \; : \; r_\tau(y, x_\tau) \Longrightarrow \theta(\langle \mathtt{R}, y\rangle) \geq \theta(\zeta(y, x_\tau)) + 1. \qquad (5.11)$$

Suppose we have modeled the schedule θ as an affine function with unknown coefficients. Since everything is affine in (5.11) we are in a position to apply

Farkas theorem. The result is a set of linear equations in the coefficients of the schedule and new positive unknowns, the *Farkas multipliers*. These equations may be solved as in [53, 54]. Note that the same technique can be used so as to handle quasi-affine constraints: Intuitively, quasi-affine constraints can be considered as non-affine constraints that are described exactly by affine relations on a fuzzy parameter (the quotient of the Euclidean division). More formally, any expression $\left\lfloor \frac{f}{n} \right\rfloor$ with n an integer constant can be replaced by a new variable q defined by the predicate $r(f, q) = (0 \leq f - nq < n)$.

As shown in the previous section, it may not be possible to remove at compile-time all anti- and output-dependences when the value-based analysis is approximate. In such a case, these dependences must also be taken into account in the computation of the schedule.

It may be interesting in some cases to compute the schedule from the expression of the exact parametric sources. These sources still depend on the unknown parameters of the maximum, and some relations between non-affine constraints can entail a decrease in the number of constraints on the scheduling functions. Consider for instance the following code:

```
    do i = 1, n
S1   if .. then x = ..
    enddo
S2  if .. then x = .. else y = ..
R   s = x + y
```

We suppose that nothing is known about the predicates of the conditionals of $\mathtt{S}_1$ and $\mathtt{S}_2$. Therefore the sources of `x` and `y` are very fuzzy: the source set of `x` for operation $\langle \mathtt{R}, [] \rangle$ contains $-$ and all the previous operations of $\mathtt{S}_1$ and $\mathtt{S}_2$. When building the schedule function, we impose that $\langle \mathtt{R}, [] \rangle$ is scheduled after all operations of $\mathtt{S}_1$, and $\mathtt{S}_2$ (preserving dataflow dependences) and that the sequential order of execution is preserved between the different possible writes of `x` (preserving output dependences). As a matter of fact, this system of constraints can be simplified since it can be proved that $\langle \mathtt{R}, [] \rangle$ always depend on $\langle \mathtt{S}_2, [] \rangle$. This exact dependence between the two operations is obtained by simply introducing the structural property of the `if..then..else`. To preserve dataflow dependences, it is sufficient to impose that $\langle \mathtt{R}, [] \rangle$ is executed after $\langle \mathtt{S}_2, [] \rangle$. Hence, this kind of optimization decreases the complexity of the computation of scheduling functions.

The problem of *executing* the resulting parallel program depends on whether speculation has been allowed or not. This problem is beyond the scope of our work. The reader is referred to [28, 55] for speculative execution due to inaccurate dataflow analysis or dynamic control.

5.6 Conclusion

Code transformations, scheduling and detection of recurrences can be easily adapted to the results produced by an approximate analysis.

As for array expansion, this is a classical optimization to cut memory-based dependences and the generated code has to ensure that all reads refer to the correct memory cell. When the dataflow analysis is approximate, the main drawback of such methods is therefore that some run-time computation has to be done to decide the identity of the correct memory cell.

We presented a new and general expansion framework: A cell can be expanded at most as many times as there are classes of independent (as far as data-flow is concerned) writes. A practical algorithm was given and applied to real-life loop nests accessing arrays.

Interestingly enough, the framework does not require any precision level of the dataflow analysis, nor does it require the closure computation to be exact. Conservative approximate results are fine as well, the only drawback being a probable loss in static expansion. However, we cannot do any better, and in accordance to our definition, the static expansion we derive is still maximal. When the dataflow analysis and/or the transitive closure tool give poor results, our expansion scheme does not fail but degrades gracefully.

Most of the work in this chapter has been presented in [11] and the maximal static expansion has been described in [10].

Chapter 6

Conclusion

6.1 Contributions

Exact array dataflow analyses describe at the operation level the flow of data between the variables of a program. The advantage of these analyses is that they enable, besides other possible applications in reengineering or debugging, very aggressive parallelizing techniques on the program at compile time. However, their scope is more or less limited to static control programs, thus they can only be used on small fragments of real code. Moreover, their worst case complexity is high, which further restricts the range of programs that can be efficiently handled.

In this thesis, we have proposed methods to cope with both of these drawbacks: We have described an algorithm that computes very efficiently, with at worst polynomial time and space complexity, the dataflow graphs of a subset of static control programs, and we have presented a general framework for approximate array dataflow analysis of programs with `while` loops, `if..then..else` constructs with any predicate, any kind of array subscripts and any kind of loop bounds. Moreover, the technique involved for the computation of the approximate dataflow graph can reuse, without any change, the optimizations of exact array analysis, as well as our polynomial algorithm. This algorithm can then handle very efficiently a subset of programs with non-affine constraints.

Besides, we have shown that the accuracy of our analysis can be improved by complementary analyses that find properties on non-affine constraints, such as the numerous existing symbolic analyses. In addition to these techniques, we have proposed an iterative analysis that reuses the results of a preceding dataflow analysis of a variable so as to improve the accuracy of another one. The method consists in finding relations (mostly equality) between the values of the same variable used at two different points of the program. The computation of the dataflow graph is achieved in two steps, the first one consists in integrating the properties of non-affine constraints in the framework of the computation, and the second, independent of the first, is the computation itself.

Integrating the properties of non-affine constraints consists in transforming a system of logical clauses in which non-affine constraints appear into a system of clauses in which they no longer appear. This is done by applying repeatedly a resolution rule on the initial system of clauses. In general, an approximate analysis cannot produce in finite time the most accurate result with respect to the properties on non-affine constraints, due to decidability reasons. However, we have established the condition for which this optimal accuracy is reached. A prototype, although based on a precedent version of our formalism, validates the approach chosen for the approximate analysis.

Our approach to deal with non-affine constraints is not dependent of the dependence representation used and by comparing our analysis with Wonnacott's work [129] and with Creusillet's array region formalism [34], we have shown that approximate array analysis still provides for programs in its scope more accurate results than other kind of analyses, thus allowing more aggressive parallelization techniques.

Most traditional applications of exact array dataflow analyses only need a slight adaptation to be able to handle the results of an approximate analysis. This is not the case of array expansion, and we have presented an original method to expand array structures without dynamic restoration of the data flow. Basically, array elements which are defined by operations that can be sources of the same operation under the same conditions are not expanded.

6.2 Future Work

Several developments of the approximate dataflow analysis in itself can be considered. First of all, there are a number of techniques we described in this thesis concerning the computation of sources that can be improved:

- Resolution methods: We only described the general resolution rule and then an input linear resolution method. However, there exists many more kinds of resolutions, each of them is adapted to a particular form of clauses (for instance SLD resolution for Horn clauses, which is used by Prolog). There still remains a lot of work so as to adapt both the form of the relations on non-affine constraints and the kind of resolution method used to each other (as input linear resolution is adapted to the properties of structural analysis for example).

- Computation of the context quasts: The rules to build context quasts described in Section 4.6.1 can be optimized, as hinted at Section 4.6.3.

- Extension of the approximate analysis to programs with procedure calls: Leservot [82] has proposed an extension of the polyhedral method to these programs. The adaptation of his method to the approximate analysis seems to be possible, but we have not considered it yet.

On a more theoretical ground, with the exception of the iterative analysis, there does not exists any feed-back between the analyses that discover relations on non-affine constraints and the approximate array dataflow analysis. We have shown that it is possible to determine whether a relation improves the accuracy of the source, or whether a relation cannot be translated without adding fuzziness in a relation on the parameters of the maximum. These two criteria are not really used up to now, but should help in deciding whether it is interesting to find more properties about a non-affine constraint. This idea of feed-back however is relevant only if the symbolic analysis can refine its result progressively, or integrate the results of a more and more precise dataflow analysis. This is only possible so far with the iterative analysis.

Besides, we think that an important work of optimization of the analysis can be done in the choice of the representation of dataflow dependences. Following the idea presented in Section 4.8.4, the representation of dataflow dependences could be degraded during the analysis, either for efficiency reasons, at the cost of accuracy, or because there is no reason in keeping a precise description of the dataflow. The first case can be justified on large codes, because for instance the probability that a write is the source of a read decreases with the distance between the statements. This idea to partition the code into different fragments to be analyzed separately has already been tackled by Berthou [15]. The second reason can be motivated by the degree of parallelism that is desired. For instance, array privatization only need loop independence information, and for interprocedural analysis, regions abstract the effects of the procedures on array elements (See the thesis of Leservot [82] for such adaptation of the exact array dataflow analysis to interprocedural analysis). More generally, the accuracy of the analysis, either controlled by its representation or by the properties on non-affine constraints that have been found, should depend on the application. This seems obvious to say and most dependence analyses and tests that have been developed so far were adapted to one particular kind of application but they could not change easily the accuracy of their result. One last consideration about the representation of dataflow dependences: the approximate analysis we presented tightly depends on exact array dataflow analyses. Non-affine constraints are “eliminated” in order to apply the techniques used in the exact case. Whether our framework can be adapted to recursive programs is uncertain since there does not exists in this case an “exact” analysis.

Finally, we have not adapted some applications of exact array dataflow analyses. This is the case of the memory optimization technique described by Lefebvre [80, 81]. How to relate this optimization with the static memory expansion presented in Section 5.4 is still unknown.

Personnal Bibliography

- D. Barthou, A. Cohen and J.-F. Collard.*Maximal Static Expansion.* ACM Symp. on Principles of Programming Languages. pp. 98–106,January 1998.
- C. Ancourt, D. Barthou, C. Guettier, F. Irigoin, B. Jeannet, J. Jourdan and J. Mattioli. *Automatic Mapping of Signal Processing Applications onto Parallel Computers.* Procs. of IEEE Int. Conf. on Application-specific Systems, Architectures and Processors, pp. 350–362, Zurich, July 1997.
- D. Barthou and J.-F. Collard and P. Feautrier. *Fuzzy Array Dataflow Analysis.* J. of Parallel and Distributed Computing, vol. 40, Number 2, pp. 210–226, February 1997.
- D. Barthou and J.-F. Collard and P. Feautrier. *Applications of Fuzzy Array Dataflow Analysis.*Second International Euro-Par Conference, Lecture Notes in Computer Science, vol. 1123, pp. 424–427, 1996.
- J.-F. Collard and D. Barthou and P. Feautrier. *Fuzzy Array Dataflow Analysis.* Fifth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, Vol. 30, Number 8, pp. 92–101, San Diego, CA, August 1995.
- D. Barthou and F. Gasperoni and U. Schwiegelshohn. *Allocating Communication Channels to Parallel Tasks.* In J.J. Dongarra and B. Tourancheau eds, Environments and Tools for Parallel Scientific Computing, Elsevier Science Publishers, 1993.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass, 1986.

[2] J. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, October 1987.

[3] S. Amarasinghe, J. Anderson, M. Lam, and C.-W. Tseng. An overview of the SUIF compiler for scalable parallel machines. In *SIAM Conf. on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.

[4] Z. Ammarguellat. *Restructuration des programmes Fortran en vue de leur parallélisation.* PhD thesis, Université P. et M. Curie, Paris 6, December 1988.

[5] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, volume 26(7), pages 39–50, Williamsburg, VA, April 1991.

[6] R. Ballance, A. Maccabe, and K. Ottenstein. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 257–271, White Plains, NY, June 1990.

[7] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston / Dordrecht / London, 1988.

[8] H. Barendregt. *The Lambda Calculus. Its Syntax and Semantics.* North-Holland, 1981.

[9] M. Barreteau and P. Feautrier. Efficient mapping of interdependent scans. In *Euro-Par*, volume 1123 of *Lect. Notes in Computer Science*, pages 463–467, Lyon, France, August 1996.

[10] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. In *ACM Symp. on Principles of Programming Languages*, pages 98–106, San Diego, CA, January 1998.

[11] D. Barthou, J.-F. Collard, and P. Feautrier. Applications of fuzzy array dataflow analysis. In *Euro-Par*, volume 1123 of *Lect. Notes in Computer Science*, pages 424–428, Lyon, France, August 1996.

[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] A. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. on El. Computers*, EC-15:757–762, 1966.

[14] M. Berry et al. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. *Int. J. of Supercomputer Applications*, 3:5–40, March 1989.

[15] J.-Y. Berthou. *Construction d'un paralléliseur de logiciels scientifiques de grande taille guidée par des mesures de performances.* PhD thesis, Université P. et M. Curie, Paris 6, October 1993.

[16] W. Blume. Success and limitations in automatic parallelization of the Perfect Benchmarks programs. Technical Report CSRD 1249, U. of Illinois at Urbana-Champaign, July 1992.

[17] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Trans. on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[18] W. Blume and R. Eigenmann. An overview of symbolic analysis techniques needed for the effective parallelization of the perfect benchmarks. In *Int. Conf. on Parallel Processing*, pages 233–238, Lanham, MD, August 1994.

[19] W. Blume and R. Eigenmann. Symbolic range propagation. In *9th Int. Symp. on Parallel Processing*, pages 357–363, Los Alamitos, CA, April 1995.

[20] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The next generation in parallelizing compilers. In *Int. Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[21] P. Boulet. Bouclettes: A Fortran loop parallelizer. In *High Performance Computing and Networking*, volume 1067 of *Lect. Notes in Computer Science*, pages 784–791, Brussels, April 1996.

[22] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? In *Scalable High-Performance Computing Conf.*, pages 568–576, Knoxville, Tenn., May 1994. IEEE Computer Society Press.

[23] T. Brandes. The importance of direct dependences for automatic parallelization. In *ACM Int. Conf. on Supercomputing*, pages 407–417, St. Malo, France, July 1988.

[24] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *ACM Int. Conf. on Supercomputing*, St. Petersburg, Florida, 1987.

[25] P.-Y. Calland, A. Darte, Y. Robert, and F. Vivien. Plugging anti and output dependence removal techniques into loop parallelization algorithms. *Parallel Computing*, 23:251–266, April 1997.

[26] Z. Chamski. *Environnement logiciel de programmation d'un accélérateur de calcul parallèle*. PhD thesis, Université de Rennes, 1993.

[27] Z. Chamski. Nested loop sequences: Towards efficient loop structures in automatic parallelization. In H. El-Rewini and B. Shriver, editors, *27th Annual Hawaii Int. Conf. on System Sciences*, volume 2, pages 14–22, Hawaii, January 1994.

[28] J.-F. Collard. Automatic parallelization of while-loops using speculative execution. *Int. J. of Parallel Programming*, 23(2):191–219, April 1995.

[29] J.-F. Collard. *Parallélisation automatique des programmes à contrôle dynamique*. PhD thesis, Université P. et M. Curie, Paris 6, 1995.

[30] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In J. Ferrante and D. Padua, editors, *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, volume 30(8), pages 92–101, Santa Barbara, CA, August 1995.

[31] J.-F. Collard, P. Feautrier, and T. Risset. Construction of `do` loops from systems of affine constraints. *Parallel Processing Letters*, 3(5), 1995.

[32] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symp. on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977.

[33] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM Symp. on Principles of Programming Languages*, San Diego, CA, 1978.

[34] B. Creusillet. *Array Region Analyses and Applications*. PhD thesis, École Nationale Supérieure des Mines de Paris, December 1996.

[35] B. Creusillet and F. Irigoin. Exact versus approximate array region analyses. In *Int. Workshop on Languages and Compilers for Parallel Computing*, volume 1239 of *Lect. Notes in Computer Science*, pages 86–93, San Jose, CA, August 1996.

[36] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form. In *ACM Symp. on Principles of Programming Languages*, pages 25–35, Austin, Texas, January 1989.

[37] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[38] A. Darte. *Techniques de parallélisation automatique de nids de boucles*. PhD thesis, École Normale Supérieure de Lyon, Lyon 1, 1993.

[39] A. Darte, L. Khachiyan, and Y. Robert. Linear scheduling is nearly optimal. *Parallel Processing Letters*, 1(2):73–81, 1991.

[40] A. Darte and Y. Robert. Mapping uniform loop nests onto distributed memory architectures. *Parallel Processing Letters*, 20:679–710, 1994.

[41] A. Darte and Y. Robert. Affine-by-statement scheduling of uniform and affine loop nests over parametric domains. *J. of Parallel and Distributed Computing*, 29(1):43–59, August 1995.

[42] A. Darte and F. Vivien. On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. In *Euro-Par*, volume 1123 of *Lect. Notes in Computer Science*, pages 379–388, Lyon, France, August 1996.

[43] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. Technical Report RR96-06, Laboratoire de l'Informatique du Parallélisme, 1996.

[44] P. David. *Analyse sémantique du langage C en vue de sa parallélisation*. PhD thesis, Université P. et M. Curie, Paris 6, September 1991.

[45] P. Downey. Undecidability of Presburger arithmetic with a single monadic predicate letter. Technical Report TR-18-72, Harvard University, Cambridge, 1972.

[46] E. Duesterwald, R. Gupta, and M.-L. Soffa. A practical data flow framework for array reference analysis and its use in optimization. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 68–77, Albuquerque, NM, June 1993.

[47] A. Dumay. *Traitement des indexations non linéaires en parallélisation automatique : une méthode de linéarisation contextuelle.* PhD thesis, Université P. et M. Curie, December 1992.

[48] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect-Benchmark programs. In *Int. Workshop on Languages and Compilers for Parallel Computing*, volume 589 of *Lect. Notes in Computer Science*, pages 65–83, Berlin, Germany, August 1992.

[49] P. Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing*, pages 429–441, St Malo, France, 1988.

[50] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[51] P. Feautrier. Asymptotically efficient algorithms for parallel architectures. In M. Cosnard and C. Girault, editors, *Decentralized systems*, pages 273–284, Lyon, France, December 1989. North-Holland.

[52] P. Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, February 1991.

[53] P. Feautrier. Some efficient solutions to the affine scheduling problem, part I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[54] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, December 1992.

[55] P. Feautrier. Basis of parallel speculative execution. In *Euro-Par*, volume 1300 of *Lect. Notes in Computer Science*, pages 3–14, Passau, Germany, August 1997.

[56] R. Floyd. Assigning meanings to programs. In J. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Symposia in Applied Mathematics*, pages 19–32. AMS, Providence, RI, 1967.

[57] High Performance Fortran Forum. *High Performance Fortran Language Specification*, November 1994. Version 1.1.

[58] P. Granger. *Analyses sémantiques de congruences.* PhD thesis, École Polytechnique, July 1991.

[59] E. Granston and A. Veidenbaum. Combining flow and dependence analyses to expose redundant array accesses. *Int. J. of Parallel Programming*, 23(5):423–470, October 1995.

[60] E. De Greef, F. Catthoor, and H. De Mann. Reducing storage size for static control programs mapped to parallel architectures. Presented at Dagstuhl seminar on loop parallelization, April 1996.

[61] M. Griebl and J.-F. Collard. Generation of synchronous code for automatic parallelization of `while` loops. In *Euro-Par*, volume 966 of *Lect. Notes in Computer Science*, pages 315–326, Stockholm, August 1995.

[62] M. Griebl and C. Lengauer. The loop parallelizer LooPo — announcement. *Lect. Notes in Computer Science*, 1239:603–607, 1997.

[63] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance portable implementation of the MPI message passing interface standard. Technical Report ANL/MCS-TM-213, Mathematic and Computer Science Division, Argonne National Laboratory, IL, 1996.

[64] T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software : Practice and Experience*, 20(2):133–155, February 1990.

[65] J. Gu, Z. Li, and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *ACM Int. Conf. on Supercomputing*, Barcelona, Spain, December 1995.

[66] M. Haghighat and C. Polychronopoulos. Symbolic analysis: A basis for parallelization, optimization, and scheduling of programs. In *Int. Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lect. Notes in Computer Science*, pages 567–585, Portland, OR, August 1993.

[67] M. Haghighat and C. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Trans. on Programming Languages and Systems*, 18(4):477–518, July 1996.

[68] M. Hall. *Managing interprocedural optimization.* PhD thesis, Rice University, Houston, Texas, April 1991.

[69] M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *ACM Int. Conf. on Supercomputing*, San Diego, CA, December 1995.

[70] C. Heckler and L. Thiele. Computing linear data dependences in nested loop programs. *Parallel Processing Letters*, 4(3):193–204, September 1994.

[71] M. Hind, M. Burke, P. Carini, and S. Midkiff. An empirical study of precise interprocedural array analysis. *Scientific Programming*, 3(3):255–271, 1994.

[72] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.-W. Tseng. An overview of the Fortran D programming system. In *Int. Workshop on Languages and Compilers for Parallel Computing*, volume 589 of *Lect. Notes in Computer Science*, Berlin, Germany, 1992.

[73] F. Irigoin, P. Jouvelot, and R. Triolet. Overview of the PIPS project. In P. Feautrier and F. Irigoin, editors, *Second Int. Workshop on Compilers for Parallel Computers*, pages 199–212, Paris, December 1990.

[74] F. Irigoin and R. Triolet. Computing dependence direction vectors and dependence cones with linear systems. Technical Report ENSMP-CAI-87-E94, École Nationale Supérieure des Mines de Paris, 1987.

[75] F. Irigoin and R. Triolet. Supernode partitioning. In *ACM Symp. on Principles of Programming Languages*, pages 319–328, San Diego, CA, January 1988.

[76] 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.

[77] I. Kodukula and K. Pingali. Transformations of imperfectly nested loops. In *ACM Int. Conf. on Supercomputing*, Pittsburgh, PA, November 1996.

[78] L. Lamport. The parallel execution of DO loops. *Communication of the ACM*, 17(2):83–93, February 1978.

[79] J. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE Trans. on Parallel and Distributed Systems*, 7(4):812–826, October 1993.

[80] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. Technical Report RR-97/08, Laboratoire PR*i*SM, Université of Versailles, France, March 1997. http://www.prism.uvsq.fr.

[81] V. Lefebvre and P. Feautrier. Storage management in parallel programs. In *5th Euromicro Workshop on Parallel and Distributed Processing*, pages 181–188, London, January 1997.

[82] A. Leservot. *Analyse interprocédurale du flot des données.* PhD thesis, Université P. et M. Curie, Paris 6, March 1996.

[83] D. Levine, D. Callahan, and J. Dongarra. A comparative study of automatic vectorizing compilers. *Parallel Computing*, 17:1223–1244, 1991.

[84] Z. Li and P.-C. Yew. Efficient interprocedural analysis and program parallelization and restructuring. *ACM SIGPLAN Notices*, 9(23):85–99, 1988.

[85] Z. Li, P.-C. Yew, and C.-Q. Zhu. Data dependence analysis on multidimensional array references. In *ACM Int. Conf. on Supercomputing*, pages 215–224, Crete, Greece, June 1989.

[86] D. Loveland. *Automated Theorem Proving: A logical Basis.* North Holland, Amsterdam, 1978.

[87] F. Masdupuy. Array operations abstraction using semantic analysis of trapezoid congruences. In *ACM Int. Conf. on Supercomputing*, pages 226–235, Washington, DC, July 1992.

[88] F. Masdupuy. *Array indices relational semantic analysis using rational cosets and trapezoids.* PhD thesis, École Polytechnique, December 1993.

[89] V. Maslov. Delinerization: an efficient way to break multiloop dependence equations. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, volume 27(7), pages 152–161, San Francisco, CA, June 1992.

[90] V. Maslov. Lazy array data-flow dependence analysis. In *ACM Symp. on Principles of Programming Languages*, pages 311–325, Portland, OR, 1994.

[91] V. Maslov and W. Pugh. Simplifying polynomial constraints over integers to make dependence analysis more precise. *Lect. Notes in Computer Science*, 854:737–744, 1994.

[92] D. Maydan. *Accurate analysis of Array References.* PhD thesis, Stanford University, October 1992.

[93] D. Maydan, S. Amarasinghe, and M. Lam. Array dataflow analysis and its use in array privatization. In *ACM Symp. on Principles of Programming Languages*, pages 2–15, Charleston, SC, January 1993.

[94] D. Maydan, J. Hennessy, and M. Lam. Effectiveness of data dependence analysis. *Int. J. of Parallel Programming*, 23(1):63–81, February 1995.

[95] M. Minoux. *Programmation Mathématique, théorie et algorithmes.* Dunod, Paris, 1983.

[96] *Objective Caml*, 1998. INRIA, Rocquencourt, France.

[97] Manuel de Référence de PAF. Groupe "Calcul Parallèle" du MASI, January 1990. Available on request from P. Feautrier.

[98] P. Petersen and D. Padua. Static and dynamic evaluation of data dependence analysis techniques. *IEEE Trans. on Parallel and Distributed Systems*, 7(11):1121–1132, November 1996.

[99] K. Pieper. *Parallelizing Compilers: Implementation and Effectiveness.* PhD thesis, Stanford University, Computer Systems, June 1993.

[100] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Int. Conference on Parallel Processing, Vol. 2: Software*, pages 39–48, Pennsylvania, August 1989.

[101] W. Pugh. Uniform techniques for loop optimization. In *ACM Int. Conf. on Supercomputing*, pages 341–352, Cologne, Germany, June 1991.

[102] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communication of the ACM*, 8:102–114, August 1992.

[103] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *Int. Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lect. Notes in Computer Science*, pages 546–566, Portland, OR, August 1993.

[104] W. Pugh and D. Wonnacott. Experiences with constraint-based array dependence analysis. In A. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, volume 874 of *Lect. Notes in Computer Science*, Rosario, Orcas Island, WA, May 1994.

[105] W. Pugh and D. Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Trans. on Programming Languages and Systems*, 16(4):1248–1278, July 1994.

[106] W. Pugh and D. Wonnacott. Going beyond integer programming with the Omega test to eliminate false data dependences. *IEEE Trans. on Parallel and Distributed Systems*, 6(2):204–211, February 1995.

[107] W. Pugh and D. Wonnacott. Nonlinear array dependence analysis. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, volume 768 of *Lect. Notes in Computer Science*, Troy, NY, May 1995.

[108] L. Rauchwerger and D. Padua. The PRIVATIZING DOALL test: A run-time technique for DOALL loop identification and array privatization. In *ACM Int. Conf. on Supercomputing*, pages 33–43, Manchester, UK, July 1994.

[109] L. Rauchwerger and D. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In D. Wall, editor, *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, volume 30(6), New York, NY, June 1995.

[110] X. Redon. *Detection et exploitation des récurrences dans les programmes scientifiques en vue de leur parallélisation.* PhD thesis, Université P. et M. Curie, Paris 6, January 1995.

[111] X. Redon and P. Feautrier. Detection of reductions in sequentials programs with loops. In A. Bode, M. Reeve, and G. Wolf, editors, *Int. Conf. on Parallel Architectures and Languages*, pages 132–145, Munich, Germany, June 1993.

[112] X. Redon and P. Feautrier. Scheduling reductions. In *ACM Int. Conf. on Supercomputing*, pages 117–125, Manchester, UK, July 1994.

[113] J. Robinson. Theorem proving on the computer. *J. of the ACM*, pages 163–174, 1963.

[114] C. Rosene. *Incremental Dependence Analysis.* PhD thesis, Rice University, Houston, Texas, March 1990.

[115] B. Ryder and M. Paul. Elimination algorithms for data flow analysis. *Computing Surveys*, 18(3):277–316, September 1986.

[116] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, New York, 1986.

[117] Z. Shen, Z. Li, and P. Yew. An empirical study of Fortran programs for parallelizing compilers. *IEEE Trans. on Parallel and Distributed Systems*, 1(3):356–364, 1990.

[118] J. Stern. *Fondements Mathématiques de l'Informatique.* McGraw-Hill, Paris, 1990.

[119] V. Sunderam. PVM : A framework for parallel distributed computing. *Concurrency : Practice and Experience*, 2(4):315–339, December 1990.

[120] R. Triolet. *Contribution à la parallélisation automatique de programmes Fortran comportant des appels de procédures.* PhD thesis, Université P. et M. Curie, Paris 6, 1984.

[121] R. Triolet, P. Feautrier, and F. Irigoin. Direct parallelization of call statements. In *ACM Symp. on Compiler Construction*, pages 176–185, Palo Alto, CA, June 1986.

[122] P. Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis.* PhD thesis, U. of Illinois at Urbana-Champaign, 1995.

[123] P. Tu and D. Padua. Automatic array privatization. In *Int. Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lect. Notes in Computer Science*, pages 500–521, Portland, OR, August 1993.

[124] P. Tu and D. Padua. Gated SSA-Based demand-driven symbolic analysis for parallelizing compilers. In *ACM Int. Conf. on Supercomputing*, pages 414–423, Barcelona, Spain, July 1995.

[125] D. Wilde and S. Rajopadhye. Memory reuse analysis in the polyhedral model. In *Euro-Par*, volume 1123 of *Lect. Notes in Computer Science*, pages 389–397, Lyon, France, August 1996.

[126] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[127] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman and The MIT Press, 1989.

[128] M. Wolfe. Beyond induction variables. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 85–122, San Francisco, CA, June 1992.

[129] D. Wonnacott. *Constraint-Based Array Dependence Analysis.* PhD thesis, University of Maryland, 1995.

[130] Y.-Q. Yang. *Tests des dépendances et transformations de programmes.* PhD thesis, École Nationale Supérieure des Mines de Paris, November 1994.

[131] Y.-Q. Yang, C. Ancourt, and F. Irigoin. Minimal data dependence abstractions for loop transformations: Extended version. *Int. J. of Parallel Programming*, 23(4):359–388, August 1995.

[132] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schawald. Vienna Fortran — a language specification. Technical Report 21, ICASE, 1992.

[133] H. Zima and B. Chapman. *Supercompilers for parallel and vector computers.* ACM Press, 1991.

Index

$\alpha_k^p(.)$, *see* parameter of the maximum
θ, *see* schedule
φ, 176
$\sigma(.)$, *see* source
$\varsigma_i^p(.)$, 74, *see also* polyhedral method
$\hat{\varsigma}_k^p(.,.)$, 107
$\prec$, *see* sequential order
Ω, 172
$x[k..l]$, 61
$-$, 74
$\ll$, *see* lexicographic order
$\langle .,. \rangle$, *see* operation
$\lhd$, *see* textual order

abstract interpretation, 114
affinization, 95
array region, 143, 145–146
  extended array region, 151
atom image, 146

C_k, 105, *see also* non-affine constraint
$c_h(.,.)$, *see* non-affine constraint
$c_i^*(.,.)$, 119, *see also* iterative analysis
Caravan, 139
Chinese remainder theorem, 81
clause, 123
context, 76
  quast, 132
copy-in, 68
copy-out, 68

$\mathsf{D}_k(.)$, *see* parameter domain
$d_{\mathtt{SR}}$, 63
$d_{\mathtt{S}}$, 63
dataflow
  analysis, 57
  array dataflow analysis, 57
  graph (DFG), 63
dependence
  covered, killed, 63
  depth, 63
  direct dependence, 63
  distance/direction, 64, 91
  level, 64
  memory-based dependence, 90, 147
  polyhedron, 64
  relation, 64, 90, 92, 93, 143, 146
  test, 65
  value-based dependence, 90, 148

e_h, 104, *see also* non-affine constraint
environment, 72
existence predicate, 72
expansion, 166
  array expansion, 68
  maximal static expansion, 166, 172
  static expansion, 172
  total expansion, 68

FADA, 103, 143
Farkas theorem, 128, 129, 184
Fourier-Motzkin elimination, 79
ϕ-function, 114, 166

hidden variable, 144

$\mathsf{I}(.)$, *see* existence predicate
IN, 146, *see also* array region
induction variable detection, 115
iteration vector, 61
iterative analysis, 118

$\mathsf{L}_k^p(.)$, 106
$l_k^p(.,.)$, 116, *see also* properties on non-affine constraints
Last Write Tree (LWT), 93
lazy algorithm, 77
lexicographic order, 61
literal, 123

M, 174
$\mathsf{M}_{\mathtt{S}\rightarrow\mathtt{R}}$, *see* memory-based dependence
m_k, 105, *see also* non-affine constraint

non-affine constraint, 62, 104

operation, 58
OUT, 146, *see also* array region

P, *see* properties on non-affine constraints
$\hat{\mathsf{P}}$, *see* properties on parameters
parameter
  domain, 104
  of the maximum, 107
  removing parameters, 133
  structure parameter, 61
partial cover, 90, 149
polyhedral method, 72
Presburger arithmetic, 90
privatization, 68, 146, 156, 166
program checking, 162
properties, 108, 114, 122
  on non-affine constraints, 108, 114
  on parameters, 108

$\mathsf{Q}_i(.)$, 73, *see also* polyhedral method
$\hat{\mathsf{Q}}_k^p(.)$, 106, *see also* parameter of the maximum
$\mathsf{Q}_i^p(.)$, 73, *see also* polyhedral method
quasi-affine expression, 71
quast, 64, 74, 91, *see also* context
  combination rules, 74
  extension of quast structure, 80, 84
  simplification, 76

R, 173
reaching definition, 143, 145
READ, 146, *see also* array region
recurrence, 69, 164
  system of linear recurrence equations (SLRE), 164
region, *see* array region
resolution, 123
  input linear resolution (ILR), 127
  simple resolution rule, 123

$\mathsf{S}(.)$, *see* source set
schedule, 67, 183
sequencing predicate, 72
sequential order, 62, 63
simplex, 79
single assignment, 68
  gated single assignment, 114
  static single assignment, 114, 148, 166
sink, 57
source, 57, 73
  set, 109
speculative execution, 99, 184
static control program, 72
structural analysis, 117
subscript equation, 72
symbolic analysis, 99, 114–116, 148

textual order, 63

unification, 123
uninterpreted function symbol, 147

$\mathsf{V}_{\mathtt{S}\rightarrow\mathtt{R}}$, *see* value-based dependence

W, 172
WRITE, 146, *see also* array region

Résumé

L'analyse des dépendances de flot de données est une étape cruciale lors de la parallélisation. La description détaillée des dépendances entre opérations et pour chaque élément de tableau rend possible l'application de techniques de parallélisation performantes. Cependant, ce type d'analyse a deux principaux inconvénients : son coût élevé et son domaine restreint à des dépendances affines en fonction des compteurs de boucles.

On décrit d'abord dans cette thèse un algorithme polynômial pour le calcul des dépendances affines, dont la complexité et le domaine d'application sont meilleurs que ceux des méthodes existantes. Puis, dans la continuité des travaux de J.-F. Collard, on propose un cadre général pour l'analyse, éventuellement approchée, de n'importe quelle dépendance. Le modèle de programmes est formé des programmes sans procédure, comportant des accès quelconques aux éléments de tableaux. Une méthode itérative originale trouve des propriétés entre les contraintes non-affines du problème afin d'améliorer la précision du résultat. Notre méthode est capable de tirer parti de n'importe quelle caractérisation affine de ces contraintes et possède un critère d'optimalité de l'approximation.

Enfin, plusieurs applications traditionnelles de l'analyse de flot de données sont adaptées à notre méthode approchée et nous détaillons plus particulièrement l'expansion mémoire, en donnant une méthode offrant un compromis entre surcoût à l'exécution, taille mémoire et degré de parallélisme.

Mots clés : analyse de flot de données pour tableaux, parallélisation automatique, analyse de dépendances, contraintes non-linéaires, expansion mémoire.

Abstract

Array dataflow dependence analysis is paramount for automatic parallelization. The description of dependences at the operation and array element level has been shown to improve significantly the output of many code optimizations. But this kind of analysis has two main issues: its high cost and its scope limited to a small number of programs.

We first describe a new polynomial-time algorithm, outperforming other current methods in terms of both complexity and application domain. Then, in the continuity of the work done by J.-F. Collard, we present a general framework so as to handle any kind of dependences, by possibly producing approximate dependences. The model of programs is extended to any reducible control graph and any kind of references to array elements. An original method called iterative analysis, finds relations between non-affine constraints so as to improve the accuracy of the method. Besides, we provide a criterion ensuring that the approximation obtained is the best with respect to the information gathered on non-affine constraints by other analyses.

Finally, several traditional applications of dataflow analyses are adapted to our method in order to take advantage of its results, and we detail more specifically an array expansion that is a trade-off between run-time overhead, memory requirement and degree of parallelism.

Keywords: array dataflow analysis, automatic parallelization, dependence analysis, non-linear constraints, memory expansion.