THÈSE de DOCTORAT de l'UNIVERSITÉ de VERSAILLES

Spécialité : Informatique

présentée par

Albert COHEN

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

Sujet de la thèse :

Analyse et transformation de programmes : du modèle polyédrique aux langages formels

Program Analysis and Transformation: From the Polytope Model to Formal Languages

Soutenue le 21 décembre 1999 devant le jury composé de :

Jean	Berstel	Rapporteur
Luc	Bougé	Examinateur
Jean-François	Collard	Directeur
Paul	Feautrier	Directeur
William	Jalby	Président
Patrice	Quinton	Rapporteur
Bernard	Vauquelin	Rapporteur

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

Remerciements

Cette thèse a été préparée au sein du laboratoire PRiSM (Parallélisme, Réseaux, Systèmes et Modélisation) de l'Université de Versailles Saint-Quentin-en-Yvelines, entre septembre 1996 et décembre 1999, sous la direction de Jean-François Collard et Paul Feautrier.

Je voudrais tout d'abord m'adresser à Jean-François Collard (chargé de recherches au CNRS) qui a encadré cette thèse, et avec qui j'ai eu la chance de faire mes premiers pas dans la recherche scientifique. Ses conseils, sa disponibilité extraordinaire, son dynamisme en toutes circonstances, et ses idées éclairées ont fait beaucoup plus qu'entretenir ma motivation. Je remercie vivement Paul Feautrier (professeur au PRiSM) pour sa confiance et pour son intérêt à suivre mes résultats. À travers son expérience, il m'a fait découvrir à quel point la recherche est enthousiasmante, au delà des difficultés et des succès ponctuels.

Je suis très reconnaissant envers tous les membres de mon Jury ; notamment envers Jean Berstel (professeur à l'Université de Marne-la-Vallée), Patrice Quinton (professeur à l'IRISA, Université de Rennes) et Bernard Vauquelin (professeur au LaBRI, Université de Bordeaux), pour l'intérêt et la curiosité qu'ils ont porté à l'égard de mes travaux et pour le soin avec lequel ils ont relu cette thèse, y compris lorsque la problématique n'appartenait pas à leurs domaines de recherches. Un grand merci à Luc Bougé (professeur au LIP, École Normale Supérieure de Lyon) pour sa participation à ce Jury et pour ses suggestions et commentaires éclairés. Merci enfin à William Jalby (professeur au PRiSM) pour avoir accepté de présider ce Jury et pour m'avoir souvent conseillé avec bonne humeur.

J'exprime également toute ma gratitude à Guy-René Perrin pour ses encouragements et pour l'accès à « sa » machine parallèle, à Olivier Carton pour son aide précieuse sur un domaine très exigeant, à Denis Barthou, Ivan Djelic et Vincent Lefebvre pour leur collaboration essentielle aux résultats de cette thèse. Je me souviens aussi de passionnantes discussions avec Pierre Boulet, Philippe Clauss, Christine Eisenbeis et Sanjay Rajopadhye ; et je n'oublie pas non plus l'aide efficace des ingénieurs et des secrétaires du laboratoire. Je repense aux bons moments passés avec les tous les membres du « monastère » et avec les compagnons de route du PRiSM qui sont devenus mes amis.

Merci enfin à ma famille pour son soutien constant et inconditionnel, avec une pensée particulière pour mes parents et pour ma femme Isabelle.

Dedicated to a Brave GNU World

http://www.gnu.org

This document was typeset using LaTeX and the french package.

Graphics were designed using xfig, gnuplot and the GasTeX package.

Albert.Cohen@prism.uvsq.fr

Table of Contents

List of Figures **7**

List of Algorithms **9**

Présentation en français **11**
Grandes lignes de la thèse, en français.
Dissertation summary, in French.

1 Introduction **53**
1.1 Program Analysis 54
1.2 Program Transformations for Parallelization 57
1.3 Thesis Overview 60

2 Framework **61**
2.1 Going Instancewise 61
2.2 Program Model 63
2.2.1 Control Structures 63
2.2.2 Data Structures 64
2.3 Abstract Model 65
2.3.1 Naming Statement Instances 66
2.3.2 Sequential Execution Order 70
2.3.3 Adressing Memory Locations 71
2.3.4 Loop Nests and Arrays 74
2.4 Instancewise Analysis 75
2.4.1 Conflicting Accesses and Dependences 76
2.4.2 Reaching Definition Analysis 77
2.4.3 An Example of Instancewise Reaching Definition Analysis 78
2.4.4 More About Approximations 80
2.5 Parallelization 81
2.5.1 Memory Expansion and Parallelism Extraction 81
2.5.2 Computation of a Parallel Execution Order 82
2.5.3 General Efficiency Remarks 85

3 Formal Tools **87**
3.1 Presburger Arithmetics 87
3.1.1 Sets, Relations and Functions 88
3.1.2 Transitive Closure 89
3.2 Monoids and Formal Languages 90
3.2.1 Monoids and Morphisms 90
3.2.2 Rational Languages 91
3.2.3 Algebraic Languages 92
3.2.4 One-Counter Languages 94

3.3 Rational Relations . . . 97
3.3.1 Recognizable and Rational Relations . . . 97
3.3.2 Rational Transductions and Transducers . . . 98
3.3.3 Rational Functions and Sequential Transducers . . . 99
3.4 Left-Synchronous Relations . . . 101
3.4.1 Definitions . . . 102
3.4.2 Algebraic Properties . . . 104
3.4.3 Functional Properties . . . 107
3.4.4 An Undecidability Result . . . 109
3.4.5 Studying Synchronizability of Transducers . . . 110
3.4.6 Decidability Results . . . 112
3.4.7 Further Extensions . . . 113
3.5 Beyond Rational Relations . . . 114
3.5.1 Algebraic Relations . . . 114
3.5.2 One-Counter Relations . . . 116
3.6 More about Intersection . . . 119
3.6.1 Intersection with Lexicographic Order . . . 119
3.6.2 The case of Algebraic Relations . . . 120
3.7 Approximating Relations on Words . . . 121
3.7.1 Approximation of Rational Relations by Recognizable Relations . . . 121
3.7.2 Approximation of Rational Relations by Left-Synchronous Relations . . . 121
3.7.3 Approximation of Algebraic and Multi-Counter Relations . . . 122

4 Instancewise Analysis for Recursive Programs 123
4.1 Motivating Examples . . . 123
4.1.1 First Example: Procedure `Queens` . . . 123
4.1.2 Second Example: Procedure `BST` . . . 125
4.1.3 Third Example: Function `Count` . . . 125
4.1.4 What Next? . . . 126
4.2 Mapping Instances to Memory Locations . . . 126
4.2.1 Induction Variables . . . 126
4.2.2 Building Recurrence Equations on Induction Variables . . . 128
4.2.3 Solving Recurrence Equations on Induction Variables . . . 133
4.2.4 Computing Storage Mappings . . . 134
4.2.5 Application to Motivating Examples . . . 137
4.3 Dependence and Reaching Definition Analysis . . . 139
4.3.1 Building the Conflict Transducer . . . 139
4.3.2 Building the Dependence Transducer . . . 140
4.3.3 From Dependences to Reaching Definitions . . . 141
4.3.4 Practical Approximation of Reaching Definitions . . . 143
4.4 The Case of Trees . . . 145
4.5 The Case of Arrays . . . 147
4.6 The Case of Composite Data Structures . . . 148
4.7 Comparison with Other Analyses . . . 150
4.8 Conclusion . . . 154

5 Parallelization via Memory Expansion 155
5.1 Motivations and Tradeoffs . . . 155
5.1.1 Conversion to Single-Assignment Form . . . 156
5.1.2 Run-Time Overhead . . . 157
5.1.3 Single-Assignment for Loop Nests . . . 160
5.1.4 Optimization of the Run-Time Overhead . . . 161

5.1.5 Tradeoff between Parallelism and Overhead . . . 168
5.2 Maximal Static Expansion . . . 168
5.2.1 Motivation . . . 168
5.2.2 Problem Statement . . . 173
5.2.3 Formal Solution . . . 174
5.2.4 Algorithm . . . 176
5.2.5 Detailed Review of the Algorithm . . . 177
5.2.6 Application to Real Codes . . . 180
5.2.7 Back to the Examples . . . 181
5.2.8 Experiments . . . 185
5.2.9 Implementation . . . 185
5.3 Storage Mapping Optimization . . . 186
5.3.1 Motivation . . . 187
5.3.2 Problem Statement and Formal Solution . . . 191
5.3.3 Optimality of the Expansion Correctness Criterion . . . 194
5.3.4 Algorithm . . . 195
5.3.5 Array Reshaping and Renaming . . . 196
5.3.6 Dealing with Tiled Parallel Programs . . . 199
5.3.7 Schedule-Independent Storage Mappings . . . 200
5.3.8 Dynamic Restoration of the Data-Flow . . . 201
5.3.9 Back to the Examples . . . 201
5.3.10 Experiments . . . 204
5.4 Constrained Storage Mapping Optimization . . . 205
5.4.1 Motivation . . . 206
5.4.2 Problem Statement . . . 209
5.4.3 Formal Solution . . . 210
5.4.4 Algorithm . . . 214
5.4.5 Building Expansion Constraints . . . 215
5.4.6 Graph-Coloring Algorithm . . . 217
5.4.7 Dynamic Restoration of the Data-Flow . . . 219
5.4.8 Parallelization after Constrained Expansion . . . 222
5.4.9 Back to the Motivating Example . . . 223
5.5 Parallelization of Recursive Programs . . . 226
5.5.1 Problems Specific to Recursive Structures . . . 227
5.5.2 Algorithm . . . 228
5.5.3 Generating Code for Read References . . . 230
5.5.4 Privatization of Recursive Programs . . . 232
5.5.5 Expansion of Recursive Programs: Practical Examples . . . 233
5.5.6 Statementwise Parallelization . . . 235
5.5.7 Instancewise Parallelization . . . 240
5.6 Conclusion . . . 242

6 Conclusion **245**
6.1 Contributions . . . 245
6.2 Perspectives . . . 247

Bibliography **249**

Index **259**

List of Figures

1.1 Simple examples of memory expansion . . . 58
1.2 Run-time restoration of the flow of data . . . 59
1.3 Exposing parallelism . . . 59

2.1 About run-time instances and accesses . . . 62
2.2 Procedure `Queens` and control tree . . . 67
2.3 Control automata for program `Queens` . . . 69
2.4 Hash-table declaration . . . 72
2.5 An `inode` declaration . . . 73
2.6 Computation of Parikh vectors . . . 74
2.7 Execution-dependent storage mappings . . . 77

3.1 Studying the Lukasiewicz language . . . 95
3.2 One-counter automaton for the Lukasiewicz language . . . 96
3.3 Sequential and sub-sequential transducers . . . 100
3.4 Synchronous and δ-synchronous transducers . . . 103
3.5 Left-synchronous realization of several order relations . . . 103
3.6 A left and right synchronizable example . . . 104

4.1 Procedure `Queens` and control tree . . . 124
4.2 Procedure `BST` and compressed control automaton . . . 125
4.3 Procedure `Count` and compressed control automaton . . . 126
4.4 First example of induction variables . . . 127
4.5 More examples of induction variables . . . 128
4.6 Procedure `Count` and control automaton . . . 138
4.7 Rational transducer for storage mapping f of program `BST` . . . 146
4.8 Rational transducer for conflict relation κ of program `BST` . . . 146
4.9 Rational transducer for dependence relation δ of program `BST` . . . 147
4.10 Rational transducer for storage mapping f of program `Queens` . . . 147
4.11 One-counter transducer for conflict relation κ of program `Queens` . . . 149
4.12 Pseudo-left-synchronous transducer for the restriction of κ to $\mathbf{W} \times \mathbf{R}$ . . . 150
4.13 One-counter transducer for the restriction of dependence relation δ to flow dependences . . . 151
4.14 One-counter transducer for reaching definition relation σ of program `Queens` . . . 152
4.15 Simplified one-counter transducer for σ . . . 152

5.1 Interaction of reaching definition analysis and run-time overhead . . . 159
5.2 Basic optimizations of the generated code for ϕ functions . . . 163
5.3 Repeated assignments to the same memory location . . . 164
5.4 Improving the SA algorithm . . . 165
5.5 Parallelism extraction versus run-time overhead . . . 167
5.6 First example . . . 169
5.7 First example, continued . . . 170

5.8 Expanded version of the first example . . . 170
5.9 Second example . . . 170
5.10 Partition of the iteration domain ($N = 4$) . . . 171
5.11 Maximal static expansion for the second example . . . 172
5.12 Third example . . . 172
5.13 Inserting copy-out code . . . 181
5.14 Parallelization of the first example. . . . 185
5.15 Experimental results for the first example . . . 186
5.16 Computation times, in milliseconds. . . . 186
5.17 Convolution example . . . 187
5.18 Knapsack program . . . 188
5.19 KP in single-assignment form . . . 189
5.20 Instancewise reaching definitions, schedule, and tiling for KP . . . 190
5.21 Partial expansion for KP . . . 190
5.22 Cases of $f_e^{\mathrm{EXP}}(v) \neq f_e^{\mathrm{EXP}}(w)$ in (5.17) . . . 194
5.23 Motivating examples for each constraint in the definition of the interference relation 195
5.24 An example of block-regular storage mapping . . . 200
5.25 Time and space optimization . . . 205
5.26 Performance results . . . 205
5.27 Motivating example . . . 206
5.28 Parallelization of the motivating example . . . 207
5.29 Performance results for storage mapping optimization . . . 208
5.30 Maximal static expansion . . . 208
5.31 Maximal static expansion combined with storage mapping optimization . . . 209
5.32 What we want to achieve . . . 210
5.33 Strange interplay of constraint and coloring relations . . . 213
5.34 How we achieve constrained storage mapping optimization . . . 214
5.35 Solving the constrained storage mapping optimization problem . . . 215
5.36 Single-assignment form conversion of program `Queens` . . . 234
5.37 Implementation of the read reference in statement r . . . 235
5.38 Privatization of program `Queens` . . . 236
5.39 Parallelization of program `BST` . . . 237
5.40 Second motivating example: program `Map` . . . 237
5.41 Parallelization of program `Queens` via privatization . . . 239
5.42 Parallel resolution of the n-Queens problem . . . 240
5.43 Instancewise parallelization example . . . 241
5.44 Automatic instancewise parallelization of procedure `P` . . . 243

List of Algorithms

Recurrence-Build ($program$) . . . 130
Recurrence-Rewrite ($program, system$) . . . 131
Recurrence-Solve ($system$) . . . 134
Compute-Storage-Mappings ($program$) . . . 135
Dependence-Analysis ($program$) . . . 141
Reaching-Definition-Analysis ($program$) . . . 145
Abstract-SA ($program, \mathbf{W}, \sigma$) . . . 157
Abstract-Implement-Phi ($expanded$) . . . 158
Convert-Quast ($quast, ref$) . . . 161
Loop-Nests-SA ($program, \sigma$) . . . 161
Loop-Nests-Implement-Phi ($expanded$) . . . 162
Abstract-ML-SA ($program, \mathbf{W}, \sigma^{\mathrm{ML}}$) . . . 166
Loop-Nests-ML-SA ($program, \sigma^{\mathrm{ML}}$) . . . 166
Abstract-Implement-Phi-Not-SA ($expanded$) . . . 167
Maximal-Static-Expansion ($program, \kappa, \sigma$) . . . 177
MSE-Convert-Quast ($quast, ref$) . . . 177
Compute-Representatives ($equivalence$) . . . 178
Enumerate-Representatives (rel, fun) . . . 179
Storage-Mapping-Optimization ($program, \sigma, \not\kappa, <_{\mathrm{PAR}}$) . . . 196
SMO-Convert-Quast ($quast, ref$) . . . 197
Build-Expansion-Vector ($S, \bowtie$) . . . 198
Partial-Renaming ($program, \bowtie$) . . . 199
Constrained-Storage-Mapping-Optimization ($program, \kappa, \sigma, \equiv, <_{\mathrm{PAR}}$) . . . 216
CSMO-Convert-Quast ($quast, ref$) . . . 216
Cyclic-Coloring ($\Diamond\!\Diamond$) . . . 218
Near-Block-Cyclic-Coloring ($\Diamond\!\Diamond, shape$) . . . 219
CSMO-Implement-Phi ($expanded$) . . . 220
CSMO-Efficiently-Implement-Phi ($expanded$) . . . 221
Recursive-Programs-SA ($program, \sigma$) . . . 229
Recursive-Programs-Implement-Phi ($expanded$) . . . 230
Recursive-Programs-Online-SA ($program, \sigma$) . . . 232
Statementwise-Parallelization ($program, \kappa$) . . . 238
Instancewise-Parallelization ($program, \kappa$) . . . 242

Présentation en français

Après une introduction détaillée, 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 pourra donc se reporter à la partie correspondante en anglais pour y trouver le détail des algorithmes ainsi que des exemples.

Table des matières

I Introduction 12
- I.1 Analyse de programmes 13
- I.2 Transformations de programmes pour la parallélisation 16
- I.3 Organisation de cette thèse 19

II Modèles 20
- II.1 Une vision par instances 20
- II.2 Modèle de programmes 21
- II.3 Modèle formel 22
- II.4 Analyse par instances 25
- II.5 Parallélisation 26

III Outils mathématiques 27
- III.1 Arithmétique de Presburger 27
- III.2 Langages formels et relations rationnelles 28
- III.3 Relations synchrones à gauche 31
- III.4 Dépasser les relations rationnelles 32
- III.5 Compléments sur les approximations 34

IV Analyse par instance pour programmes récursifs 34
- IV.1 Exemples introductifs 34
- IV.2 Relier instances et cellules mémoire 35
- IV.3 Analyse de dépendances et de définitions visibles 38
- IV.4 Les résultats de l'analyse 39
- IV.5 Comparaison avec d'autres analyses 41

V Expansion et parallélisation 42
- V.1 Motivations et compromis 42
- V.2 Expansion statique maximale 44
- V.3 Optimisation de l'occupation en mémoire 45
- V.4 Expansion optimisée sous contrainte 45
- V.5 Parallélisation de programmes récursifs 46

VI Conclusion 49
- VI.1 Contributions 49
- VI.2 Perspectives 51

I Introduction

Les progrès accomplis en matière de technologie des processeurs résultent de plusieurs facteurs : une forte augmentation de la fréquence, des bus plus larges, l'utilisation de plusieurs unités fonctionnelles et éventuellement de plusieurs processeurs, le recours à des hiérarchies mémoire complexes pour compenser les temps d'accès, et un développement global des capacités de stockage. Une conséquence est que le modèle de machine devient de moins en moins simple et uniforme : en dépit de la gestion matérielle des caches, de l'exécution superscalaire et des architectures parallèles à mémoire partagée, la recherche des performances optimales pour un programme donné devient de plus en plus complexe. De bonnes optimisations pour un cas particulier peuvent conduire à des résultats désastreux avec une architecture différente. De plus, la gestion matérielle n'est pas capable de tirer partie efficacement des architectures les plus complexes : en présence de hiérarchies mémoire profondes, de mémoires locales, de calcul *out of core*, de parallélisme d'instructions ou de parallélisme à gros grain, une aide du compilateur est nécessaire pour obtenir de bonnes performances.

L'industrie des architectures et des compilateurs tout entière affronte en réalité ce que la communauté du calcul à hautes performances a découvert depuis des années. D'une part, et pour la plupart des applications, les architectures sont trop disparates pour définir des critères d'efficacité pratiques et pour développer des optimisations spécifiques pour une machine donnée. D'autre-part, les programmes sont écrits de telle sorte que les techniques traditionnelles d'optimisation et de parallélisation ont tout le mal du monde à nourrir la bête de calcul l'on s'apprête à installer dans un banal ordinateur portable.

Pour atteindre des performances élevées à l'aide des microprocesseurs modernes et des ordinateurs parallèles, un programme — ou bien l'algorithme qu'il implémente — doit posséder un degré suffisant de parallélisme. Dans ces conditions, les programmeurs ou les compilateurs doivent mettre en évidence ce parallélisme et appliquer les transformations nécessaires pour adapter le programme aux caractéristiques de la machine. Une autre exigence est que le programme soit portable sur des architectures différentes, afin de suivre l'évolution rapide des machines parallèles. Les deux possibilités suivantes sont ainsi offertes aux programmeurs.

- Premièrement, les langages à parallélisme explicite. La plupart sont des extensions parallèles de langages séquentiels. Ces langages peuvent être à parallélisme de données, comme HPF, ou combiner parallélisme de données et de tâches, comme les extensions OpenMP pour architectures à mémoire partagée. Quelques extensions sont proposées sous la forme de bibliothèques : PVM et MPI par exemple, ou bien des environnements de haut niveau comme IML de l'Université de l'Illinois [SSP99] ou Cilk du MIT [MF98]. Toutes ces approches facilitent la programmation d'algorithmes parallèles. En revanche, le programmeur est chargé de certaines opérations techniques comme la distribution des données sur les processeurs, les communications et les synchronisations. Ces opérations requièrent une connaissance approfondie de l'architecture et réduisent notablement la portabilité.

- Deuxièmement, la parallélisation automatique d'un langage séquentiel de haut niveau. Les avantages évidents de cette approche sont la portabilité et la simplicité de la programmation. Malheureusement, la tâche qui incombe au compilateur paralléliseur devient écrasante. En effet, le programme doit tout d'abord être analysé afin de comprendre — au moins partiellement — quels calculs sont effectués et où réside le parallélisme. Le compilateur doit alors générer un code parallèle, en prenant en compte les spécificités de l'architecture. Le langage source usuel pour la parallélisation automatique est le Fortran 77. En effet, de nombreuses applications scientifiques ont été écrites en Fortran, n'autorisant que des structures de données et de contrôle relativement simples. Plusieurs études considèrent néanmoins la parallélisation du C ou de langages fonctionnels comme Lisp. Ces recherches sont moins avancées que l'approche historique mais plus proches de ce travail : elles considèrent les structures de données et de contrôle les plus générales. De nombreux projets de recherche existent : Parafrase-2 et Polaris [BEF+96] de l'Université de l'Illinois, PIPS de l'École des Mines de Paris [IJT90], SUIF de l'Université de Stanford [H+96], le compilateur McCat/Earth-C de l'Université Mc Gill [HTZ+97], LooPo de l'Université de Passau [GL97], et PAF de l'Université de Versailles ; il y a également un nombre croissant d'outils de parallélisation commerciaux, comme CFT, FORGE, FORESYS ou KAP.

Nous nous intéressons principalement aux techniques de parallélisation automatique et semi-automatique : cette thèse aborde à la fois l'analyse et la transformation de programmes.

I.1 Analyse de programmes

Optimiser ou paralléliseur un programme revient généralement à transformer son code source, en améliorant un certain nombre de paramètres de l'exécution. Pour appliquer une transformation de programme à la compilation, on doit s'assurer que l'algorithme implémenté n'est pas touché au cours de l'opération. Étant donné qu'un algorithme peut être implémenté de bien des manières différentes, la validation d'une transformation de programmes requiert un processus d'ingénierie à l'envers (*reverse engineering*) pour établir l'information la plus précise possible sur ce que fait le programme. Cette technique fondamentale d'analyse de programmes tente de résoudre le problème difficile de la mise en évidence *statique* — c.-à-d. à la *compilation* — d'informations sur les propriétés *dynamiques* — c.-à-d. à l'*exécution*.

Analyse statique

En matière d'analyse de programmes, les premières études se sont portées sur les propriétés de l'état de la machine entre l'exécution de deux instructions. Ces états sont appelés *points de programmes*. De telles propriétés sont dites *statiques* car elles recouvrent *toutes les exécutions possibles* conduisant à un point de programme donné. Bien entendu, ces propriétés sont calculées lors de la compilation, mais le sens de l'adjectif « statique » ne vient pas de là : il serait probablement plus approprié de parler d'analyse « syntaxique ».

L'analyse de *flot de données* est le premier cadre général proposé pour formaliser le grand nombre d'analyses statiques. Parmi les nombreuses présentations de ce formalisme [KU77, Muc97, ASU86, JM82, KS92, SRH96], on peut identifier les points communs suivants. Pour décrire les exécutions possibles, la méthode usuelle consiste à construire le *graphe de flot de contrôle* du programme [ASU86]; en effet, ce graphe représente tous les points comme des sommets, et les arêtes entre ces sommets sont étiquetées par des instructions du programme. L'ensemble de toutes les exécutions possibles est alors l'ensemble de tous les *chemins* depuis l'état initial jusqu'au point de programme considéré. Les propriétés en un point donné sont définies de la façon suivante : puisque chaque instruction peut modifier une propriété, on doit prendre en compte tous les chemins conduisant au point de programme et *rassembler* (meet) toutes les informations sur ces chemins. La formalisation de ces idées est souvent appelée *rassemblement sur tous les chemins* ou meet over all paths (MOP). Bien sûr, l'opération de *rassemblement* dépend de la propriété recherchée et de l'abstraction mathématique pour celle-ci.

En revanche, le nombre potentiellement infini de chemins interdit toute évaluation de propriétés à partir de la spécification MOP. Le calcul est réalisé en propageant les résultats intermédiaires — en avant ou en arrière — le long des arêtes du graphe de flot de contrôle. On procède alors à une résolution *itérative* des *équations de propagation*, jusqu'à ce qu'un *point fixe* soit atteint. C'est la méthode dite du *point fixe maximal* ou maximal fix-point (MFP). Dans le cas intra-procédural, Kam et Ullman [KU77] ont prouvé que MFP calcule effectivement le résultat défini par MOP — c.-à-d. MFP *coïncide* avec MOP — lorsque quelques propriétés simples de l'abstraction mathématique sont satisfaites ; et ce résultat a été étendu à l'analyse inter-procédurale par Knoop et Steffen [KS92].

Les abstractions mathématiques pour les propriétés de programmes sont très nombreuses, en fonction de l'application et de la complexité de l'analyse. La structure de *treillis* englobe la plupart des abstractions car elle autorise le calcul des *rassemblements* (meet) — aux points de rencontre — et des *jointures* (join) — associées aux instructions. Dans ce cadre, Cousot et Cousot [CC77] ont proposé un schéma d'*approximation* fondé sur des connections de Galois semi-duales entre les états *concrets* de l'exécution et les propriétés *abstraites* à la compilation. Ce formalisme appelé *interprétation abstraite* a deux intérêts principaux : tout d'abord, il permet de construire systématiquement des abstractions des propriétés à l'aide de treillis, et d'un autre coté, il garantit que tout point fixe calculé dans le treillis abstrait correspond à une *approximation conservatrice* d'un point fixe dans le treillis des états concrets. Tout en étendant le concept d'analyse de flot de données, l'interprétation abstraite facilite les preuves de correction et d'optimalité des analyses de programmes. Des applications pratiques de l'interprétation abstraite et des méthodes itératives associées sont présentées dans [Cou81, CH78, Deu92, Cre96].

Malgré d'indéniables succès, les analyses de flot de données — fondées ou non sur l'interprétation abstraite — ont rarement été à la base des techniques de parallélisation automatique. Certaines raisons importantes ne sont pas de nature scientifique, mais de bonnes raisons expliquent également ce fait :

– les techniques MOP/MFP sont principalement orientées vers les optimisations classiques avec des abstractions relativement simples (les treillis ont souvent une hauteur bornée) ; leur correction et leur efficacité dans un véritable compilateur sont les enjeux déterminants, alors que la précision et l'expressivité de l'abstraction mathématique sont à la base de la parallélisation automatique ;

– dans l'industrie, les méthodes de parallélisation se sont traditionnellement concentrées sur les nids de boucles et sur les tableaux, avec des degrés importants de parallélisme de données et des structures de contrôle simples (non récursives, du premier ordre) ; prouver la correction d'une analyse est facile dans ces conditions, alors que l'application à des programmes réels et l'implémentation dans un compilateur deviennent des enjeux majeurs ;

– l'interprétation abstraite convient aux langages fonctionnels avec une sémantique opérationnelle propre et simple ; les problèmes soulevés sont alors orthogonaux aux questions pratiques liées aux langages impératifs et bas niveau, traditionnellement plus adaptés aux architectures parallèles (on verra que cette situation évolue).

En conséquence, les analyses de flot de données existantes sont généralement des analyses statiques qui calculent des propriétés d'un *point de programme* donné ou d'une *instruction* donnée. De tels résultats sont utiles aux techniques classiques de vérification et d'optimisation [Muc97, ASU86, SKR90, KRS94], mais pour la parallélisation automatique on a besoin d'informations supplémentaires.

– Que dire des *différentes instances* d'un point de programme ou d'une instruction *à l'exécution*? Puisque les instructions sont généralement exécutées plusieurs fois, on s'intéresse à *l'itération de boucle* ou à *l'appel de procédure* qui conduit à l'exécution de telle instruction.

– Que dire des *différents éléments* d'une structure de données? Puisque les tableaux et les structures de données allouées dynamiquement ne sont pas atomiques, on s'intéresse à *l'élément de tableau* ou *au nœud de l'arbre* qui est accédé par une *instance* donnée d'une instruction.

Analyse par instances

Les analyses de programmes pour la parallélisation automatique constituent un domaine assez restreint, comparé avec l'immensité des propriétés et des techniques étudiées dans le cadre de l'analyse statique. Le modèle de programme considéré est également plus restreint — la plupart du temps — puisque les applications traditionnelles des paralléliseurs sont les codes numériques avec des nids de boucles et des tableaux.

Dès le début — avec les travaux de Banerjee [Ban88], Brandes [Bra88] et Feautrier [Fea88a] — les analyses sont capables d'identifier des propriétés *au niveau des instances et des éléments*. Alors que la seule structure de contrôle était la boucle `for/do`, les méthodes itératives avec de solides fondations sémantiques paraissaient inutilement complexes. Pour se concentrer sur la résolution des problèmes cruciaux que sont l'abstraction des itérations de boucles et des effets les éléments de tableaux, la conception de modèles simples et spécialisés fut à coup sûr préférable. Les premières analyses étaient des *tests de dépendance* [Ban88] et des *analyses de dépendances* qui rassemblent des informations sur les instances d'instructions accédant à la même cellule mémoire, l'un des accès étant une écriture. Des méthodes plus précises ont été conçues pour calculer, pour chaque élément de tableau lu dans une expression, l'instance de l'instruction qui a produit la valeur. Elles sont souvent appelées *analyses de flot de données pour tableaux* [Fea91, MAL93], mais nous préférons le terme d'*analyse de définitions visibles par instances* pour favoriser la comparaison avec une technique particulière d'analyse *statique* de flot de données appelée *analyse de définitions visibles* [ASU86, Muc97]. Une information aussi précise améliore significativement la qualité des techniques de transformation, et donc les performances des programmes parallèles.

Les analyses par instances ont longtemps souffert de sévères restrictions sur leur modèle de programmes : ceux-ci devaient initialement ne comporter que des boucles sans instructions conditionnelles, avec des bornes et des indices de tableaux affines, et sans appels de procédures. Ce modèle limité englobe déjà bon nombre de codes numériques, et il a également le grand intérêt de permettre le calcul *exact* des dépendances et des définitions visibles [Fea88a, Fea91]. Lorsque l'on cherche à supprimer des restrictions, l'une des difficultés vient de l'impossibilité d'établir des résultats exacts, seule une information *approchée* sur les dépendances est disponible à la compilation : cela induit des approximations trop grossières sur les définitions visibles. Un calcul direct de ces définitions visibles est donc nécessaire. De telles techniques ont été récemment mises au point par Barthou, Collard et Feautrier [CBF95, BCF97, Bar98] et par Pugh et Wonnacott [WP95, Won95], avec des résultats extrêmement précis dans le cas *intra-procédural*. Par la suite, et dans le cas des tableaux et nids de boucles sans restrictions, notre analyse de définitions visibles par instances sera l'*analyse floue de flot des données* ou *fuzzy array dataflow analysis* (FADA) de Barthou, Collard et Feautrier [Bar98].

Il existe de nombreuses extensions de ces analyses qui sont capables de prendre en compte les appels de procédure [TFJ86, HBCM94, CI96], mais ce ne sont pas *pleinement* des analyses *par instances* car elles ne distinguent pas les exécutions multiples d'une instruction associées à des *appels différents* de la procédure englobante. En effet, cette thèse présente la première analyse qui soit pleinement par instances pour des programmes comportant des appels de procédures — éventuellement récursifs.

I.2 Transformations de programmes pour la parallélisation

Il est bien connu que les dépendances limitent la parallélisation des programmes écrits dans un langage impératif ainsi que leur compilation efficace sur les processeurs modernes et les super-calculateurs. Une méthode générale pour réduire le nombre de dépendances consiste à réduire la réutilisation de la mémoire en affectant des cellules mémoires distinctes à des écritures indépendantes, c'est-à-dire à *expanser* les structures de données.

Il y a de nombreuses techniques pour calculer des *expansions de la mémoire*, c'est-à-dire pour transformer les accès mémoire dans les programmes. Les méthodes classiques comportent : le renommage de variables ; le découpage ou l'unification de structures de données du même type ; le redimensionnement de tableaux, en particulier l'ajout de nouvelles dimensions ; la conversion de tableaux en arbres ; la modification du degré d'un arbre ; la transformation d'une variable globale en une variable locale.

Les références en lecture sont expansées également, en utilisant les définitions visibles pour implémenter la référence expansée [Fea91]. La figure 1 présente trois programmes pour lesquels aucune exécution parallèle n'est possible, en raison des dépendances de sortie (certains détails du code sont omis). Les versions expansées sont présentées en partie droite de la figure, pour illustrer l'intérêt de l'expansion de la mémoire pour l'extraction du parallélisme.

Malheureusement, lorsque le flot de contrôle ne peut pas être prédit à la compilation, un travail supplémentaire est nécessaire lors de l'exécution pour préserver le flot de données d'origine : des fonctions ϕ peuvent être nécessaires pour « rassembler » les définitions en provenance de divers chemins de contrôle entrants. Ces fonctions ϕ sont semblables — mais non identiques — à celles du formalisme d'*assignation unique statique* ou *static single-assignment* (SSA) de Cytron *et al.* [CFR+91], et Collard et Griebl les ont été étendues pour la première fois aux méthodes d'expansion par instances [GC95, Col98]. L'argument d'une fonction ϕ est l'*ensemble des définitions visibles possibles* pour la référence en lecture associee (cette interprétation est très différente de la sémantique usuelle des fonctions ϕ du formalisme SSA). La figure 2 propose deux programmes avec des expressions conditionnelles et des index de tableau inconnus. Des versions expansées avec fonctions ϕ sont données en partie droite de la figure.

L'expansion n'est pas une étape obligatoire de la parallélisation ; elle reste cependant une technique très générale pour exposer plus de parallélisme dans les programmes. En ce qui concerne l'implémentation de programmes parallèles, deux visions différentes sont possibles, en fonction du langage et de l'architecture.

```
int x;                          int x1, x2;
x = ···; ··· = x;               x1 = ···; ··· = x1;
x = ···; ··· = x;               x2 = ···; ··· = x2;
```

Après expansion, c.-à-d. après renommage de `x` en `x1` et `x2`, les deux premières instructions peuvent être exécutées en parallèle avec les deux autres.

```
   int A[10];                       int A1[10], A2[10][10];
   for (i=0; i<10; i++) {           for (i=0; i<10; i++) {
s1   A[0] = ···;               s1     A1[i] = ···;
     for (j=1; j<10; j++) {           for (j=1; j<10; j++) {
s2     A[j] = A[j-1] + ···;    s2       A2[i][j] = { if (j=1) A1[i];
     }                                               else A2[i][j-1]; }
                                                  + ···;
                                      }
```

Après expansion, c.-à-d. après renommage du tableau `A` en `A1` et `A2` puis ajout d'une dimension au tableau `A2`, la boucle `for` est parallèle. La définition visible par instances de la référence `A[j-1]` dépend des valeurs de `i` et `j`, comme le montre l'implémentation avec une instruction conditionnelle.

```
int A[10];                      struct Tree {
void Proc (int i) {               int value;
  A[i] = ···;                     Tree *left, *right;
  ··· = A[i];                   } *p;
  if (···) Proc (i+1);          void Proc (Tree *p, int i) {
  if (···) Proc (i-1);            p->value = ···;
}                                 ··· = p->value;
                                  if (···) Proc (p->left, i+1);
                                  if (···) Proc (p->right, i-1);
                                }
```

Après expansion, les deux appels de procédure peuvent être exécutés en parallèle. L'allocation dynamique de la structure `Tree` est omise.

Figure 1. Quelques exemples d'expansion

La première exploite le *parallélisme de contrôle*, c'est-à-dire le parallélisme entre des instructions différentes du même bloc de programme. Le but consiste à remplacer le plus d'exécutions séquentielles d'instructions par des exécutions parallèles. En fonction du langage, il y a plusieurs syntaxes différentes pour coder ce type de parallélisme, et celles-ci peuvent ne pas toutes avoir le même pouvoir d'expression. Nous préférons la syntaxe **spawn/sync** de Cilk [MF98] (proche de celle de OpenMP) aux *blocs parallèles* de Algol 68 et du compilateur EARTH-C [HTZ+97]. Comme dans [MF98], les synchronisations portent sur toutes les activités asynchrones commencées dans le *bloc englobant*, et des *synchronisations implicites* sont ajoutées aux points de retour des procédures. En ce qui concerne l'exemple de la figure 3, l'exécution de A, B et C en parallèle suivie séquentiellement de D puis de E a été écrite dans une syntaxe à la Cilk. En pratique, chaque instruction de cet exemple serait probablement un appel de procédure.

```
   int x;                        int x1, x2;
s1 x = ···;                   s1 x1 = ···;
s2 if (···) x = ···;          s2 if (···) x2 = ···;
r  ··· = x;                   r  ··· = φ({s1, s2});
```

Après expansion, on ne peut pas décider à la compilation quelle est la valeur lue par l'instruction r. On ne sait seulement que celle-ci ne peut venir que de s_1 ou de s_2, et le calcul de cette valeur est caché dans l'expression $\phi(\{s_1, s_2\})$. Celle-ci observe si s_2 a été exécutée, si oui elle retourne la valeur de x2, sinon celle de x1.

```
   int A[10];                    int A1[10], A2[10];
s1 A[i] = ···;                s1 A1[i] = ···;
s2 A[···] = ···;              s2 A2[···] = ···;
r  ··· = A[i];                r  ··· = φ({s1, s2});
```

Après expansion, on ne sait pas à la compilation quelle est la valeur lue par l'instruction r, puisque l'on ne connaît pas l'élément du tableau A écrit par l'instruction s_2.

Figure 2. Restauration du flot de données à l'exécution

```
spawn A;
spawn B;
spawn C;
sync;   // attente de la terminaison de A, B et C
D;
E;
```

Figure 3. Syntaxe du parallélisme de contrôle

La deuxème vision est exploite le *parallélisme de données*, c'ést-à-dire le parallélisme entre des instances différentes de la même instruction ou du même bloc. Le modèle à parallélisme de données a été longuement étudié dans le cas des nids de boucles [PD96], en raison de son adéquation avec les techniques efficaces de parallélisation pour les algorithmes numériques et pour les opérations répétitives sur de gros jeux de données. On utilisera une syntaxe similaire à la déclaration de boucles parallèles en OpenMP, où toutes les variables sont supposées *partagées* par défaut, et une *synchronisation implicite* est ajoutée à la fin de chaque sortie de boucle.

Pour générer du code à parallélisme de données, beaucoup d'algorithmes utilisent des transformations de boucles intuitives comme la fission, la fusion, l'échange, le renversement, la torsion, la réindexation de boucles et le réordonnancement des instructions. Mais le parallélisme de données est également adapté à l'expression d'un ordre d'exécution parallèle sous forme d'ordonnancement, c'est-à-dire en affectant une date d'exécution à chaque instance d'une instruction. Le schéma de programme de la figure 4 montre donne une idée de la méthode générale pour implémenter un tel ordonnancement [PD96]. Le concept de *front d'exécution* $F(t)$ est fondamental pusiqu'il rassemble toutes les instances $\imath$ qui s'exécutent à la date t.

Le premier algorithme d'ordonnancement est dû à Kennedy et Allen [AK87], lequel a

```
for (t=0; t<=L; t++) {   // L est la latence de l'ordonnancement
  parallel for (ı ∈ F(t))
    execute instance ı
  // synchronisation implicite
}
```

Figure 4. Implémentation classique d'un ordonnancement dans le modèle à parallélisme de données

inspiré de nombres méthodes. Elles se fondent toutes sur des abstractions relativement approximatives des dépendances, comme les niveaux, les vecteurs et les cônes de dépendance. La complexité raisonable et la facilité d'implémentation dans un compilateur industriel constituent les avantages principaux de ces méthodes ; les travaux de Banerjee [Ban92] et plus récemment de Darte et Vivien [DV97] donnent une vision globale de ces algorithmes. Une solution générale a été proposée par Feautrier [Fea92]. L'algorithme proposé est très utile, mais l'absence de support pour décider du paramètre de l'ordonnancement que l'on doit optimiser constitue un point faible : est-ce la latence L, le nombre de communications (sur une machine à mémoire distribuée), la largeur des fronts ?

Pour finir, il est bien connu que le parallélisme de contrôle est plus général que le parallélisme de données, en ce sens que tout programme à parallélisme de données peut être réécrit dans un modèle à parallélisme de contrôle, sans perte de parallélisme. C'est d'autant plus vrai pour les programmes récursifs où la distinction entre les deux paradigmes n'est pas très claire [Fea98]. En revanche, pour des programmes et des architectures réels, le parallélisme de données à longtemps été nettement plus adapté au calcul massivement parallèle — principalement en raison du surcoût associé à la gestion des activités. Des avancées récentes dans le materiel et les logiciels ont poutant montré que la situation est entrain d'évoluer : d'excellents résultats pour des programmes parallèles récursifs (simulations de jeux comme les échecs, et algorithmes de tri) ont été obtenus avec Cilk par exemple [MF98].

I.3 Organisation de cette thèse

Quatre chapitres structurent cette thèse avant la conclusion finale, et ceux-ci se reflètent dans les sections suivantes. La section II — résumant le chapitre 2 — décrit un formalisme général pour l'analyse et la transformation de programmes, et présente les définitions utiles aux chapitres suivants. Le but est d'être capable d'étudier une large classe de programmes, des nids de boucles avec tableaux aux programmes et structures de données récursifs.

Des résultats mathématiques sont rassemblés dans la section III — résumant le chapitre 3 ; certains sont bien connus, comme l'arithmétique de Presburger et la théorie des langages formels ; certains sont plutôt peu courants dans les domaines du parallélisme et de la compilation, comme les transductions rationnelles et algébriques ; et les autres sont principalement des contributions, comme les transductions synchrones à gauche et les techniques d'approximation pour transductions rationnelles et algébriques.

La section IV — résumant le chapitre 4 — s'attaque à l'analyse de par instances de programmes récursifs. Celle-ci est fondée sur une extension de la notion de variable d'induction aux programmes récursifs et sur de nouveaux résultats en théorie des langages formels. Deux algorithmes pour l'analyse de dépendance et de définition visible sont proposés. Ceux-ci sont expérimentés sur des exemples.

Les techniques de parallélisation fondées sur l'expansion de la mémoire constituent l'objet de la section V — résumant le chapitre 5. Les trois premières sous-sections présentent des techniques pour expanser les nids de boucles sans restriction d'expressions conditionnelles, de bornes de boucles et d'index de tableaux ; la quatrième sous-section est une contribution à l'optimisation simultanée des paramètres d'expansion et de parallélisation ; et la cinquième sous-section présente nos résultats sur l'expansion et la parallélisation de programmes récursifs.

II Modèles

Afin de conserver un formalisme et un vocabulaire constant tout au long de cette thèse, nous présentons un cadre général pour décrire des analyses et des transformations de programmes. Nous avons mis l'accent sur la représentation des propriétés de programmes au niveau des instances, tout en maintenant une certaine continuité avec les autres travaux du domaine. Nous ne cherchons à concurrencer aucun formalisme existant [KU77, CC77, JM82, KS92] : l'objectif principal consiste à établir des résultats convaincants sur la pertinence et l'efficacité de nos techniques.

Après une présentation formelle des instances d'instructions et des exécutions d'un programme, nous définissons un modèle de programmes pour le reste de cette étude. Nous décrivons ensuite les abstractions mathématiques associées, avant de formaliser les notions d'analyse et de transformation de code.

II.1 Une vision par instances

Au cours de l'exécution, chaque instruction peut être exécutée un certain nombre de fois, à cause des structures de contrôle englobantes. Pour décrire les propriétés du flot de données aussi précisément que possible, nos techniques doivent être capables de distinguer entre ces *différentes exécutions d'une même instruction*. Pour une instruction s, une ***instance*** de s à l'exécution est une exécution particulière de s au cours de l'exécution du programme. Dans le cas des nids de boucles, on utilise souvent les compteurs de boucles pour *nommer* les instances, mais cette technique n'est pas toujours applicable : un schéma général de nommage sera étudié dans la section II.3.

Les programmes dépendent parfois de l'état initial de la mémoire et interagissent avec leur environnement, plusieurs exécutions du même code sont donc associées à des ensembles d'instances différents et à des propriétés du flot incompatibles. Nous n'aurons pas besoin ici d'un degré élevé de formalisation : une ***exécution*** e d'un programme P est donnée par une *trace d'exécution* de P, c'est-à-dire une séquence finie ou infinie (lorsque le programme ne termine pas) de *configurations* (états de la machine). L'ensemble de toutes les exécutions possibles est noté $\mathbf{E}$. Pour un programme donné, on note $\mathbf{I}_e$ l'ensemble des instances associées à l'exécution $e \in \mathbf{E}$. En plus de représenter l'exécution, l'indice e rappelle que l'ensemble $\mathbf{I}_e$ est « exact » : ce n'est pas une approximation.

Bien entendu, chaque instruction peut comporter plusieurs (y compris zéro) *références* à la mémoire, l'une d'entre elles étant éventuellement une écriture (c.-à-d. en partie gauche). Un couple $(\imath, r)$ constitué d'une *instance* d'instruction et d'une *référence dans l'instruction* est appelé un *accès*. Pour une exécution donnée $e \in \mathbf{E}$ d'un programme, l'ensemble de tous les accès est noté $\mathbf{A}_e$. Il se partitionne en : $\mathbf{R}_e$, l'ensemble de toutes les *lectures*, c.-à-d. les accès effectuant une opération de lecture en mémoire ; et $\mathbf{W}_e$, l'ensemble de toutes les écritures, c.-à-d. les accès effectuant une opération d'écriture en mémoire. Dans le cas d'une instruction comportant une référence à la mémoire en partie gauche, on confond souvent les accès en écriture associés et les instances de l'instruction.

II.2 Modèle de programmes

Nos programmes seront écrits dans un style impératif, avec une syntaxe à la C (avec des extensions syntaxiques de C++). Les pointeurs sont autorisés, et les tableaux à plusieurs dimensions sont accédés avec la syntaxe `[`i_1`,`$\ldots$`,`i_n`]` — ce n'est pas la syntaxe du C — pour faciliter la lecture. Cette étude s'intéresse principalement aux structures du premier ordre, mais des techniques d'approximation permettent de prendre également en compte les pointeurs de fonction [Cou81, Deu90, Har89, AFL95]. Les appels récursifs, les boucles, les instructions conditionnelles, et les mécanismes d'exception sont autorisés ; on suppose en revanche que les `goto` ont été préalablement éliminés par des algorithmes de restructuration de code [ASU86, Bak77, Amm92].

Nous ne considérerons que les structures de données suivantes : les scalaires (booléens, entiers, flottants, pointeurs...), les enregistrements (ou *records*) de scalaires non récursifs, les tableaux de scalaires ou d'enregistrements, les arbres de scalaires ou d'enregistrements, les arbres de tableaux et les tableaux d'arbres (même chaînés récursivement). Pour simplifier, nous supposons que les tableaux sont toujours accédés avec leur syntaxe spécifique (l'opérateur `[]`) et que l'arithmétique de pointeurs est donc interdite. Les structures d'arbres sont accédées à l'aide de pointeurs explicites (à travers les opérateurs `*` et `->`).

La « forme » des structures de données n'est pas explicite dans les programmes C : il n'est pas évident de savoir si telle structure est une liste ou un arbre et non un graphe quelconque. Des informations supplémentaires données par le programmeur peuvent résoudre le problème [KS93, FM97, Mic95, HHN92], de même que des analyses à la compilation de la forme des structures de données [GH96, SRW96]. L'association des pointeurs à une instance donnée d'une structure d'arbre n'est pas évidente non plus : il s'agit d'un cas particulier de l'analyse d'alias [Deu94, CBC93, GH95, LRZ93, EGH94, Ste96]. Par la suite, nous supposerons que de telles techniques ont été appliquées par le compilateur.

Une question importante à propos des structures de données : comment sont-elles construites, modifiées et détruites ? La forme des tableaux est souvent connue statiquement, mais il arrive que l'on ait recours à des *tableaux dynamiques* dont la taille évolue à chaque dépassement de bornes (c'est le cas dans la section V) ; en revanche, les structures à base de pointeurs sont allouées dynamiquement avec des instructions explicites. Feautrier a étudié le problème dans [Fea98] et nous aurons la même vision : toutes les structures de données sont supposées construites jusqu'à leur extension maximale — éventuellement infinie. La correction d'une telle abstraction est garantie lorsque l'on *interdit toute insertion et toute suppression à l'exécution*. Cette règle très stricte souffre tout de même deux exceptions que nous étudierons après avoir introduit l'abstraction mathématique pour les structures de données. Il n'en reste pas moins que de nombreux programmes ne respectent malheureusement pas cette règle.

II.3 Modèle formel

Nous présentons d'abord une méthode de nommage pour les instances d'instructions, puis nous proposons une abstraction mathématique des cellules mémoire.

Nommer les instances d'instructions

Désormais, on suppose que chaque instruction porte une étiquette, l'*alphabet des étiquettes* est noté Σ_{CTRL}. Les boucles méritent une attention particulière : elles ont trois étiquettes, la première représente l'entrée dans la boucle, la deuxième correspond à la vérification de la condition, et la troisième représente l'itération [1]. De la même manière, les instructions conditionnelles ont deux labels : un pour la condition et pour la branche `then`, un autre pour la branche `else`. Nous étudierons l'exemple de la figure 5 ; cette procédure calcule toutes les solutions du problème des n reines.

```
       int A[n];
P      void Queens (int n, int k) {
I        if (k <n) {
A/A/a      for (int i=0; i<n; i++) {
B/B/b        for (int j=0; j<k; j++)
r              ··· = ··· A[j] ···;
J            if (···) {
s              A[k] = ···;
Q              Queens (n, k+1);
             }
           }
         }
       }

       int main () {
F        Queens (n, 0);
       }
```

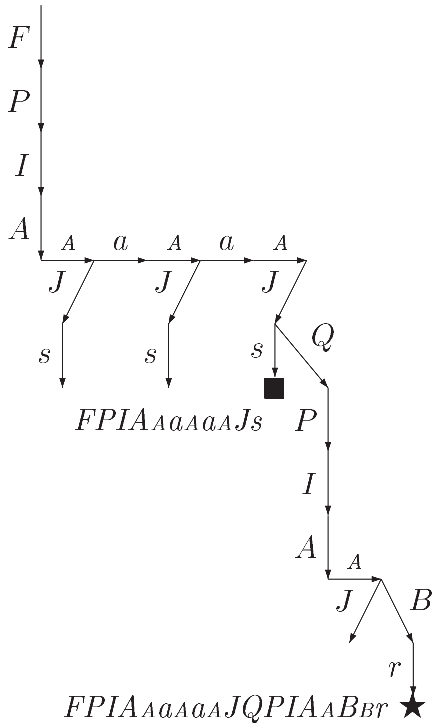


Figure 5. La procédure `Queens` et un arbre de contrôle (partiel)

Les *traces d'exécution* sont souvent utilisées pour nommer les instances à l'exécution. Elles sont généralement définies comme un chemin de l'entrée du *graphe de flot de contrôle* jusqu'à une instruction donnée. [2] Chaque exécution d'une instruction est enregistrée, y compris les retours de fonctions. Dans notre cas, les traces d'exécution ont un certain nombre d'inconvénients, le plus grave étant qu'une instance donnée peut avoir plusieurs traces d'exécution différentes en fonction de l'exécution du programme. Ce point interdit l'utilisation des traces pour donner un unique nom à chaque instance. Notre solution utilise une autre représentation de l'exécution du programme [CC98, Coh99a, Coh97, Fea98]. Pour une exécution donnée, chaque instance d'une instruction se situe à l'extrémité

1. En C, la vérification se fait juste après l'entrée dans la boucle et avant chaque itération
2. Sans se soucier des expressions conditionnelles et des bornes de boucles.

d'une *unique* liste (ordonnée) d'entrées de blocs, d'itérations de boucles et d'appels de procédures. À chaque liste correspond un certain mot : la concaténation des étiquettes des instructions. Ces concepts sont illustrés sur l'arbre de la figure 5, dont la définition est donnée ultérieurement.

Définition 1 L'*automate de contrôle* d'un programme est un *automate fini* dont les états sont les instructions et où une transition d'un état q à un état q' exprime que l'instruction q' apparaît dans le bloc q. Une telle transition est étiquetée par q'. L'état initial est la première instruction exécutée, et *tous les états sont finaux*.

Les mots acceptés par l'automate de contrôle sont appelés ***mots de contrôle***. Par construction, ils décrivent un *langage rationnel* L_{CTRL} inclus dans Σ^*_{CTRL}.

Si $\mathbf{I}$ est l'union de tous les ensembles d'instances $\mathbf{I}_e$ pour toute exécution donnée $e \in \mathbf{E}$, il y a une injection naturelle de $\mathbf{I}$ sur le langage L_{CTRL} des mots de contrôle. Ce résultat nous permet de parler du « mot de contrôle d'une instance ». En général, les ensembles $\mathbf{E}$ et $\mathbf{I}_e$ — pour une exécution donnée e — ne sont pas connus à la compilation. Nous considérerons souvent l'ensemble de toutes les instances susceptibles d'être exécutées, indépendamment des instructions conditionnelles et des bornes de boucles. Cet ensemble est en bijection avec l'ensemble des mots de contrôle. Nous parlerons donc également de « l'instance w », qui signifie « l'instance dont le mot de contrôle est w ».

On remarque que certains états n'ont qu'une transition entrante et une transition sortante. En pratique, on considère souvent un ***automate de contrôle compressé*** où tous ces états sont éliminés. Cette transformation n'a pas de conséquences sur les mots de contrôle. Les automates du programme `Queens` sont décrits sur la figure 6.

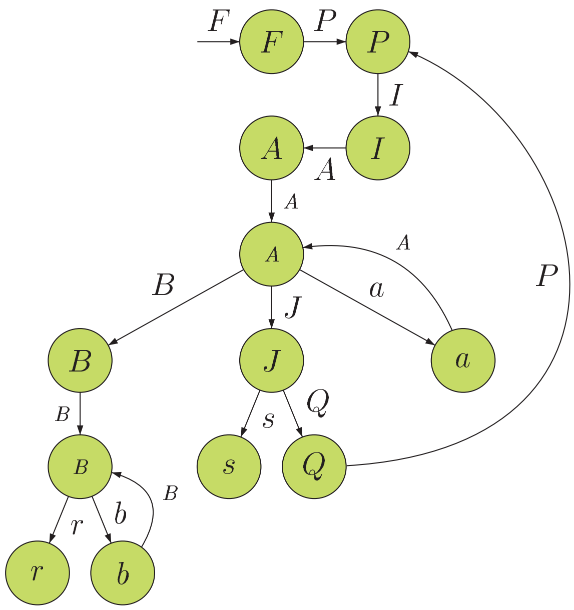


Figure 6.a. Automate de contrôle

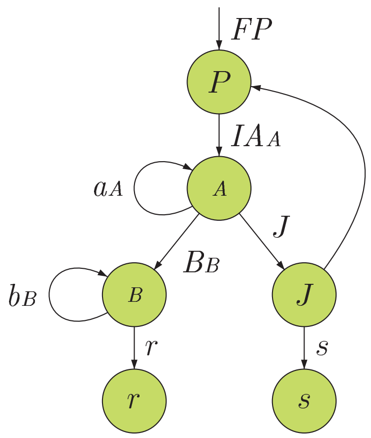


Figure 6.b. Automate de contrôle compressé pour `Queens`

Figure 6. Automates de contrôle

L'ordre d'exécution séquentiel d'un programme définit un ordre total sur les instances que l'on note $<_{\text{SEQ}}$. De plus, on peut définir un ***ordre textuel*** partiel $<_{\text{TXT}}$ sur les instructions du programme : les instructions d'un même bloc sont ordonnées selon leur apparition, et les instructions apparaissant dans des blocs différents sont incomparables. Dans le cas des boucles, l'étiquette de l'*itération* s'exécute *après* toutes les instructions du corps de boucle. Pour la procédure `Queens` on a $B <_{\text{TXT}} J <_{\text{TXT}} a$, $r <_{\text{TXT}} b$ et $s <_{\text{TXT}} Q$. Cet ordre textuel engendre un ***ordre lexicographique*** sur les mots de contrôle (ordre du dictionnaire) noté $<_{\text{LEX}}$. Cet ordre est partiel sur Σ^*_{CTRL} et sur L_{CTRL} (notamment à cause des instructions conditionnelles). Par construction de l'ordre textuel, une instance $\imath'$ s'exécute avant une instance $\imath$ si et seulement si leurs mots de contrôle w' et w respectifs vérifient $w' <_{\text{LEX}} w$.

Enfin, le langage des mots de contrôle s'interprète facilement comme un arbre infini, dont la racine est nommée ε et chaque arête est étiquetée par une instruction. Chaque nœud correspond alors au mot de contrôle égal à la concaténation des étiquettes sur la branche issue de la racine. Un tel arbre est appelé ***arbre de contrôle***. Un arbre d'appel partiel pour le programme `Queens` est donné par la figure 5.

L'adressage des cellules mémoire

Nous généralisons ici un certain nombre de formalismes que nous avions proposés précédemment [CC98, Coh99a, Coh97, Fea98, CCG96]. Celui-ci s'inspire également d'approches assez diverses [Ala94, Mic95, Deu92, LH88].

Sans surprise, les éléments de tableau sont indexés par des entiers ou des vecteurs d'entiers. L'adressage des arbres se fait en concaténant les *étiquettes des arêtes* en partant de la racine. L'adresse de la racine est donc ε et celle du nœud `root->l->r` dans un arbre binaire est lr. L'ensemble des noms d'arêtes est noté Σ_{DATA} ; la disposition des arbres en mémoire est donc décrite par un *langage rationnel* $L_{\text{DATA}} \subset \Sigma^*_{\text{DATA}}$.

Pour travailler à la fois sur les arbres et sur les tableaux, on note que ces deux structures partagent la même abstraction mathématique : le monoïde (voir Section III.2). En effet, les langages rationnels (adressage des arbres) sont des sous-ensembles de *monoïdes libres* avec la concaténation des mots, et les ensembles de vecteurs d'entiers (indexation des tableaux) sont des *monoïdes commutatifs libres* avec l'addition des vecteurs. L'abstraction d'une structure de données par un monoïde est notée M_{DATA}, et le sous-ensemble de ce monoïde associé aux éléments valides de la structure sera noté L_{DATA}.

Le cas des emboîtements d'arbres et de tableaux est un peu plus complexe, mais il révèle l'expressivité des abstractions sous forme de monoïdes. Toutefois, nous ne parlerons pas davantage de ces structures hybrides dans ce résumé en français. Par la suite, l'abstraction pour n'importe quelle structure de données de notre modèle de programmes sera un sous-ensemble L_{DATA} du monoïde M_{DATA} avec la loi $\bullet$.

Il est temps de revenir sur l'interdiction des insertions et des suppressions de la section précédente. Notre formalisme est capable en réalité de gérer les deux exceptions suivantes : puisque le flot des données ne dépend pas du fait que l'insertion d'un nœud s'effectue au début du programme ou en cours d'exécution, les *insertions en queue de liste et aux feuilles des arbres* sont permises ; lorsque des *suppressions* sont effectuées *en queue de liste ou aux feuilles des arbres*, l'abstraction mathématique est toujours correcte mais risque de conduire à des approximations trop conservatrices.

Nids de boucles et tableaux

De nombreuses applications numériques sont implémentées sous formes de nids de boucles sur tableaux, notamment en traitement du signal et dans les codes scientifiques ou multimédia. Énormément de résultats d'analyse et de transformation ont été obtenus pour ces programmes. Notre formalisme décrit sans problème ce genre de codes, mais il semble plus naturel et plus simple de revenir à des notions plus classiques pour nommer les instances et adresser la mémoire. En effet, les vecteurs d'entiers sont plus adaptés que les mots de contrôle, car les $\mathbb{Z}$-modules ont une structure beaucoup plus riche que celle de simples monoïdes commutatifs.

En utilisant des correspondances de Parikh [Par66], nous avons montré que les *vecteurs d'itérations* — le formalisme classique pour nommer les instances dans les nids de boucles — sont une interprétation particulière des mots de contrôle, et que les deux notions sont équivalentes en l'absence d'appels de procédures. Enfin, les instances d'instructions ne se réduisent pas uniquement à des vecteurs d'itération, et nous introduisons les notations suivantes (qui généralisent les notations intuitives de la section II.1) : $\langle S, x\rangle$ représente l'instance de l'instruction S dont le vecteur d'itération est x ; $\langle S, x, \texttt{ref}\rangle$ représente l'accès construit à partir de l'instance $\langle S, x\rangle$ et de la référence `ref`.

D'autres comparaisons entre vecteurs d'itération et mots de contrôle sont présentées dans la section IV.5.

II.4 Analyse par instances

La définition des exécutions d'un programme n'est pas très pratique puisque notre modèle utilise des mots de contrôle et non des traces d'exécution. Nous préférons ici utiliser une vision équivalente où l'*exécution séquentielle* $e \in \mathbf{E}$ d'un programme est un couple $(<_{\text{SEQ}}, f_e)$, où $<_{\text{SEQ}}$ est l'ordre d'exécution séquentiel sur *toutes les instances possibles* et f_e associe chaque *accès* à la cellule mémoire qu'il lit ou écrit. On remarque que $<_{\text{SEQ}}$ ne dépend pas de l'exécution, l'ordre séquentiel étant *déterministe*. Au contraire, le domaine de f_e est exactement l'ensemble $\mathbf{A}_e$ des *accès* associés à l'exécution e. La fonction f_e est appelée la *fonction d'accès* pour l'exécution e du programme [CC98, Fea98, CFH95, Coh99b, CL99]. Pour simplifier, lorsque l'on parlera du « programme $(<_{\text{SEQ}}, f_e)$ », on entendra l'ensemble des exécutions $(<_{\text{SEQ}}, f_e)$ du programme pour $e \in \mathbf{E}$.

Conflits d'accès et dépendances

Les analyses et transformations requièrent souvent des informations sur les « conflits » entre accès à la mémoire. Deux accès a et a' sont en *conflit* s'ils accèdent — en lecture où en écriture — à la même cellule mémoire : $f_e(a) = f_e(a')$.

L'analyse des conflits ressemble beaucoup à l'analyse d'*alias* [Deu94, CBC93] et s'applique également aux analyses de caches [TD95]. La *relation de conflit* — la relation entre conflits d'accès — est notée κ_e pour une exécution donnée e. Comme on ne peut généralement pas connaître exactement f_e et κ_e, l'*analyse des conflits d'accès* consiste à déterminer une *approximation conservatrice* κ de la relation de conflit qui soit compatible avec n'importe quelle exécution du programme :

$$\forall e \in \mathbf{E}, \forall v, w \in \mathbf{A}_e : \quad \big(f_e(v) = f_e(w) \implies v \,\kappa\, w\big).$$

Pour paralléliser, on a besoin de conditions suffisantes pour autoriser que deux accès s'exécutent dans un ordre quelconque. Ces conditions s'expriment en terme de *dépendances* : un accès a dépend d'un *autre* accès a' si l'un d'entre eux est une écriture, s'ils sont en conflit — $f_e(a) = f_e(a')$ — et si a' s'exécute avant a — $a' <_{\text{SEQ}} a$. La *relation de dépendance* pour une exécution e est notée δ_e : a dépend de a' est noté $a' \,\delta_e\, a$.

$$\forall e \in \mathbf{E}, \forall a, a' \in \mathbf{A}_e : \quad a' \,\delta_e\, a \stackrel{\text{def}}{\iff} (a \in \mathbf{W}_e \vee a' \in \mathbf{W}_e) \wedge a' <_{\text{SEQ}} a \wedge f_e(a) = f_e(a').$$

Une *analyse de dépendances* se contente à nouveau d'un résultat approché δ, tel que

$$\forall e \in \mathbf{E}, \forall a, a' \in \mathbf{A}_e : \quad \big(a' \, \delta_e \, a \implies a' \, \delta \, a\big).$$

Analyse de définitions visibles

Dans certains cas, on recherche une information plus précise que les dépendances : étant donné une lecture en mémoire, on veut connaître l'instance qui a produit la valeur. L'accès en lecture est appelé *utilisation* et l'instance qui a produit la valeur est appelée *définition visible*. Il s'agit en fait de la dernière instance — selon l'ordre d'exécution — en dépendance avec l'utilisation. La fonction associant son unique définition visible à chaque accès en lecture est notée σ_e :

$$\forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e : \quad \sigma_e(u) = \max_{<_{\text{SEQ}}} \big\{ v \in \mathbf{W}_e : v \, \delta_e \, u \big\}.$$

Il se peut qu'une instance en lecture n'ait en fait aucune définition visible dans le programme considéré. On ajoute donc une instance virtuelle $\bot$ qui s'exécute avant toutes les instances du programme et initialise toutes les cellules mémoire.

Lorsque l'on effectue une *analyse de définitions visibles*, on calcule une *relation* σ qui approxime de manière conservatrice les fonctions σ_e :

$$\forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e, v \in \mathbf{W}_e : \quad \big(v = \sigma_e(u) \implies v \, \sigma \, u\big).$$

On peut aussi voir σ comme une fonction qui calcule des *ensembles de définitions visibles possibles*. Lorsque $\bot$ apparaît dans un ensmble d'instances, une valeur non initialisée risque d'être lue. Cette information peut être utilisée pour vérifier les programmes.

Par la suite on aura besoin de considérer des ensembles approchés d'instances et d'accès : On a déja rencontré la notation $\mathbf{I}$ qui représente l'ensemble de toutes les instances possibles pour n'importe quelle exécution d'un programme donné :

$$\forall e \in \mathbf{E} : \quad \big(\imath \in \mathbf{I}_e \implies \imath \in \mathbf{I}\big);$$

De même, on utilisera les approximations conservatrices $\mathbf{A}$, $\mathbf{R}$ et $\mathbf{W}$ des ensembles $\mathbf{A}_e$, $\mathbf{R}_e$ et $\mathbf{W}_e$.

II.5 Parallélisation

Avec le modèle introduit par la section II.4, *paralléliser* un programme $(<_{\text{SEQ}}, f_e)$ signifie construire un programme $(<_{\text{PAR}}, f_e^{\text{EXP}})$, où $<_{\text{PAR}}$ est un *ordre d'exécution parallèle*, c'est-à-dire un ordre partiel et un sous ordre de $<_{\text{SEQ}}$. On appelle *expansion de la mémoire* le fait de construire une nouvelle fonction d'accès f_e^{EXP} à partir de f_e. Bien sûr, un certain nombre de propriétés doivent être satisfaites par $<_{\text{PAR}}$ et f_e^{EXP} afin de préserver la sémantique de l'exécution séquentielle.

L'expansion de la mémoire a pour but de réduire le nombre de dépendances superflues qui sont dues à la réutilisation des mêmes cellules mémoire. Indirectement, l'expansion met donc en évidence plus de parallélisme. On considère en effet une relation de dépendance δ_e^{EXP} pour une exécution e du programme expansé :

$$\forall e \in \mathbf{E}, \forall a, a' \in \mathbf{A}_e :$$
$$a' \, \delta_e^{\text{EXP}} \, a \overset{\text{def}}{\iff} (a \in \mathbf{W}_e \vee a' \in \mathbf{W}_e) \wedge a' <_{\text{SEQ}} a \wedge f_e^{\text{EXP}}(a) = f_e^{\text{EXP}}(a').$$

Pour définir un ordre parallèle compatible avec n'importe quelle exécution du programme, on doit considérer une approximation conservatrice δ^{EXP}. Cette approximation est en générale induite par la stratégie d'expansion (voir section V.4 par exemple).

Théorème 1 (correction d'un ordre parallèle) La condition suivante garantit que l'ordre d'exécution parallèle est correct pour le programme expansé (il préserve la sémantique du programme d'origine).

$$\forall (\imath_1, r_1), (\imath_2, r_2) \in \mathbf{A} : \quad (\imath_1, r_1)\, \delta^{\text{EXP}}\, (\imath_2, r_2) \implies \imath_1 <_{\text{PAR}} \imath_2.$$

On remarque que δ_e^{EXP} coincide avec σ_e lorsque le programme est mis en assignation unique. On supposera donc que $\delta^{\text{EXP}} = \sigma$ pour paralléliser de tels programmes.

Enfin, on ne reviendra pas ici sur les techniques utilisées pour calculer effectivement un ordre d'exécution parallèle, et pour générer le code correspondant. Les techniques de parallélisation de programmes récursifs sont relativement récentes et seront étudiées dans la section 5.5. En ce qui concerne les méthodes associées aux nids de boucles, de nombreux algorithmes d'*ordonnancement* et de *partitionnement* — ou de *pavage* (*tiling*) — ont été proposés ; mais leur description ne paraît pas indispensable à la bonne compréhension des techniques étudiées par la suite.

III Outils mathématiques

Cette section rassemble les rappels et les contributions portant sur les abstractions mathématiques que nous utilisons. Le lecteur intéressé par les techniques d'analyse et de transformation peut se contenter de noter les définitions et théorèmes principaux.

III.1 Arithmétique de Presburger

Nous avons besoin de manipuler des ensembles, des fonctions et des relations sur des vecteurs d'entiers. L'arithmétique de Presburger nous convient particulièrement puisque la plupart des questions intéressantes sont décidables dans cette théorie. On la définit à partir des formules logiques construites sur $\forall$, $\exists$, $\neg$, $\vee$, $\wedge$, l'égalité et l'inégalité de contraintes affines entières. La satisfaction d'une formule de Presburger est au cœur de la plupart des calculs symboliques avec des contraintes affines : c'est un problème NP-complet de *programmation linéaire en nombres entiers* [Sch86]. Les algorithmes utilisés sont super-exponentiels dans le pire cas [Pug92, Fea88b, Fea91], mais d'une grande efficacité pratique sur des problèmes de taille moyenne.

Nous utilisons principalement Omega [Pug92] dans nos expérimentations et implémentations de prototypes ; la syntaxe des ensembles, relations et fonctions étant très proche des notations mathématiques usuelles. PIP [Fea88b] — l'outil *paramétrique* de programmation linéaire en nombre entiers — utilise une autre représentation pour les relations affines : la notion d'*arbre de sélection quasi-affine* ou *quasi-affine selection tree*, plus simplement appelé *quast*.

Définition 2 (quast) Un quast représentant une relation affine est une expression conditionnelle à plusieurs niveaux, dans laquelle les prédicats sont des tests sur le signe de formes quasi-affines [3] et les feuilles sont des ensembles de vecteurs décrits dans l'arith-

3. Les formes quasi-affines étendent les formes affines avec des divisions entières par des constantes et des restes de telles divisions.

métique de Presburger étendue avec $\perp$ — qui précède tout autre vecteur pour l'ordre lexicographique.

Lorsque des ensembles vides apparaissent dans les feuilles, ils diffèrent du singleton $\{\perp\}$ et décrivent les vecteurs qui ne sont pas dans le domaine de la relation. Des exemples seront donnés dans la section V.

Une opération classique sur les relations consiste à déterminer la clôture transitive. Les algorithmes classiques ne considèrent que des graphes *finis*. Malheureusement, dans le cas des relations affines, il se trouve que la clôture d'une relation affine n'en est généralement pas une.

Nous utiliserons donc des techniques d'approximation développées par Kelly et al. et implémentées dans Omega [KPRS96]. L'idée générale consiste à se ramener à une sous-classe par approximation, puis de calculer exactement la clôture.

III.2 Langages formels et relations rationnelles

Certains concepts font partie du fond commun en informatique théorique, comme les *monoïdes*, les *langages rationnels* et *algébriques*, les *automates finis*, et les *automates à pile*. Les ouvrages de référence sont [HU79] et [RS97a], mais il existe également de nombreuses introductions en français. Nous nous contenterons donc de fixer les notations utilisées par la suite, à l'aide d'un exemple classique. Dans un deuxième temps, nous étudierons des objets mathématiques plus originaux : nous présenterons les résultats essentiels sur la classe des *relations rationnelles* entre monoïdes de type fini.

Langages formels : exemple et notations

Le langage de Lukasiewicz est un exemple simple de *langage à un compteur* — c.-à-d. reconnu par un automate à un compteur — sous-classe des *langages algébriques*. Le langage de Lukasiewicz Ł sur un alphabet $\{a, b\}$ est engendré par l'axiome ξ et la *grammaire* dont les productions sont

$$\xi \longrightarrow a\xi\xi \mid b.$$

Ce langage est apparenté aux langages de Dyck [Ber79], ses premiers mots étant

$$b, abb, aabbb, ababb, aaabbbb, aababbb, \ldots$$

L'encodage d'un compteur sur une pile se fait de la façon suivante : trois symboles sont utilisés, Z est le symbole de fond de pile, I code les nombres positifs, et D les code nombres négatifs ; ZI^n représente donc l'entier n, ZD^n représente $-n$, et Z code la valeur 0 du compteur. La figure 7 décrit un automate à pile acceptant le langage Ł ainsi que son interprétation en termes de compteur.

Une généralisation naturelle des langages à un compteur consiste à en mettre plusieurs : il s'agit alors d'une machine de Minsky [Min67]. Cependant, les automates à deux compteurs ont déjà le même pouvoir d'expression que les machines de Turing, et la plupart des questions intéressantes deviennent donc indécidables. Pourtant, en imposant quelques restrictions sur la famille des langages à plusieurs compteurs, des résultats de décidabilité récents ont été obtenus. L'étude de ces objets parait riche en applications, notamment dans le cas des travaux de Comon et Jurski [CJ98].

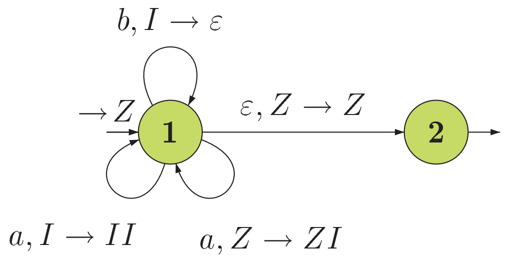


Figure 7.a. Automate à pile

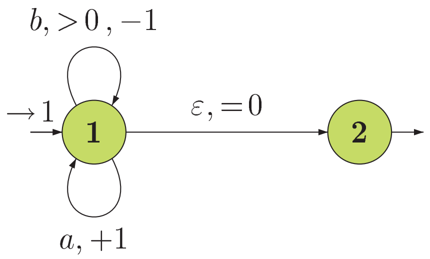


Figure 7.b. Automate à un compteur associé

Figure 7. Exemples d'automates

Relations rationnelles

Nous nous contentons de quelques rappels ; consulter [AB88, Eil74, Ber79] pour de plus amples détails. Soit M un monoïde. Un sous-ensemble R de M est un ***ensemble reconnaissable*** s'il existe a monoïde fini N, un morphisme α de M dans N et un sous-ensemble P de N tels que $R = \alpha^{-1}(P)$.

Ces ensembles généralisent les langages rationnels tout en conservant la structure d'*algèbre booléenne* : en effet, la classe des ensembles reconnaissables est close pour l'union, l'intersection et le complémentaire. Les ensembles reconnaissables sont également clos pour la concaténation, mais pas pour l'opération *étoile*. C'est le cas en revanche de la classe des ensembles rationnels, dont la définition étend celle des langages rationnels : soit M un monoïde, la classe des ***ensembles rationnels*** de M est la plus petite famille de sous-ensembles de M comportant $\varnothing$ et les singletons $\{m\} \subset M$, close pour l'union, la concaténation et l'opération étoile.

En général, les ensembles rationnels ne sont pas clos pour le complémentaire et l'intersection. Si M est de la forme $M_1 \times M_2$, où M_1 et M_2 sont deux monoïdes, un sous-ensemble reconnaissable de M est appelé ***relation reconnaissable***, et un sous-ensemble rationnel de M est appelé ***relation rationnelle***. Le résultat suivant décrit la « structure » des relations reconnaissables.

Théorème 2 (Mezei) Une relation reconnaissable $R \subset M_1 \times M_2$ est une union finie d'ensembles de la forme $K \times L$ où K et L sont des ensembles rationnels de M_1 et M_2.

Par la suite nous ne considérerons que des ensembles reconnaissables et rationnels qui sont des relations entre monoïdes de type fini.

Les ***transductions*** donnent une vision « plus fonctionnelle » des relations reconnaissables et rationnelles. À partir d'une relation R entre des monoïdes M_1 et M_2, on définit une transduction τ de M_1 dans M_2 comme une fonction de M_1 dans l'ensemble $\mathfrak{P}(M_2)$ des parties de M_2, telle que $v \in \tau(u)$ ssi uRv. Une transduction est reconnaissable (resp. rationnelle) ssi son graphe est une relation reconnaissable (resp. rationnelle). Ces deux classes sont closes pour l'inversion, et la classe des transductions reconnaissables est également close pour la composition.

Celle des transductions rationnelles est également close pour la composition dans le cas de monoïdes *libres* : c'est le théoreme de Elgot et Mezei [EM65, Ber79], fondamental pour l'analyse de dépendances (section IV).

Théorème 3 (Elgot and Mezei) Si A, B et C sont des alphabets, $\tau_1 : A^* \to B^*$ et

$\tau_2 : B^* \to C^*$ sont des transductions rationnelles, alors $\tau_2 \circ \tau_1 : A^* \to C^*$ est une transduction rationnelle.

La représentation « mécanique » des relations et transductions rationnelles est appelée transducteur rationnel ; ceux-ci étendent naturellement les automates finis en ajoutant un « ruban de sortie » :

Définition 3 (transducteur rationnel) Pour un monoïde « d'entrée » M_1 et un monoïde « de sortie » M_2 [4], on définit un transducteur rationnel $\mathcal{T} = (M_1, M_2, Q, I, F, E)$ avec un ensemble fini d'états Q, un ensemble d'états initaux $I \subset Q$, an ensemble d'états finaux $F \subset Q$, et un ensemble fini de transitions (ou arêtes) $E \subset Q \times M_1 \times M_2 \times Q$.

Le théorème de Kleene assure que les relations rationnelles de $M_1 \times M_2$ sont exactement les relations reconnues par un transducteur rationnel. On note $|\mathcal{T}|$ la transduction reconnue par le transducteur $\mathcal{T}$: on dit que $\mathcal{T}$ *réalise* la transduction $|\mathcal{T}|$. Lorsque les monoïdes M_1 et M_2 sont *libres*, l'élément neutre est le mot vide noté ε.

Théorème 4 Les problèmes suivants sont décidables pour les relations rationnelles : est-ce que deux mots sont en relation (en temps linéaire), la vacuité, la finitude.

Soient R et R' deux relations rationnelles sur des alphabets A et B avec au moins deux lettres. Il n'est pas décidable de savoir si $R \cap R' = \varnothing$, $R \subset R'$, $R = R'$, $R = A^* \times B^*$, $(A^* \times B^*) - R$ est fini, R est reconnaissable.

Quelques résultats intéressants concernent les transductions qui sont des fonctions partielles. Une *fonction rationnelle* $\psi : M_1 \to M_2$ est une transduction rationnelle qui est une fonction partielle, c.-à-d. telle que $\mathrm{Card}(\psi(u)) \leq 1$ pour tout $u \in M_1$. Étant donnés deux alphabets A et B, il est décidable qu'une transduction rationnelle de A^* dans B^* est une fonction partielle (en $\mathcal{O}(\mathrm{Card}(Q)^4)$ [Ber79, BH77]). On peut également décider si une fonction rationnelle est incluse dans une autre et si elles sont égales.

Parmi les transducteurs réalisant des fonctions rationnelles, on s'intéresse notamment à ceux que l'on peut « calculer à la volée » en lisant leur entrée. Soient A et B deux alphabets. Un transducteur est *séquentiel* lorsqu'il est étiqueté sur $A \times B^*$ et que son *automate d'entrée* (obtenu en omettant les sorties) est déterministe. Un transducteur séquentiel réalise une fonction rationnelle. Cette notion de « calcul à la volée » est un peu trop restrictive, on considère plutôt l'extension suivante :

Définition 4 (transducteur sous-séquentiel) Pour deux alphabets A et B, un transducteur *sous-séquentiel* $(\mathcal{T}, \rho)$ sur $A^* \times B^*$ est un couple où $\mathcal{T}$ est un transducteur séquentiel avec F pour ensemble d'états finaux, et où $\rho : F \to B^*$ est une fonction. La fonction ψ réalisée par $(\mathcal{T}, \rho)$ est définie comme suit : si $u \in A^*$, la valeur $\psi(u)$ est définie s'il existe un chemin dans $\mathcal{T}$ acceptant $(u|v)$ aboutissant à un état final q ; dans ce cas $\psi(u) = v\rho(q)$.

En d'autres termes, ρ ajoute un mot à la fin de la sortie d'un transducteur séquentiel. Partant d'une démonstration de Choffrut [Cho77], Béal et Carton [BC99b] ont proposé un algorithme polynomial pour décider si une fonction rationnelle est sous-séquentielle, et un autre pour décider si une sous-séquentielle est séquentielle. Ils ont également proposé un algorithme polynomial pour trouver une réalisation sous-séquentielle d'une fonction rationnelle, lorsqu'elle existe.

4. Les monoïdes M_1 et M_2 sont souvent omis de la définition.

III.3 Relations synchrones à gauche

Les relations rationnelles ne sont pas closes pour l'intersection, mais cette opération est indispensable dans le cadre de l'analyse de dépendances. Feautrier [Fea98] a proposé un « semi-algorithme » pour répondre à la question indécidable de la vacuité d'une intersection de relations rationnelles : l'algorithme ne termine à coup sûr que lorsque l'intersection n'est pas vide. Puisque nous voulons calculer cette intersection, nous adoptons une approche différente : on se ramène — par approximations conservatrices — à une classe de relations rationnelles avec une structure d'*algèbre booléenne* (c.-à-d. avec l'union, l'intersection et le complémentaire).

Les relations reconnaissables constituent bien une algèbre booléene, mais nous avons construit une classe plus générale : les *relations synchrones à gauche*. Cette classe a été étudiée indépendamment par Frougny et Sakarocitch [FS93], mais notre représentation est différente, les preuves sont nouvelles et de nouveaux résultats ont été obtenus. Ce travail est le résultat d'une collaboration avec Olivier Carton (Université de Marne-la-Vallée).

On rappelle une définition classique, équivalente à la propriété de préservation de la longueur pour les mots d'entrée et de sortie : Un transducteur rationnel sur des alphabets A et B est *synchrone* s'il est étiqueté sur $A \times B$. Nous étendons cette notion de la façon suivante.

Définition 5 (synchronisme à gauche) Un transducteur rationnel sur des alphabels A et B est *synchrone à gauche* s'il est étiqueté sur $(A \times B) \cup (A \times \{\varepsilon\}) \cup (\{\varepsilon\} \times B)$ et seules des transitions étiquetées sur $A \times \{\varepsilon\}$ (resp. $\{\varepsilon\} \times B$) peuvent *suivre* des transitions étiquetées sur $A \times \{\varepsilon\}$ (resp. $\{\varepsilon\} \times B$).

Une relation ou une transduction rationnelle est synchrone à gauche si elle peut être réalisée par un transducteur synchrone à gauche. Un transducteur rationnel est *synchronisable à gauche* s'il réalise une relation synchrone à gauche.

La figure 8 montre des transducteurs synchrones à gauche sur un alphabet A qui réalisent l'ordre préfixe et l'ordre lexicographique ($<_{\text{TXT}}$ est un ordre particulier sur A).

Pour les transducteurs suivants, x et y remplacent respectivement $\forall x \in A$ et $\forall y \in A$.

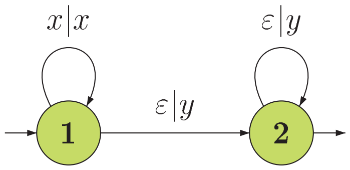


Figure 8.a. Ordre préfixe

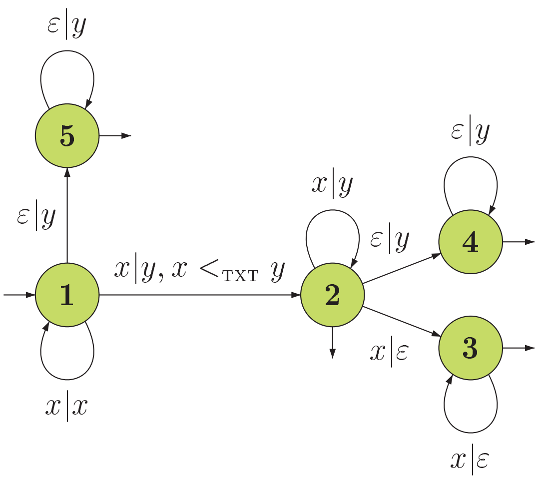


Figure 8.b. Ordre lexicographique

Figure 8. Exemple de transducteurs synchrones à gauche

Il est connu que les transducteurs synchrones constituent une algèbre booléenne [5].

Théorème 5 La classe des relations synchrones à gauche constitue une algèbre booléenne : elle est close pour l'union, l'intersection et le complémentaire. De plus, les relations reconnaissables sont synchrones à gauche ; si S est synchrone et T est synchrone à gauche, alors ST est synchrone à gauche ; si T est synchrone à gauche et R est reconnaissable, alors TR est synchrone à gauche. Enfin, la classe des relations synchrones à gauche est close pour la composition.

Les relations synchrones sont décidables parmi les relations rationnelles [Eil74] , mais ce n'est pas le cas des relations reconnaissables [Ber79] et nous avons montré qu'il en est de même des relations synchrones à gauche.

On s'intéresse cependant à certains cas particuliers pour lesquels une relation rationnelle peut être prouvée synchrone à gauche. À cet effet, on rappelle la notion de ***taux de transmission*** d'un chemin étiqueté par (u, v) : il s'agit du rapport $|v|/|u| \in \mathbb{Q}^+ \cup \{+\infty\}$. Si $\mathcal{T}$ est un transducteur synchrone à gauche, les cycles de $\mathcal{T}$ ne peuvent avoir que trois taux de transmission possibles : 0, 1 et $+\infty$. Tous les cycles d'une même composante fortement connexe doivent avoir le même taux de transmission, seuls les composants de taux 0 peuvent suivre ceux de taux 0, et seuls les composants de taux $+\infty$ peuvent suivre ceux de taux $+\infty$. Il existe une réciproque partielle :

Théorème 6 Si le taux de transmission de chaque cycle d'un transducteur rationnel est 0, 1 ou $+\infty$, et si aucun cycle de taux 1 suit un cycle de taux différent de 1, alors le transducteur est *synchronisable à gauche.*

Nous pouvons donc "resynchroniser" une certaine classe de transducteurs synchronisables à gauche, à savoir les transducteurs satisfaisant les hypothèses du théorème 6. En se fondant sur un algorithme de Béal et Carton [BC99a], on peut écrire un algorithme de resynchronisation pour calculer des approximations synchrones à gauche de relations rationnelles. Cette technique sera utilisée dans la section III.5.

Nous terminons sur des propriétés de décidabilité, essentielles pour l'analyse de dépendances et de définitions visibles.

Lemme 1 Soient R et R' des relations synchrones à gauche sur des alphabets A et B. Il est décidable que $R \cap R' = \varnothing$, $R \subset R'$, $R = R'$, $R = A^* \times B^*$, $(A^* \times B^*) - R$ est fini.

Nous travaillons toujours sur la décidabilité des relations reconnaissables parmi les synchrones à gauche.

III.4 Dépasser les relations rationnelles

Nous avons parfois besoin d'une puissance d'expression supérieure à celle des relations rationnelles. Nous utiliserons donc la notion de *relation algébrique* — où hors-contexte — qui étend naturellement celle de langage algébrique. Ces relations sont définies à partir des transducteurs à pile :

Définition 6 (transducteur à pile) Étant donnés deux alphabets A et B, un ***transducteur à pile*** $\mathcal{T} = (A^*, B^*, \Gamma, \gamma_0, Q, I, F, E)$ est constitué d'un alphabet de pile Γ [6], un mot *non vide* γ_0 dans Γ^+ appelé mot de pile initial, un ensemble fini d'états Q, un

5. Toutes les propriétés étudiées dans cette section ont des preuves constructives.
6. Les alphabets A et B sont souvent omis de la définition.

ensemble $I \subset Q$ d'états initiaux, un ensemble $F \subset Q$ d'états finaux, et un ensemble fini de transitions (où arêtes) $E \subset Q \times A^* \times B^* \times \Gamma \times \Gamma^* \times Q$.

La notion de transducteur à pile *réalisant une relation* est définie de la même manière que celle d'automate à pile réalisant un langage.

Définition 7 (relation algébrique) La classe des relations réalisées par des transducteurs à pile est appelée classe des *relations algébriques*.

Bien entendu, les *transductions algébriques* constituent la vision fonctionnelle des relations algébriques.

Théorème 7 Les relations algébriques sont closes pour l'union, la concaténation et l'opération étoile. Elles sont également closes pour la composition avec des transductions rationnelles. L'image d'un langage rationnel par une transduction algébrique est un langage algébrique.

Les questions suivantes sont décidables pour les relations algébriques : est-ce que deux mots sont en relation (en temps linéaire), la vacuité, la finitude.

Il y a très peu de résultats sur les transductions algébriques qui sont des fonctions partielles, appelées *fonctions algébriques*. En particulier, nous ne connaissons pas de sous-classe de ces fonctions qui soit « calculable à la volée » au sens des fonctions sous-séquentielles.

Néanmoins, une sous-classe intéressante des relations algébriques est celle des *relations à un compteur*, réalisées par un *transducteur à un compteur* — définition semblable à celle d'un automate à un compteur. On peut également considérer plus d'un compteur, mais l'on obtient alors la même puissance d'expression que les machines de Turing. Cette classe nous intéresse lorsque nous sommes amenés à composer des transductions *rationnelles* entre monoïdes non libres (le théorème de Elgot et Mezei ne s'applique plus).

Théorème 8 Soient A et B deux alphabets et n un entier positif. Si $\tau_1 : A^* \to \mathbb{Z}^n$ et $\tau_2 : \mathbb{Z}^n \to B^*$ sont des transductions rationnelles, alors $\tau_2 \circ \tau_1 : A^* \to B^*$ est une transduction à n compteurs.

Ce théorème sera utilisé pour l'analyse de dépendances, principalement avec $n = 1$. De plus, on peut déduire un résultat important de la preuve du théorème :

Proposition 1 Soient A et B deux alphabets et n un entier positif. Soient $\tau_1 : A^* \to \mathbb{Z}^n$ et $\tau_2 : \mathbb{Z}^n \to B^*$ des transductions rationnelles et $\mathcal{T}$ un transducteur à n compteurs réalisant $\tau_2 \circ \tau_1 : A^* \to B^*$ (calculé avec le théorème 8. Alors, le *transducteur rationnel sous-jacent* à $\mathcal{T}$ — obtenu en omettant les manipulations de pile — est *reconnaissable*.

Ce résultat garantit la clôture pour l'intersection avec n'importe quelle transduction rationnelle, d'après le résultat suivant :

Proposition 2 Soit R_1 une relation algébrique réalisée par un transducteur à pile dont le transducteur rationnel sous-jacent est *synchrone à gauche*, et soit R_2 une relation *synchrone à gauche*. Alors $R_1 \cap R_2$ est une relation algébrique, et on peut construire un transducteur à pile qui la réalise dont le transducteur rationnel sous-jacent est *synchrone à gauche*.

Enfin, le théorème 8 s'étend aux monoïdes partiellement commutatifs libres associés aux emboîtements d'arbres et de tableaux, que nous n'abordons pas dans ce résumé.

III.5 Compléments sur les approximations

Le calcul d'intersection est très utilisé dans le cadre de nos techniques d'analyse et de transformation de programmes. Les relations rationnelles et algébriques ne sont pas closes pour cette opération ; mais nous avons identifié des sous-classes qui le sont. Nous montrons ici comment s'y ramener en appliquant des approximations conservatrices.

Plusieurs méthodes permettent d'approcher des relations rationnelles par des relations reconnaissables. L'idée générale consiste à considérer le produit cartésien de l'entrée et de la sortie. Des techniques plus précises consistent à effectuer cette opération pour chaque couple d'un état initial et d'un état final, et pour chaque composante fortement connexe. Le résultat est toujours une relation reconnaissable, grâce au théorème 2.

L'approximation par des relations synchrones à gauche est fondée sur l'algorithme de resynchronisation, et donc sur le théorème 6. Lorsque l'algorithme échoue, on remplace une composante fortement connexe par une approximation reconnaissable et on recommence. Des optimisations permettent de n'appliquer qu'une seule fois l'algorithme de resynchronisation.

L'approximation de relations algébriques — où à plusieurs compteur — peut se faire de deux manières : soit on approxime la pile — ou les compteurs — par des états supplémentaires, soit on approxime le transducteur rationnel sous-jacent par un transducteur synchrone à gauche. Les deux techniques seront utilisées par la suite.

IV Analyse par instance pour programmes récursifs

Après un certain nombre de travaux sur l'analyse par instances de programmes récursifs [CCG96, Coh97, Coh99a, Fea98, CC98], nous présentons une évolution majeure avec un formalisme plus général et une automatisation complète du processus. Au delà de l'objectif théorique d'obtenir le maximum de précision possible, nous verrons dans la section V.5 comment ces informations permettent d'améliorer les techniques de parallélisation automatique de programmes récursifs.

En partant d'exemples réels, nous discutons du calcul de variables d'induction puis nous présentons les analyses de dépendances et de définitions visibles proprement dites. Cette section se termine sur une comparaison avec les analyses statiques et avec les travaux récents portant sur l'analyse par instances de nids de boucles.

IV.1 Exemples introductifs

Nous étudions deux exemples pour donner un aperçu intuitif de notre analyse par instances pour structures récursives. Un troisième exemple est présenté dans la thèse, mais il utilise une structure hybride entre arbres et tableaux dont nous ne parlons pas ici.

Premier exemple: le programme `Queens`

Nous considérons à nouveau la procédure `Queens` présentée dans la section II.3. Le programme est reproduit sur la figure 9 avec un arbre de contrôle partiel.

Nous étudions les dépendances entre les *instances à l'exécution* des instructions. Observons par exemple l'instance $FPIA\textsc{a}a\textsc{a}a\textsc{a}JQPIA\textsc{a}B\textsc{b}r$ de l'instruction r, représentée par une étoile sur la figure 9.b. La variable `j` est initialisée à 0 par l'instruction B et incrémentée par l'instruction b, nous savons donc que la valeur de `j` en $FPIA\textsc{a}a\textsc{a}a\textsc{a}JQPIA\textsc{a}B\textsc{b}r$ est 0 ;

```
      int A[n];
P     void Queens (int n, int k) {
I       if (k <n) {
A/A/a     for (int i=0; i<n; i++) {
B/B/b       for (int j=0; j<k; j++)
r             ··· = ··· A[j] ···;
J           if (···) {
s             A[k] = ···;
Q             Queens (n, k+1);
            }
          }
        }
      }

      int main () {
F       Queens (n, 0);
      }
```

Figure 9.a. Procédure `Queens`

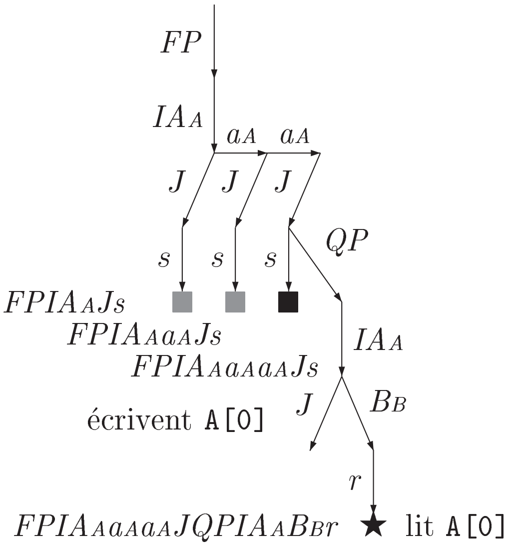


Figure 9.b. Arbre de contrôle (compressé)

Figure 9. La procédure `Queens` et un arbre de contrôle

donc $FPIA_Aa_Aa_AJQPIA_AB_Br$ lit `A[0]`. Observons à présent les instances de s, représentées par des carrés. La variable `k` est initialisée à 0 lors du premier appel à `Queens`, puis elle est incrémentée par l'appel récursif Q. Les instances $FPIA_AJs$, $FPIA_Aa_AJs$ et $FPIA_Aa_Aa_AJs$ écrivent donc dans `A[0]`, et sont ainsi en dépendance avec$FPIA_Aa_Aa_AJQPIA_AB_Br$.

Laquelle de ces définitions atteint elle $FPIA_Aa_Aa_AJQPIA_AB_Br$? En observant la figure a nouveau, on remarque que l'instance $FPIA_Aa_Aa_AJs$ — le carré noir — s'exécute en dernier. De plus, on peut assurer que cette instance est exécutée lorsque la lecture $FPIA_Aa_Aa_AJQPIA_AB_Br$ s'exécute. Les autres écritures sont donc écrasées par $FPIA_Aa_Aa_AJs$ qui est ainsi la définition visible de $FPIA_Aa_Aa_AJQPIA_AB_Br$. Nous montrerons ultérieurement comment généraliser cette approche intuitive.

Deuxième exemple : le programme `BST`

Considérons à présent la procédure `BST` de la figure 10. Cette procédure échange les valeurs des nœuds pour convertir un arbre binaire en arbre binaire de recherche, ou *binary search tree*. Les nœuds de l'arbre sont référencés par des pointeurs, et `p->value` contient la valeur entière du nœud. Il y a peu de dépendances sur ce programme : les seules sont des anti-dépendances entre certaines instances d'instructions à l'intérieur des blocs I_1 ou I_2. Par conséquent, l'analyse de définition visible donne un résultat très simple : la seule définition visible de tout accès en lecture est $\perp$.

IV.2 Relier instances et cellules mémoire

On a définit dans la section II.4 la notion de *fonction d'accès*. Celle-ci relie les *accès* aux cellules mémoire qu'ils lisent ou écrivent. Nous avons désormais besoin d'expliciter ces fonctions, et nous introduisons pour cela la notion de *variable d'induction*. En présence de

```
P   void BST (tree *p) {
I1    if (p->l!=NULL) {
L       BST (p->l);
I2      if (p->value <p->l->value) {
a         t = p->value;
b         p->value = p->l->value;
c         p->l->value = t;
        }
      }
J1    if (p->r!=NULL) {
R       BST (p->r);
J2      if (p->value >p->r->value) {
d         t = p->value;
e         p->value = p->r->value;
f         p->r->value = t;
        }
      }
    }

    int main () {
F     if (root!=NULL) BST (root);
    }
```

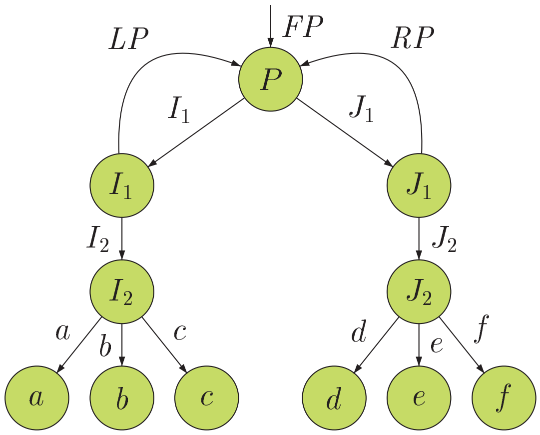


Figure 10. Procédure `BST` et automate de contrôle (compressé)

procédures récursives, cette notion historiquement liée aux nids de boucles [Wol92] doit être redéfinie. Pour simplifier l'exposition, nous supposons que chaque variable possède un *nom distinctif unique* ; on pourra ainsi parler sans ambiguïté de « la variable `i` ». Notre définition des *variables d'induction* est la suivante :

- les arguments *entiers* d'une fonction qui sont initialisés par une constante ou par une variable entière d'induction plus une constante, à chaque appel récursif ;

- les compteurs de boucle *entiers* translatés d'une constante à chaque itération ;

- les arguments de type *pointeur* qui sont initialisés par une constante ou par une variable d'induction de type *pointeur* éventuellement déréférencée.

L'analyse requiert un certain nombre d'hypothèses supplémentaires sur le modèle de programme de la section II.2 : les structures de données analysées doivent être déclarées *globales* ; les indices de tableaux doivent être des fonctions affines des *variables d'induction* entières et de constantes symboliques ; et les accès aux arbres doivent déréférencer une *variable d'induction* de type pointeur ou une constante.

Préalablement à l'analyse de dépendances, nous devons calculer les fonctions d'accès afin de décrire les conflits éventuels. Soit α une instruction et w une instance de α. La *valeur de la variable `i` à l'instance* w est définie comme la valeur de `i` immédiatement *après* exécution de l'instance w de l'instruction α. Cette valeur est notée $[\![\mathtt{i}]\!](w)$.

En général, la valeur d'une variable en un mot de contrôle donné dépend de l'exécution. Pourtant, grâce aux restrictions que nous avons imposées au modèle de programme, les *variables d'induction* sont complètement déterminées par les *mots de contrôle*. On montre que pour deux exécutions différentes e et e', les valeurs de deux variables d'induction sont identiques sur en un mot de contrôle donné. Les fonctions d'accès pour différentes exécutions coïncident donc, et nous considérerons donc par la suite une fonction d'accès f *indépendante de l'exécution*.

Le résultat suivant montre que les variables d'induction sont décrites par des *équations récurrentes* :

Lemme 2 On considère le monoïde $(M_{\text{DATA}}, \bullet)$ qui abstrait la structure de données considérée, une instruction α, et une variable d'induction `i`. L'effet de l'instruction α sur la valeur de `i` est décrit par l'une des équations suivantes :

$$\begin{aligned} \text{ou bien} \quad \exists \beta \in M_{\text{DATA}}, \mathtt{j} \in \text{INDUC}: \quad & \forall u\alpha \in L_{\text{CTRL}} : [\![\mathtt{i}]\!](u\alpha) = [\![\mathtt{j}]\!](u) \bullet \beta \\ \text{ou alors} \quad \exists \beta \in M_{\text{DATA}}: \quad & \forall u\alpha \in L_{\text{CTRL}} : [\![\mathtt{i}]\!](u\alpha) = \beta \end{aligned}$$

où INDUC est l'ensemble des variables d'induction du programme, y compris `i`.

Le résultat sur la procédure `Queens` est le suivant. On ne s'intéresse qu'aux variables inductives `i` et `k`, seules utiles pour l'analyse de dépendances.

$$\begin{aligned} \text{De l'appel principal } F: \quad & [\![\text{Arg}(\mathtt{Queens}, 2)]\!](F) = 0 \\ \text{De la procédure } P: \quad & \forall uP \in L_{\text{CTRL}} : [\![\mathtt{k}]\!](uP) = [\![\text{Arg}(\mathtt{Queens}, 2)]\!](u) \\ \text{De l'appel récursif } Q: \quad & \forall uQ \in L_{\text{CTRL}} : [\![\text{Arg}(\mathtt{Queens}, 2)]\!](uQ) = [\![\mathtt{k}]\!](u) + 1 \\ \text{De l'entrée de boucle } B: \quad & \forall uB \in L_{\text{CTRL}} : [\![\mathtt{j}]\!](uB) = 0 \\ \text{De l'itération de boucle } b: \quad & \forall ub \in L_{\text{CTRL}} : [\![\mathtt{j}]\!](ub) = [\![\mathtt{j}]\!](u) + 1 \end{aligned}$$

$\text{Arg}(\mathtt{proc}, num)$ représente le num^{e} argument *effectif* d'une procédure `proc`, et toutes les autres instructions laissent les variables inchangées.

On a conçu un algorithme pour construire automatiquement un tel système décrivant l'évolution des variables d'induction dans un programme. Combiné avec le résultat suivant, cet algorithme permet de construire automatiquement la fonction d'accès.

Théorème 9 La fonction d'accès f — qui associe chaque accès possible dans $\mathbf{A}$ à la cellule mémoire qu'il lit ou écrit — est une fonction rationnelle de Σ^*_{CTRL} dans M_{DATA}.

Le résultat pour le programme `Queens` est le suivant :

$$\begin{aligned} \big\{(ur|f(ur, \mathtt{A[j]}))\big\} &= (FPIA_A|0) \cdot \big((JQPIA_A|0) + (a_A|0)\big)^* \cdot (B_B|0) \cdot (b_B|1)^* \cdot (r|0) \\ \big\{(us|f(us, \mathtt{A[k]}))\big\} &= (FPIA_A|0) \cdot \big((JQPIA_A|1) + (a_A|0)\big)^* \cdot (Js|0) \end{aligned}$$

On a appliqué la même technique au programme `BST` :

$$\begin{aligned} & \forall \alpha \in \{I_2, a, b\} : \\ \big\{(u\alpha|f(u\alpha, \mathtt{p\text{->}value}))\big\} &= (FP|\varepsilon) \cdot \big((I_1LP|l) + (J_1RP|r)\big)^* \cdot (I_1I_2\alpha|\varepsilon) \\ & \forall \alpha \in \{I_2, b, c\} : \\ \big\{(u\alpha|f(u\alpha, \mathtt{p\text{->}l\text{->}value}))\big\} &= (FP|\varepsilon) \cdot \big((I_1LP|l) + (J_1RP|r)\big)^* \cdot (I_1I_2\alpha|l) \\ & \forall \alpha \in \{J_2, d, e\} : \\ \big\{(u\alpha|f(u\alpha, \mathtt{p\text{->}value}))\big\} &= (FP|\varepsilon) \cdot \big((I_1LP|l) + (J_1RP|r)\big)^* \cdot (J_1J_2\alpha|\varepsilon) \\ & \forall \alpha \in \{J_2, e, f\} : \\ \big\{(u\alpha|f(u\alpha, \mathtt{p\text{->}r\text{->}value}))\big\} &= (FP|\varepsilon) \cdot \big((I_1LP|l) + (J_1RP|r)\big)^* \cdot (J_1J_2\alpha|r) \end{aligned}$$

IV.3 Analyse de dépendances et de définitions visibles

À l'aide des fonctions d'accès, notre premier objectif consiste à calculer la relation entre les accès conflictuels à la mémoire. Nous ne pouvons pas espérer un résultat exact en général, mais on peut profiter du fait que la fonction d'accès f ne dépend pas de l'exécution. La relation de conflit approchée que nous calculons est la suivante :

$$\forall u, v \in L_{\text{CTRL}} : \quad u \,\kappa\, v \overset{\text{def}}{\iff} v \in f^{-1}(f(v)).$$

D'après le théorème de Elgot et Mezei (section III.2) et le théorème 8, la composition de f^{-1} et de f est soit une transduction rationnelle soit une transduction à plusieurs compteurs. Le nombre de compteurs correspond à la dimension du tableau accédé, et on peut se ramener à un seul compteur par une approximation conservatrice.

On remarque que tester la vacuité de κ est équivalent à *l'analyse d'alias* entre pointeurs [Deu94, Ste96], et la vacuité d'une relation rationnelle ou algébrique est décidable.

Pour établir le transducteur décrivant les dépendances, on doit d'abord restreindre la relation κ aux couples d'accès comportant au moins une écriture, puis on intersecte avec l'ordre lexicographique. En utilisant les techniques des sections III.3, III.4 et III.5, on peut toujours calculer une approximation conservatrice δ. Celle-ci est réalisée par un transducteur à un compteur dans le cas des tableaux, et par un transducteur rationnel dans le cas des arbres. De plus, grâce à la proposition 1, l'intersection avec l'ordre lexicographique n'est pas approximative dans le cas des tableaux.

Si l'on cherche à calculer les définitions visibles à partir de l'information approchée sur les dépendances, on aura beaucoup de mal à obtenir un résultat précis. Passée la première étape de restriction de δ aux seules dépendances de flot, on doit utiliser des propriétés additionnelles sur le flot des données. La technique principale que nous utilisons est fondée sur une propriété structurelle des programmes :

Définition 8 (ancêtre) On définit Σ_{UNCO} : un sous-ensemble de Σ_{CTRL} constitué de toutes les étiquettes de *blocs* qui ne sont *pas* des instructions conditionnelles ou des corps de boucles, et de tous les appels de procédure (non gardés), c.-à-d. les blocs dont l'exécution est *inconditionnelle*.

Soient r et s deux instructions dans Σ_{CTRL}, et soit u un préfixe *strict* d'un mot de contrôle $wr \in L_{\text{CTRL}}$ (une instance de r). Si $v \in \Sigma^*_{\text{UNCO}}$ est tel que $uvs \in L_{\text{CTRL}}$, alors uvs est appelé ancêtre de wr.

Cette définition se comprend aisément sur un arbre de contrôle comme celui de la figure 9.b page 35: le carré noir $FPIA\text{aaaa}Js$ est un ancêtre de $FPIA\text{aaaa}JQPIA\text{a}B\text{r}$, mais pas les carrés gris adjacents. Les ancêtres ont les deux propriétés suivantes :

1. l'exécution de wr implique celle de u qui est sur le chemin de la racine au nœud wr ;

2. l'exécution de u implique celle de uvs car $v \in \Sigma^*_{\text{UNCO}}$.

Ainsi, si une instance s'exécute, *tous ses ancêtres* le font également. Pour appliquer ce résultat à l'analyse de définitions visibles, on commence par identifier les instances dont l'exécution est garantie par la propriété des ancêtres, puis on applique des règles d'élimination de transitions sur le transducteur des dépendances de flot. On obtient un transducteur qui réalise une approximation σ des définitions visibles.

L'intégration de ces idées dans l'algorithme d'analyse de définitions visibles étant relativement technique, nous en resterons la dans ce résumé.

IV.4 Les résultats de l'analyse

Revenons tout d'abord sur le cas des structures d'arbres. La fonction d'accès pour le programme BST est un transducteur rationnel décrit par la figure 11.

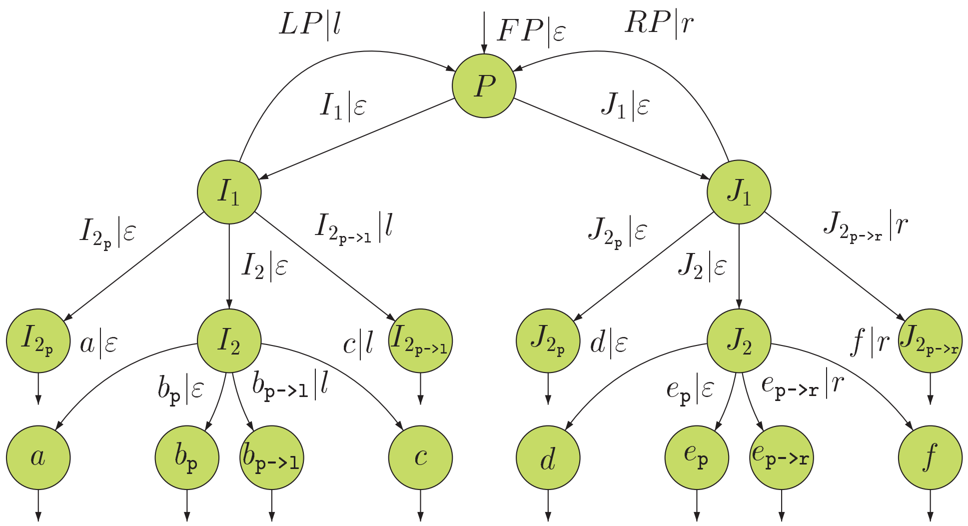


Figure 11. Transducteur rationnel pour la fonction d'accès f du programme BST

Le transducteur du conflit réalisant κ est toujours rationnel dans le cas des arbres. Lorsque le résultat est un transducteur synchrone à gauche, on peut calculer les dépendances sans approximation, sinon une approximation de κ à l'aide d'un transducteur synchrone à gauche est nécessaire. Le résultat pour BST est décrit par la figure 12.

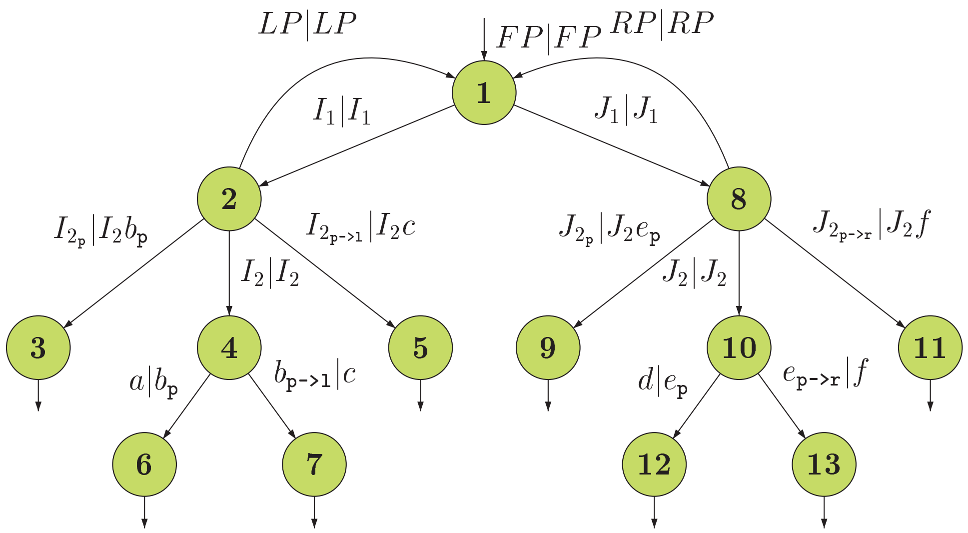


Figure 12. Transducteur rationnel pour la relation de dépendance δ du programme BST

On retrouve sur ce résultat le fait que les dépendances se situent entre les instances des instructions d'un même bloc I_1 ou J_1. Nous verrons que ce résultat permet de paralléliser le programme.

Étudions à présent le cas des tableaux. La fonction d'accès pour le programme `Queens` est décrite par un transducteur rationnel de Σ^*_{CTRL} dans $M_{\text{DATA}} = \mathbb{Z}$, donné sur la figure 13.

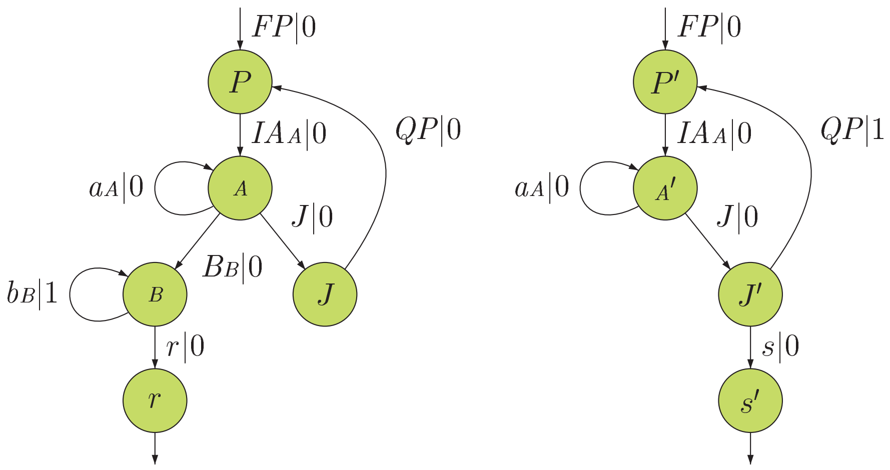


Figure 13. Transducteur rationnel pour la fonction d'accès f du programme `Queens`

On utilise le théorème 8 pour calculer un transducteur à un compteur réalisant la relation de conflit κ. Pour obtenir la relation de dépendance, on applique l'algorithme de resynchronisation au transducteur rationnel sous-jacent (qui est reconnaissable), le calcul est toujours exact. Le résultat pour `Queens` est donné par la figure 14.

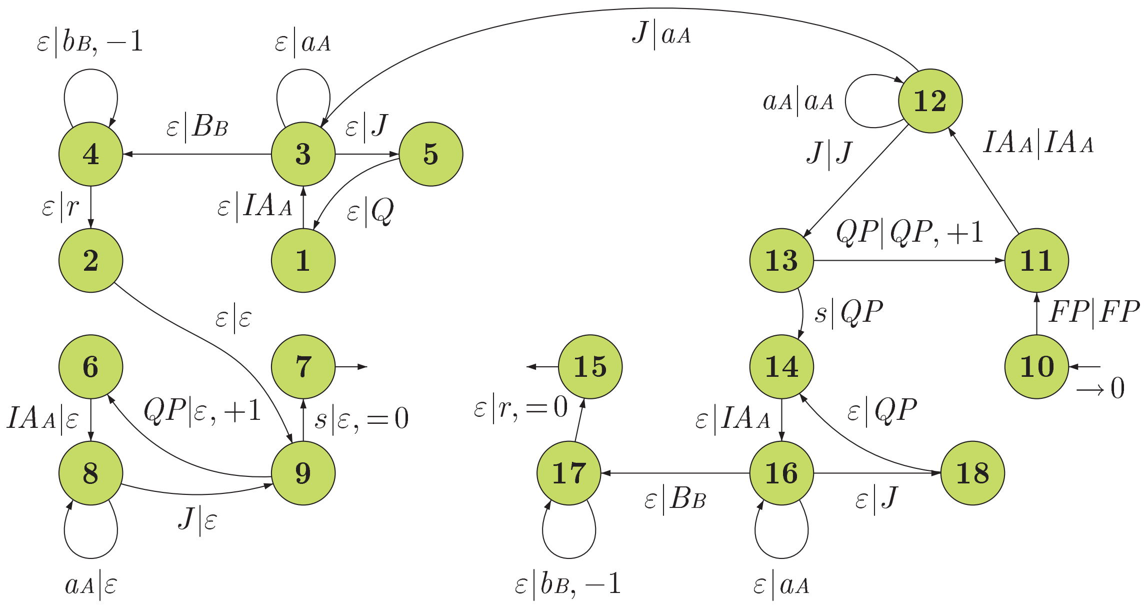


Figure 14. Transducteur à un compteur pour les dépendances de flot

On peut désormais effectuer l'analyse de définition visibles : en utilisant des informations supplémentaires sur les instructions conditionnelles du programme `Queens` on démontre que seuls des ancêtres d'une instance de r peuvent être des définitions visibles. Cette propriété très forte permet d'éliminer toutes les transitions qui ne mènent pas à un ancêtre dans le transducteur des dépendances. Le résultat est donné par la figure 15. On peut montrer facilement que le résultat est exact : une unique définition visible est calculée pour chaque accès en lecture.

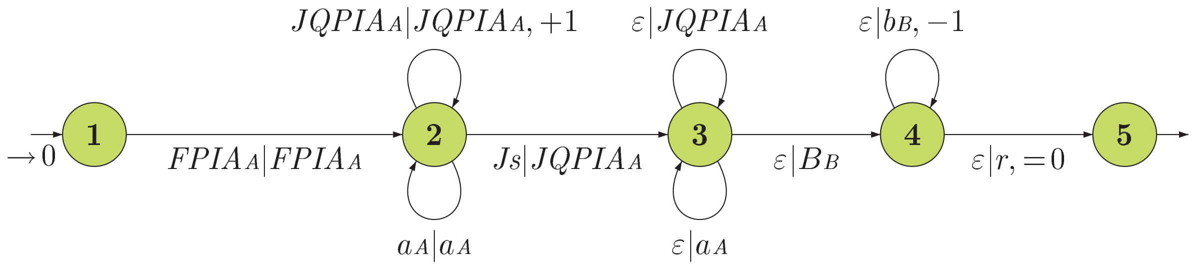


Figure 15. Transducteur à un compteur pour σ

IV.5 Comparaison avec d'autres analyses

Parmi les restrictions du modèle de programme, certaines peuvent être éliminées à l'aide de transformations préalables. De surcroît, de nombreuses restrictions semblent pouvoir être retirées dans des versions futures de l'analyse, à l'aide d'approximation adéquates. Il subsiste néanmoins une restriction très importante qui est fermement enracinée dans notre formalisme, et nous ne voyons pas de méthode générale pour s'en passer : les insertions et suppressions dans les arbres ne sont autorisées qu'au niveau des feuilles.

Les analyses statiques de dépendance et de définition visibles obtiennent généralement des résultats similaires, qu'elles soient fondées sur l'interprétation abstraite [Cou81, JM82, Har89, Deu94] ou d'autres formalismes d'analyse de flot de données [LRZ93, BE95, HHN94, KSV96]. Une étude intéressante des analyses statiques utiles en parallélisation est proposée dans [RR99]. Il est aisé de comparer notre technique avec ces analyses : *aucune* ne travail au niveau des *instances*. Aucune n'atteint la précision nécessaire pour identifier quelle instance de quelle instruction est en conflit, en dépendance, ou est une définition visible possible. Ces analyses sont cependant utiles pour lever un certain nombre de restrictions de notre modèle de programmes, et pour calculer des propriétés utiles à l'analyse de définitions visibles par instances. Il est plus intéressant de comparer ces analyses en matière d'applications à la parallélisation, voir section V.5.

Comparons à présent avec les analyses par instance pour nids de boucles, par exemple avec la FADA [BCF97, Bar98]. Sur l'intersection commune de leurs modèles de programmes, le résultat général n'est pas surprenant : les résultats de la FADA sont bien plus précis. En effet, nous n'utilisons les informations sur les instructions conditionnelles qu'à travers des analyses externes, des approximations supplémentaires sont nécessaires dans le cas de tableaux à plusieurs dimensions, les transducteurs rationnels et algébriques n'ont pas un pouvoir d'expression assez élevé pour manipuler des paramètres entiers (un seul compteur peut être décrit), et des opérations fondamentales comme l'intersection nécessitent parfois des approximations. On peut tout de même noter des points positifs : l'exactitude du résultat peut être décidée en temps polynômial sur les transducteurs rationnels ; la vacuité est toujours décidable, ce qui permet une détection automatique des variables non initialisées ; dans le cas des arbres, les tests de dépendance s'effectuent sur des langages rationnels de mots de contrôle, ce qui est très utile pour la parallélisation ;

enfin, dans le cas des tableaux, les tests de dépendance sont équivalents à l'intersection d'un langage rationnel avec un langage algébrique.

V Expansion et parallélisation

Les recherches sur l'expansion de la mémoire portent principalement sur les nids de boucles affines. Les techniques les plus courantes sont la mise en assignation unique [Fea91, GC95, Col98], la privatisation [MAL93, TP93, Cre96, Li92] et de nombreuses optimisations pour la gestion efficace de la mémoire [LF98, CFH95, CDRV97, QR99]. Lorsque le flot de contrôle n'est pas prévisible à la compilation ou lorsque les index de tableaux ne sont pas affines, le problème de la restauration du flot des données devient capital, et les convergences d'intérêt avec le formalisme SSA (*static single-assignment*) [CFR+91] sont très nettes. En partant d'exemples simples, nous étudions les problèmes spécifiques aux nids de boucles non affines, et proposons des algorithmes de mise en assignation unique. De nouvelles techniques d'expansion et d'optimisation de l'occupation en mémoire sont ensuite proposées pour la parallélisation automatique de codes irréguliers.

Les principes du calcul parallèle en présence de procédures récursives sont très différents de ceux des nids de boucles, et les méthodes de parallélisation existantes se fondent généralement sur des tests de dépendance au niveau des instructions, alors que notre analyse décrit la relation de dépendance au niveau des instances ! Nous montrons que cette information très précise permet d'améliorer notablement les techniques classiques de parallélisation. Nous étudions aussi la possibilité d'expanser la mémoire dans les programmes récursifs, et cette étude se termine par des résultats expérimentaux.

V.1 Motivations et compromis

La mise en assignation unique ou *single-assignment form conversion* (SA) est l'une des méthodes d'expansion les plus classiques. elle correspond au cas extrême où chaque cellule mémoire est écrite au plus une fois au cours de l'exécution. Elle diffère donc de la mise en assignation unique statique (SSA) [CFR+91, KS98], où l'expansion se limite à des renommages de variables.

L'idée consiste à remplacer chaque assignation d'une structure de données $\mathtt{D}$ par une assignation à une nouvelle structure $\mathtt{D}_{\text{EXP}}$ dont les éléments sont du même type que ceux de $\mathtt{D}$, et sont en bijection avec l'ensemble $\mathbf{W}$ de tous les accès en écriture possibles au cours de l'exécution. Dans une deuxième étape, les références en lecture doivent être mises à jour en conséquence : c'est ce que l'on appelle *la restauration du flot des données*. On utilise pour cela les définitions visibles par instances : pour une exécution donnée $e \in \mathbf{E}$, la référence à $\mathtt{D}$ en lecture $\langle \imath, \mathtt{ref} \rangle$ doit être remplacée par un accès à l'élément de $\mathtt{D}_{\text{EXP}}$ associé à $\sigma_e(\langle \imath, \mathtt{ref} \rangle)$. Puisque l'on ne dispose que d'une approximation σ des définitions visibles, cette technique n'est applicable que lorsque $\sigma(\langle \imath, \mathtt{ref} \rangle)$ est un singleton. Si ce n'est pas le cas, on doit générer un code de restauration *dynamique* du flot des données. Ce code est généralement représenté par une fonction ϕ, dont l'argument est l'ensemble $\sigma(\langle \imath, \mathtt{ref} \rangle)$ des définitions visibles possibles.

Pour générer le code de restauration dynamique associé aux fonctions ϕ, on utilise une structure de données supplémentaire en bijection avec $\mathtt{D}_{\text{EXP}}$: cette structure est notée $\mathtt{\Phi D}_{\text{EXP}}$. On doit mémoriser deux informations dans $\mathtt{\Phi D}_{\text{EXP}}$: l'adresse de la cellule mémoire écrite dans le programme d'origine et l'identité de la dernière instance qui a écrit une valeur dans cette cellule. Comme le programme est en assignation unique, l'instance est déjà décrite par l'élément de $\mathtt{D}_{\mathrm{EXP}}$ lui même : $\Phi\mathtt{D}_{\mathrm{EXP}}$ doit donc contenir des *adresses de cellules mémoire*. L'utilisation de cette structure est la suivante : on initialise $\Phi\mathtt{D}_{\mathrm{EXP}}$ à `NULL` ; puis à chaque assignation de $\mathtt{D}_{\mathrm{EXP}}$ on écrit dans $\Phi\mathtt{D}_{\mathrm{EXP}}$ l'adresse de la cellule mémoire écrite dans le programme d'origine ; enfin une référence $\phi(set)$ est implémentée par un calcul de maximum — selon l'ordre séquentiel — de tous les $\imath \in set$ tels que $\Phi\mathtt{D}_{\mathtt{EXP}}[\imath]$ soit égal à l'adresse de la cellule mémoire lue dans le programme d'origine.

L'analyse de définitions visibles par instances est à la base de la restauration du flot des données [Col98] : des résultats précis permettent non seulement de réduire le nombre de fonctions ϕ, mais également de simplifier les arguments de celles-ci, et donc d'optimiser les calculs de maximum au cours de l'exécution. On remarquera également que le calcul de σ à l'exécution peut lui même se révéler coûteux, même en l'absence de fonction ϕ. Dans le cas des nids de boucles, le surcoût n'est pourtant dû qu'à l'implémentation du quast associé à σ ; des techniques de parcours de polyèdre [AI91] permettent d'optimiser le code généré. L'exemple de la figure 16 illustre ces remarques. Dans le cas des programmes récursifs, nous verrons que le problème du calcul de σ est plus délicat.

```
  double A[N];
T A[0] = 0;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++) {
S     A[i+j] = ···;
R     A[i] = A[i+j-1] ···;
    }
```

Figure 16.a. Programme d'origine

```
  double A[N], A_T, A_S[N, N], A_R[N, N];
T A_T = 0;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++) {
S     A_S[i, j] = ···;
R     A_R[i, j] = φ({⟨T⟩} ∪ {⟨S, i', j'⟩ :
                   (i', j') <_LEX (i, j)}) ···
    }
```

Figure 16.b. SA sans analyse de définitions visibles

```
  double A[N], A_T;
  double A_S[N, N], A_R[N, N];
T A_T = 0;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++) {
S     A_S[i, j] = ···
R     A_R[i, j] = if (j==0)
                    if (i==0) A_T
                    else A_S[i-1, j]
                  else A_S[i, j-1]
                  ···;
    }
```

Figure 16.c. SA avec une analyse précise des définitions visibles

```
double A[N], A_T;
double A_S[N, N], A_R[N, N];
A_T = 0;
A_S[1, 1] = ···;
A_R[1, 1] = A_T ···;
for (i=0; i<N; i++) {
  A_S[i, 1] = ···;
  A_R[i, 1] = A_S[i-1, 1] ···;
  for (j=0; j<N; j++) {
    A_S[i, j] = ···;
    A_R[i, j] = A_S[i, j-1] ···;
  }
}
```

Figure 16.d. Analyse précise et « épluchage » de la boucle

Figure 16. Interactions entre l'analyse de définitions visibles et le surcoût à l'exécution

L'implémentation réelle de ces techniques dépend des structures de contrôle et de données. Dans le cas des boucles et des tableaux, nous proposons des algorithmes de mise en assignation unique qui étendent les résultats existants à des nids quelconques. La mise en assignation unique de programmes récursifs est un domaine nouveau que nous étudierons dans la section V.5.

Nous avons également développé trois techniques pour optimiser le calcul des fonctions ϕ. La première applique des optimisations simples sur les structures $\Phi\mathtt{D}_{\mathrm{EXP}}$; la deuxième réduit les ensembles de définitions visibles possibles (les arguments des fonctions ϕ) à l'aide d'une nouvelle information sur le flot des données appelée *définitions visible d'une cellule mémoire* ; et la troisième élimine les redondances dans le calcul du maximum en effectuant les calculs au fur et à mesure. Cette dernière technique ne génère pas à proprement parler un programme en assignation unique, ce qui peut parfois nuire à son utilisation en parallélisation automatique. Avec une vision différente de l'expansion (pas nécessairement en assignation unique), la section V.4 propose une version améliorée de la méthode d'élimination des redondances (appelée aussi « placement optimisé des fonctions ϕ ») qui ne nuit pas à la parallélisation.

V.2 Expansion statique maximale

Le but de l'*expansion statique maximale* est d'expanser la mémoire le plus possible — et donc d'éliminer le maximum de dépendances — sans recourir à des fonctions ϕ pour restaurer le flot des données.

Considérons deux écritures v et w appartenant à l'ensemble des définitions visibles possibles d'une lecture u, et supposons qu'elles affectent la même cellule mémoire. Si v et w écrivent dans deux cellules mémoire différentes après expansion, une fonction ϕ sera nécessaire pour choisir laquelle des deux écritures définit la valeur lue par u. On introduit donc la relation $\mathfrak{R}$ entre les écritures qui sont des définitions visibles possibles pour la même lecture :

$$\forall v, w \in \mathbf{W} : \quad v\,\mathfrak{R}\,w \iff \exists u \in \mathbf{R} : v\,\sigma\,u \wedge w\,\sigma\,u.$$

Lorsque deux définitions visibles possibles pour la même lecture écrivent la même cellule mémoire dans le programme d'origine, elles doivent faire de même dans le programme expansé. Puisque « écrire dans la même cellule mémoire » est une relation d'équivalence, on considère en fait la *clôture transitive* $\mathfrak{R}^*$ de la relation $\mathfrak{R}$. En se limitant à des fonctions d'accès expansées f_e^{EXP} de la forme (f_e, ν), où ν est une certaine fonction sur les accès en écriture, on montre le résultat suivant :

Proposition 3 Une fonction d'accès $f_e^{\mathrm{EXP}} = (f_e, \nu)$ est une expansion statique maximale pour toute exécution e ssi

$$\forall v, w \in \mathbf{W}_e, f_e(v) = f_e(w) : \quad v\,\mathfrak{R}^*\,w \iff \nu(v) = \nu(w).$$

À partir de ce résultat, on peut calculer une fonction ν en énumérant les classes d'équivalence d'une certaine relation. Le formalisme est donc très général, mais l'algorithme que nous proposons est limité aux nids de boucles quelconques sur tableaux. Un certain nombre de points techniques — notamment la clôture transitive de relations affines — requièrent une attention particulière, mais ceux-ci ne sont pas traités dans ce résumé en français.

Dans le cas général, la mise en assignation unique expose plus de parallélisme que l'expansion statique, il s'agit donc d'un compromis entre surcoût à l'exécution et parallélisme extrait. Nous présentons également trois exemples, sur lesquels nous appliquons semi-automatiquement (avec Omega [Pug92]) l'algorithme d'expansion. Toutefois, un seul exemple est étudié dans ce résumé, voir section V.4.

V.3 Optimisation de l'occupation en mémoire

Nous présentons maintenant une technique pour réduire l'occupation en mémoire d'un programme expansé sans perte de parallélisme. Nous supposons ainsi qu'un ordre d'exécution parallèle $<_{\text{PAR}}$ a déjà été déterminé pour le programme d'origine $(<_{\text{SEQ}}, f_e)$ — probablement à partir de la relation approchée des définitions visibles σ. Il est intéressant de noter que cet ordre parallèle peut être obtenu par n'importe quelle technique — ordonnancement ou partitionnement par exemple — tant que le résultat peut être décrit par une relation affine.

Moyennant un calcul de clôture transitive, il est même possible de partir de l'ordre « data-flow », c'est à dire l'ordre « le plus parallèle possible » d'après la relation de définitions visibles. On obtient alors un programme expansé qui requiert (généralement) moins de mémoire que la forme en assignation unique, mais qui est compatible avec n'importe quelle exécution parallèle légale.

Notre première tâche pour formaliser le problème consiste à déterminer quelles sont les expansions correctes vis à vis de cet ordre parallèle, c.-à-d. quelles sont les fonctions d'accès expansées f_e^{EXP} qui garantissent que l'ordre d'exécution parallèle préserve la sémantique du programme d'origine. En utilisant la notation

$$
\begin{aligned}
\forall v, w \in \mathbf{W}: \quad & v \bowtie w \overset{\text{def}}{\Longleftrightarrow} \\
& \big(\exists u \in \mathbf{R} : v\,\sigma\,u \wedge w \not<_{\text{PAR}} v \wedge u \not<_{\text{PAR}} w \wedge (u <_{\text{SEQ}} w \vee w <_{\text{SEQ}} v \vee v \not\bowtie w)\big) \\
& \vee \quad \big(\exists u \in \mathbf{R} : w\,\sigma\,u \wedge v \not<_{\text{PAR}} w \wedge u \not<_{\text{PAR}} v \wedge (u <_{\text{SEQ}} v \vee v <_{\text{SEQ}} w \vee w \not\bowtie v)\big),
\end{aligned}
$$

nous avons montré le résultat suivant :

Théorème 10 (correction des fonctions d'accès) Si la condition suivante est remplie, l'expansion est correcte, c'est à dire qu'elle garantit que l'ordre d'exécution parallèle préserve la sémantique du programme d'origine.

$$\forall e \in \mathbf{E}, \forall v, w \in \mathbf{W}_e : \quad v \bowtie w \implies f_e^{\text{EXP}}(v) \neq f_e^{\text{EXP}}(w).$$

Intuitivement, une définition visible v d'une lecture u et une autre écriture w doivent écrire dans des cellules mémoires distinctes lorsque : w s'exécute entre v et u dans le programme *parallèle*, et soit w ne s'exécute *pas* entre v et u soit w assigne une autre cellule mémoire que v dans le programme *d'origine*. De plus, nous avons montré que ce critère de correction est optimal, pour une approximation donnée des définitions visibles et de la fonction d'accès du programme d'origine.

À l'aide de ce critère, la génération du code expansé requiert la *coloration d'un graphe non borné* décrit par une relation affine. La méthode est la même que dans le cas des nids de boucles affines, elle est détaillée en français dans la thèse de Lefebvre [Lef98].

V.4 Expansion optimisée sous contrainte

Nous montrons à présent qu'il est possible de combiner les deux techniques d'expansion précédentes, et nous proposons un cadre général pour optimiser simultanément le surcoût de l'expansion et le parallélisme extrait : l'*expansion contrainte optimisée*. Le formalisme et les algorithmes sont trop techniques pour faire partie de ce résumé, nous nous contenterons donc de donner un exemple illustrant l'expansion contrainte — qui généralise l'expansion statique — combinée avec l'optimisation de l'occupation en mémoire.

```
    double x;
    for (i=1; i<=M; i++) {
      for (j=1; j<=M; j++)
        if (P(i,j)) {
T         x = 0;
          for (k=1; k<=N; k++)
S           x = x ···;
        }
R     ··· = x ···;
    }
```

Figure 17.a. Programme d'origine

```
    double x_T[M+1, M+1], x_S[M+1, M+1, N+1];
    parallel for (i=1; i<=M; i++) {
      parallel for (j=1; j<=M; j++)
        if (P(i,j)) {
T         x_T[i, j] = 0;
          for (k=1; k<=N; k++)
S           x_S[i, j, k] = if (k==1) x_T[i, j];
              else x_S[i, j, k-1] ···;
        }
R     ··· = φ({⟨S, i, 1, N⟩, ..., ⟨S, i, M, N⟩}) ···;
    }
```

Figure 17.b. Mise en assignation unique

Figure 17. Exemple de parallélisation

Nous étudions le pseudo-code de la figure 17.a. Nous supposons que N est strictement positif et que le prédicat $P(i, j)$ est vrai au moins une fois pour chaque itération de la boucle externe. Les dépendances sur `x` interdisent toute exécution parallèle, on transforme donc le programme en assignation unique. Le résultat de l'analyse de définitions visibles est exact pour les instances de S, mais pas pour celles de R : une fonction ϕ est nécessaire. Les deux boucles externes deviennent alors parallèles, comme le montre la figure 17.b.

En raison de cette fonction ϕ et de l'utilisation d'un tableau a trois dimensions, on observe que l'exécution en parallèle de ce programme est environ cinq fois plus lente que l'exécution séquentielle (sur SGI Origin 2000 avec 32 processeurs). Il est donc nécessaire de réduire l'occupation en mémoire. L'application de l'algorithme de la section V.3 montre que l'expansion selon la boucle la plus interne n'est pas nécessaire, pas plus que le renommage de `x` en $\mathtt{x}_S$ et $\mathtt{x}_T$. On obtient le code de la figure 18.a. On a implémenté la fonction ϕ avec une technique optimisée de calcul à la volée (voir section V.1) et le calcul du `max` cache une synchronisation. Les performances sont donc correctes pour un petit nombre de processeurs, mais se dégradent très rapidement au delà de quatre.

L'application de l'algorithme d'expansion statique maximale permet de se débarrasser de la fonction ϕ, en interdisant l'expansion selon la boucle intermédiaire, voir figure 18.b ; seule la boucle externe reste parallèle. Le programme parallèle sur un processeur est environ deux fois plus lent que le programme séquentiel (probablement en raison des accès au tableau à deux dimensions), mais l'accélération est excellente. On observe que la variable `x` a été à nouveau expansée selon la boucle interne, bien que cela n'apporte aucun parallélisme supplémentaire : il est donc nécessaire de combiner les deux techniques d'expansion. Le résultat est très proche de l'expansion statique maximale avec une dimension de moins pour le tableau `x` : `x[i]` au lieu de `x[i, ···]`. Bien entendu, les performances sont excellentes : l'accélération est de 31, 5 sur 32 processeurs ($M = 64$ et $N = 2048$).

V.5 Parallélisation de programmes récursifs

Des techniques de parallélisation automatique pour programmes récursifs commencent à voir le jour, grâce aux environnements et aux outils — comme Cilk [MF98] — facilitant l'implémentation efficace de programmes à parallélisme de contrôle [RR99]. Nous proposons une technique de mise en assignation unique et une technique de privatisation pour

```
 double x[M+1, M+1];
 int @x[M+1];
 parallel for (i=1; i<=M; i++) {
   @x[i] = ⊥;
   parallel for (j=1; j<=M; j++)
     if (P(i,j)) {
T      x[i, j] = 0;
       for (k=1; k<=N; k++)
S        x[i, j] = x[i, j] ···;
       @x[i] = max (@x[i], j);
     }
R  ··· = x[i, @x[i]] ···;
 }
```

Figure 18.a. Optimisation de l'occupation en mémoire

```
   double x[M+1, N+1];
   parallel for (i=1; i<=M; i++) {
     for (j=1; j<=M; j++)
       if (P(i,j)) {
T        x[i, 0] = 0;
         for (k=1; k<=N; k++)
S          x[i, k] = x[i, k-1] ···;
       }
R    ··· = x[i, N] ···;
   }
```

Figure 18.b. Expansion statique maximale

Figure 18. Deux parallélisations différentes

programme récursifs, puis nous présentons deux méthodes de génération de code parallèle.

Expansion de programmes récursifs

Dans un programme récursif en assignation unique, les structures expansées ont généralement une structure d'*arbre* : ses élément sont en bijection avec les mots de contrôle. L'allocation dynamique et l'accès à ces structures est donc plus délicat que dans le cas des nids de boucles. L'idée générale est de construire chaque structure expansée $\mathtt{D}_{\text{EXP}}$ « à la volée », en propageant un pointeur sur le nœud courant. L'accès direct à $\mathtt{D}_{\text{EXP}}$ est toutefois nécessaire pour la mise à jour des références en lecture : on doit tout d'abord calculer les définitions visibles possibles à l'aide du transducteur fourni par l'analyse, puis retrouver les cellules mémoire associées dans $\mathtt{D}_{\text{EXP}}$. Même en l'absence de fonction ϕ, la restauration du flot des données risque donc d'être très coûteuse.

Si les définitions visibles sont connues exactement, σ peut être vue comme une fonction partielle de $\mathbf{R}$ dans $\mathbf{W}$. Lorsque cette fonction peut être calculée « à la volée », il est possible de générer un code efficace pour les références en lecture du programme expansé : il suffit d'implémenter le calcul pas à pas du transducteur. C'est notamment le cas pour les *transducteurs sous-séquentiels* (voir section III.2), lorsque le programme récursif manipule une structure d'arbre. En présence de tableaux, il est plus difficile de savoir si le transducteur *à un compteur* des définitions visibles est calculable « à la volée ». Nous avons toutefois proposé un algorithme de mise en assignation unique pour programmes récursif, incluant le calcul à la volée des définitions visibles lorsque cela est possible.

Nous avons étendu la notion de *privatisation* aux programmes récursifs : elle consiste à transformer les structures de données globales en variables locales. Dans le cas général, une copie des données doit être effectuée lors de chaque appel et de chaque retour d'une procédure. Ceci peut se révéler coûteux lors de la copie des structures locales dans les structures de la procédure appelante (le *copy-out*), notamment à cause des synchronisations inévitables en cas d'exécution parallèle. Toutefois, lorsque les définitions visibles sont obligatoirement des ancêtres, seule la première phase de copie (le *copy-in*) est nécessaire ;

c'est le cas du programme `Queens`, de la plupart des algorithmes de tri, et plus généralement des schémas d'exécution du type *diviser pour régner* ou *programmation dynamique*. Nous proposons donc un algorithme de privatisation pour programme récursifs, où les fonctions ϕ sont remplacées par des copies de structures de données.

Génération de code parallèle

```
    int A[n];
P   void Queens (int A[n], int n, int k) {
      int B[n];
      memcpy (B, A, k * sizeof (int));
I     if (k <n) {
A/a     for (int i=0; i<n; i++) {
B/b       for (int j=0; j<k; j++) {
r           ··· = ··· B[j] ···;
          }
J         if (···) {
s           B[k] = ···;
Q           spawn Queens (B, n, k+1);
          }
        }
      }
    }
    int main () {
F     Queens (A, n, 0);
    }
```

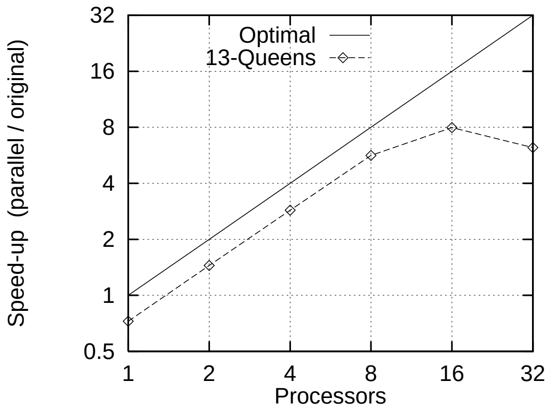


Figure 19. Privatisation et parallélisation du programme `Queens`

Nous montrons que les propriétés de décidabilité des transducteurs rationnels et algébriques permettent de réaliser des tests de dépendance efficaces. On en déduit un algorithme de parallélisation *au niveau des instructions* qui permet d'exécuter certaines instructions de manière asynchrone et qui introduit des synchronisations lorsque les dépendances l'exigent. Cet algorithme est appliqué au programme `BST`, ainsi qu'au programme `Queens`*après privatisation*, voir figure 19. L'expérimentation a été faite sur une SGI Origin 2000 pour $n = 13$. Le ralentissement sur un processeur est dû aux copies de tableaux, et dans une moindre mesure à l'ordonnanceur de Cilk [MF98].

Nous montrons également que notre algorithme de parallélisation donne de meilleurs résultats que les techniques existantes, lorsque la découverte de parallélisme nécessite une information *au niveau des instances*. Enfin, nous étudions la parallélisation *par instances* de programmes récursifs, où les synchronisations sont gardées par les conditions précises — sur le mot de contrôle — pour lesquelles une dépendance est possible. L'algorithme que nous proposons exploite pleinement le résultat de l'analyse de dépendances par instances, et la possibilité de tester efficacement si un couple de mots est reconnu par un transducteur. Un exemple concret permet de valider cette nouvelle technique.

VI Conclusion

Cette thèse se conclut par une récapitulation des principaux résultats, suivie d'une discussion sur les développements à venir.

VI.1 Contributions

Nos contributions se répartissent en quatre catégories fortement interdépendantes. Les trois premières concernent la parallélisation automatique et sont résumées dans le tableau suivant ; la quatrième catégorie concerne les transductions rationnelles et algébriques.

	Nids affines sur tableaux	Nids généraux sur tableaux	Programmes récursifs sur arbres et tableaux
Analyse de dépendances par instances	[Bra88, Ban88] [Fea88a, Fea91, Pug92]	[BCF97, Bar98] [WP95, Won95]	[Fea98] [1], section **IV**, publié dans [CC98] [2]
Analyse de définitions visibles par instances	[Fea88a, Fea91, Pug92] [MAL93]	[CBF95, BCF97, Bar98] [WP95, Won95]	section **IV**, publié dans [CC98] [2]
Mise en assignation unique	[Fea88a, Fea91]	[Col98], sections **V.1** et **V.4**	section **V.5**
Expansion statique maximale	sections **V.2** et **V.4**, publié dans [BCC98, Coh99b, BCC00]		*problème ouvert*
Optimisation de l'occupation mémoire	[LF98, Lef98] [SCFS98, CDRV97]	sections **V.3** et **V.4**, publié dans [CL99, Coh99b]	*problème ouvert*
Parallélisation par instances	[Fea92, CFH95] [DV97]	[GC95, CBF95] [Col95b]	section **V.5**

À présent, passons en revue chaque contribution.

Structures de contrôle et de données : au delà du modèle polyédrique Dans la section II, nous avons défini un modèle de programmes et des abstractions mathématiques pour les instances d'instructions et les éléments de structures de données. Ce cadre général a été utilisé tout au long de ce travail pour formaliser la présentation de nos techniques, en particulier dans le cas des structures récursives.

De nouvelles analyses de dépendances et de de définitions visibles ont été proposées dans la section IV. Elles utilisent un formalisme de la théorie des langages formels, plus précisément des transductions rationnelles et algébriques. Une nouvelle définition des variables d'induction adaptée aux programmes récursifs a permis de décrire l'effet de chaque instance à l'aide d'une transduction rationnelle ou algébrique. Une comparaison avec d'autres analyses conclut ce travail.

En revanche, lorsque nous avons conçu des algorithmes pour les nids de boucles sur tableaux — un cas particulier de notre modèle — nous sommes restés fidèles aux vecteurs

1. Il s'agit d'un *test* de dépendances pour les *arbres* uniquement.
2. Pour les *tableaux* uniquement.

d'itération et nous avons profité de la quantité d'algorithmes permettant la manipulation de relations affines dans l'arithmétique de Presburger.

Expansion de la mémoire : de nouvelles techniques pour résoudre de nouveaux problèmes L'application de l'expansion de la mémoire à la parallélisation est une technique ancienne, mais les analyses de définitions visibles par instances se sont récemment étendues aux programmes avec des expressions conditionnelles, avec des références complexes aux structures de données — par exemple des index de tableaux non affines — ou avec des appels récursifs, et cela pose de nouvelles questions. La première est de garantir que les accès en lecture dans le programme expansé réfèrent la bonne cellule mémoire ; la deuxième question réside dans l'adéquation des techniques d'expansion avec les nouveaux modèles de programmes.

Les deux questions sont traitées dans les sections V.1, V.2, V.3 et V.4, dans pour les nids de boucles (sans restrictions) sur tableaux. Nous avons présenté une nouvelle technique pour réduire le surcoût de l'expansion à l'exécution, et nous avons étendu aux nids de boucles sans restrictions une méthode de réduction de l'occupation en mémoire. La combinaison des deux a été étudiée et nous avons conçu des algorithmes pour optimiser la restauration du flot des données à l'exécution. Quelques résultats expérimentaux sont présentés pour une architecture à mémoire partagée.

L'expansion de la mémoire pour programmes récursifs est un domaine de recherche totalement nouveau, et nous avons découvert que l'abstraction mathématique pour les définitions visibles — les transductions rationnelles ou algébriques — peuvent engendrer des surcoûts importants. Nous avons néanmoins développé des algorithmes qui expansent des programmes récursifs particuliers avec un faible surcoût à l'exécution.

Parallélisme : extension des techniques classiques Notre analyse de dépendance a été mise à profit pour paralléliser des programmes récursifs. Nous avons pu démontrer les applications pratiques des transductions rationnelles et algébriques, en utilisant leurs propriétés décidables. Notre premier algorithme ressemble aux méthodes existantes, mais il profite de l'information plus précise recueillie par l'analyse et on obtient en général de meilleurs résultats. Un autre algorithme permet la parallélisation par instances de programmes récursifs : cette nouvelle technique est rendue possible par l'utilisation des transductions rationnelles et algébriques. Quelques résultats expérimentaux sont décrits, en combinant expansion et parallélisation sur un programme récursif bien connu.

Théorie des langages formels : quelques contributions et des applications Les derniers résultats de ce travail n'appartiennent pas au domaine de la compilation. Ils se trouvent principalement dans la section III.3 ainsi que dans les sections suivantes. Nous avons défini une sous-classe des transductions rationnelles qui admet une structure d'algèbre booléene et de nombreuses autres propriétés intéressantes. Nous avons montré que cette classe n'est pas décidable parmi les transductions rationnelles, mais des techniques d'approximation conservatrices permettent de bénéficier de ces propriétés dans la classe des transductions rationnelles tout entière. Nous avons également présenté quelques nouveaux résultats sur la composition de transductions rationnelles sur des monoïdes non libres, avant d'étudier l'approximation de transductions algébriques.

VI.2 Perspectives

De nombreuses questions se sont posées tout au long de cette thèse, et nos résultats suggèrent plus de recherches intéressantes qu'ils ne résolvent de problèmes. Nous commençons par aborder les questions liées aux programmes récursifs, puis nous discutons des travaux futurs dans le modèle polyédrique.

En premier lieu, la recherche d'une abstraction mathématique capable de décrire des propriétés au niveau des instances apparaît de nouveau comme un enjeu capital. Les transductions rationnelles et algébriques ont souvent donné de bons résultats, mais leur expressivité limitée a également restreint leur champ d'application. C'est l'analyse de définitions visibles qui en a le plus souffert, ainsi que l'intégration des expressions conditionnelles et des bornes de boucles dans l'analyse de dépendances. Dans ces conditions, nous aurions besoin de plus d'un compteur dans les transducteurs, tout en conservant la possibilité de savoir si un ensemble est vide et de décider d'autres propriétés intéressantes. Nous sommes donc fortement intéressés par les travaux de Comon et Jurski [CJ98] sur la décision de la vacuité dans une sous-classe des langages à plusieurs compteurs, et plus généralement nous voudrions suivre de plus près les études sur la vérification de systèmes fondées sur des classes restreintes de machines de Minsky, comme les automates temporisés. L'utilisation de plusieurs compteurs permettrait en plus d'étendre l'une des grandes idées de l'analyse floue de flot des données [CBF95] : l'insertion de nouveaux paramètres pour améliorer la précision en décrivant les propriétés des expressions non affines.

De plus, nous pensons que les propriétés de décidabilité ne sont pas forcément le point le plus important pour le choix d'une abstraction mathématique : de bonnes approximations sur les résultats sont souvent suffisantes. En particulier, nous avons découvert en étudiant les relations synchrones à gauche et les relations déterministes qu'une sous-classe avec de bonnes propriétés de décision ne peut pas être utilisée dans notre cadre général d'analyse sans méthode efficace d'approximation. L'amélioration de nos méthodes de resynchronisation et d'approximation de transducteurs rationnels est donc un enjeu important. Nous espérons aussi que ceci démontre l'intérêt mutuel des coopérations entre théoriciens et chercheurs en compilation.

Au delà de ces problèmes de formalisme, une autre voie de recherche consiste à diminuer autant que possible les restrictions imposées au modèle de programme. Comme on l'a proposé précédemment, la meilleure méthode consiste à rechercher une dégradation progressive des résultats à l'aide de techniques d'approximation. Cette idée a été étudiée dans un contexte semblable [CBF95], et l'application aux programmes récursifs promet des travaux futurs intéressants. Une autre idée serait de calculer les variables d'induction à partir des traces d'exécution (au lieu des mots de contrôle) — pour autoriser les modifications dans n'importe quelle instruction — puis de déduire des informations approximatives sur les mots de contrôle ; l'utilisation de techniques d'interprétation abstraite [CC77] serait probablement une aide précieuse pour prouver la correction de nos approximations.

Nous n'avons pas travaillé sur le problème de l'ordonnancement des programmes récursifs, car nous ne connaissons aucune méthode permettant d'assigner des ensembles d'instances à des dates d'exécution. La construction d'un transducteur rationnel des dates aux instances est peut être une bonne idée, mais la génération de code pour énumérer les ensembles d'instances devient plutôt difficile. Mais ces raisons techniques ne doivent pas cacher que l'essentiel du parallélisme dans les programmes récursifs peut d'ores et déjà être exploité par des techniques à parallélisme de contrôle, et la nécessité de recourir à un modèle d'exécution à parallélisme de données n'est pas évidente.

En plus de leur incidence sur notre étude des programmes récursifs, les techniques issues du modèle polyédrique recouvrent une partie importante de cette thèse. Un objectif majeur tout au long de ces travaux a été de conserver une certaine distance avec la représentation mathématique des relations affines. Ce point de vue a l'inconvénient de ne pas faciliter l'écriture d'algorithmes optimisés prêts à l'emploi dans un compilateur, mais il a surtout l'avantage de présenter notre approche dans toute sa généralité. Parmi les problèmes techniques qui devraient être améliorés, tant pour l'expansion statique maximale et pour l'optimisation de l'occupation en mémoire, les plus importants sont les suivants.

Nous avons présenté de nombreux algorithmes pour la restauration dynamique du flot des données, mais nous avons très peu d'expérience pratique de la parallélisation de nids de boucles avec un flot de contrôle imprévisible et des index de tableaux non affines. Comme le formalisme SSA [CFR+91] est principalement utilisé en tant que représentation intermédiaire, les fonctions ϕ sont rarement implémentées en pratique. La génération d'un code de restauration efficace est donc un problème plutôt récent.

Aucun paralléliseur pour nids de boucles sans restrictions n'a jamais été écrit. Il en résulte qu'une expérimentation de grande ampleur n'a jamais pu être conduite. Pour appliquer des analyses et des transformations précises sur des programmes réels, un important travail d'optimisation reste à conduire. Les idées principales seraient de partitionner le code [Ber93] et d'étendre nos techniques aux graphes de dépendance hiérarchiques, aux régions de tableaux [Cre96] ou aux ordonnancements hiérarchiques [CW99].

Un compilateur parallélisant doit être capable de régler automatiquement un grand nombre de paramètres : le surcoût à l'exécution, l'extraction du parallélisme, l'occupation en mémoire, le placement des calculs et des communications... Nous avons vu que le problème d'optimisation est encore plus complexe pour des nids de boucles non affines. Le formalisme d'expansion contrainte permet d'optimiser simultanément un certain nombre de paramètres liés à l'expansion de la mémoire, mais il ne s'agit que d'un premier pas.

Chapter 1

Introduction

Performance increase in computer architecture technology is the combined result of several factors: fast increase of processor frequency, broader bus widths, increased number of functional units, increased number of processors, complex memory hierarchies to deal with high latencies, and global increase of storage capacities. New improvements and architectural designs are proposed every day. The result is that the machine model is becoming less and less uniform and simple: despite the hardware support for caches, superscalar execution and shared memory multiprocessing, tuning a given program for performance becomes more and more complex. Good optimizations for some particular case can lead to disastrous results with a different machine. Moreover, hardware support is generally not sufficient when the complexity of the system becomes too high: dealing with deep memory hierarchies, local memories, out of core computations, instruction level parallelism and coarse grain parallelism requires additional support from the compiler to translate raw computation power into sustained performance. The recent shift of microprocessor technology from superscalar models to explicit instruction level parallelism is one of the most concrete signs of this trend.

Indeed, the whole of computer architecture and compiler industry is now facing what the high performance computing community has discovered for years. On the one hand, and for most applications, architectures are too diverse to define practical efficiency criteria and to develop specific optimizations for a particular machine. On the second hand, programs are written in such a way that traditional optimization and parallelization techniques have many problems to feed the huge computation monster everybody will have tomorrow in his laptop.

In order to achieve high performances on modern microprocessors and 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. The following two possibilities are offered to the programmer to meet these requirements.

- First, explicitly parallel languages. Most of these are parallel extensions of sequential languages. This includes well known data parallel languages such as HPF, and recent mixed data and control parallel approaches such as OpenMP extensions for shared memory architectures. Some extensions also appear under the form of libraries: PVM and MPI for instance, or higher-level multi-threaded environments such as IML from the University of Illinois [SSP99] or Cilk from the MIT [MF98].

These approaches makes the programming of high performance parallel algorithms possible. However, besides parallel algorithmics, the programmer 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. This requires a deep knowledge of the target architecture and reduces portability. Several efforts have been done in HPF so as to make the compiler take care of some parts of this job, but it seems that the programmer still needs to have a precise knowledge of what the compiler does.

- Second, automatic parallelization of a high level sequential language. The obvious advantages of this approach are the portability, the simplicity of programming 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 generate a parallel code which takes into account the specificities of the target architecture. Even for short programs and a simplified model of parallel machine, "optimality" in both steps is out of reach for decidability reasons. As a matter of fact, a wide panel of parallelization techniques exists, and the difficulty often lies in choosing the more appropriate.

  The usual source languages for automatic parallelization is Fortran 77. Indeed, many scientific applications have been written with Fortran, which 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 Lisp. These studies are less advanced than the historical approach, but also more related with the present work: they handle programs with general control and data structures. Many research projects already exist, among others: Parafrase-2 and Polaris [BEF+96] from the University of Illinois, PIPS from École des Mines [IJT90], SUIF from Stanford University [H+96], the McCat/EARTH-C compiler from Mc Gill University [HTZ+97], LooPo from the University of Passau [GL97], and PAF from the University of Versailles; there are also an increasing number of commercial parallelizing tools, such as CFT, FORGE, FORESYS or KAP.

We are mostly interested in automatic and semi-automatic parallelization techniques: this thesis addresses both program analysis and source to source program transformation.

1.1 Program Analysis

Optimizations and parallelizations are usually seen as source to source code transformations which improves one or several run-time parameters. To apply a program transformation at compile-time, one must check that the algorithm implemented by the program is unharmed during the process. Because an algorithm can be implemented in many different ways, applying a program transformation requires "reverse engineering" the most precise information about what the program does. This fundamental program analysis technique addresses the difficult problem of gathering *compile-time*—a.k.a. *static*—information about *run-time*—a.k.a. *dynamic*—properties.

Static Analysis

Program analyses often compute properties of the machine state between execution of two instructions. These machine states are known as *program points*. Such properties are called *static* because they cover *every possible run-time execution* leading to a given program point. Of course these properties are computed at compile-time, but this is not the meaning of the "static" adjective: "syntactic" would probably be more appropriate...

Data-flow analysis is the first proposed framework to unify the large number of static analyses. Among the various wordings and formal presentations of this framework [KU77, Muc97, ASU86, JM82, KS92, SRH96], one may expose the following common issues. To formally state the possible run-time executions, the usual method is to build the *control flow graph* of the program [ASU86]; indeed, this graph represents all program points as nodes, and edges between these nodes are labeled with program statements. The set of all possible executions is then the set of all *paths* from the initial state to the considered program point. Properties at a given program point are defined as follows: because each statement may modify some property, one must consider every path leading to the program point and *meet* all informations along these paths. The formal statement of these ideas is usually called *meet over all paths* (MOP) [KS92]. Of course, the *meet* operation depends on the property to be evaluated and on its mathematical abstraction.

However, because of the possibly unbounded number of paths, the MOP specification of the problem cannot be used for practical evaluation of static properties. Practical computation is done by—forward or backward—propagation of the intermediate results along edges of the control flow graph. An *iterative* resolution of the *propagation equations* is performed, until a *fix-point* is reached. This method is known as *maximal fixed point* (MFP). In the intra-procedural case, Kam and Ullman [KU77] have proven that MFP effectively computes the result defined by MOP—i.e. MFP *coincides* with MOP—when some simple properties of the mathematical abstraction are satisfied; and this result has been extended to inter-procedural analysis by Knoop and Steffen [KS92].

Mathematical abstractions for program properties are very numerous, depending on the application and complexity of the analysis. The *lattice* structure encompasses most abstractions because it supports computation of both *meet*—at merge points—and *join*—at computational statements—operations. In this context, Cousot and Cousot [CC77] have proposed an *approximation* framework based on semi-dual Galois connections between *concrete* run-time states of a program and *abstract* compile-time properties. This mathematical formulation called *abstract interpretation* has two main interests: first it allows systematic approaches to the construction of a lattice abstraction for program properties, and second, it ensures that any computed fix-point in the abstract lattice corresponds to a *conservative approximation* of an actual fix-point in the lattice of concrete states. While extending the concept of data-flow analysis, abstract interpretation helps proving the correctness and optimality of program analyses. Practical applications of abstract interpretation and related iterative methods can be found in [Cou81, CH78, Deu92, Cre96].

Despite the undisputable successes of data-flow and abstract interpretation frameworks, the automatic parallelization community has very rarely based its analysis techniques on one of these frameworks. Beyond the important reasons which are not of a scientific nature, we will discuss the good reasons:

- MOP/MFP techniques focus on classical optimizations techniques, with rather simple abstractions (lattices often have a bounded height); correctness and efficiency in a production compiler are the main motivations, whereas precision and expressiveness of the mathematical abstraction are the main issues for parallelization;

- in the industry, parallelization has traditionally addressed nests of loops and arrays, with high degrees of data parallelism and simple (non recursive, first order) control structures; proving the correctness of an analysis is easy in this context, whereas applications to real programs and practical implementation in a compiler become issues of critical interest;
- abstract interpretation is well suited to functional languages with clean and simple operational semantics; problems raised in this context are orthogonal with practical issues of imperative and low-level languages such as Fortran or C, traditionally more suitable for parallel architectures (but we will see that this point is evolving).

As a result, data-flow and abstract interpretation frameworks have mostly focused on static analysis techniques, which compute properties at a given *program point* or *statement*. Such results are well suited to most classical techniques for program checking and optimization [Muc97, ASU86, SKR90, KRS94], but for automatic parallelization purposes, one needs more information.

- What about *distinct run-time instances* of program points and statements? Because statements are likely to execute several times, we are interested in *which iteration of a loop* or *which call to a procedure* induced execution of some program statement.
- What about *distinct elements* in a data structure? Because arrays and dynamically allocated structures are not atomic, we are interested in *which array element* or *which graph node* is accessed by some *run-time instance* of a statement.

Because of orthogonal interests in the data-flow analysis and the automatic parallelization communities, it is not surprising that results of the ones could not be applied by the others. Indeed, a very small number of data-flow analyses [DGS93, Tzo97] addressed both instancewise and elementwise issues, but results are very far from the requirements of a compiler in terms of precision and applicability.

Instancewise Analysis

Program analyses for automatic parallelization are a rather restricted domain, compared to the broad range of properties and techniques studied in data-flow analysis frameworks. The program model considered is also more restricted—most of the time—since traditional applications of parallelizing compilers are numerical codes with loop nests and arrays.

Since the very beginning—with works by Banerjee [Ban88], Brandes [Bra88] and Feautrier [Fea88a]—analyses are oriented towards *instancewise* and *elementwise* properties of programs. When the only control structure was the `for`/`do` loop, iterative methods with a high semantical background seemed overly complex. To focus on solving critical problems such as abstracting loop iterations and effects of statement instances on array elements, designing simple and ad-hoc frameworks was obviously more profitable than trying to build on unpractical data-flow frameworks. The first analyses were *dependence tests* [Ban88] and *dependence analyses* [Bra88, Pug92] which collected information about statement instances which access the same memory location, one of the accesses being a write. More precise methods have been designed to compute, for every array element read in an expression, the very statement instance which produced the value. They are usually called *array data-flow analyses* [Fea91, MAL93], but we prefer to call them *instancewise* *reaching definition analyses* for better comparison with a specific *static* data-flow analysis technique called *reaching definition analysis* [ASU86, Muc97]. Such accurate information significantly improves the quality of program transformation techniques, hence the performance of parallel programs.

Instancewise analyses have long suffered strong program model restrictions: programs used to be nested loops without conditional statements, with affine bounds and array subscripts, and without procedure calls. This very limited model is already sufficient to address many numerical codes, and has the major interest of allowing computation of *exact* dependence and reaching definition information [Fea88a, Fea91]. One of the difficulties in removing the restrictions is that exact results cannot be hoped for anymore, and only *approximate* dependences are available at compile-time: this induces overly conservative approximations of reaching definition information. A direct computation of reaching definitions is thus needed. Recently, such direct computations have been crafted, and extremely precise *intra-procedural* techniques have been designed by Barthou, Collard and Feautrier [CBF95, BCF97, Bar98] and by Pugh and Wonnacott [WP95, Won95]. In the following, *fuzzy array dataflow analysis* (FADA) by Barthou, Collard and Feautrier [Bar98] will be our prefered instancewise reaching definition analysis for *programs with unrestricted nested loops and arrays.*

Many extensions to handle procedure calls have been proposed [TFJ86, HBCM94, CI96], but they are not *fully instancewise* in the sense that they do not distinguish between multiple executions of a statement associated with *distinct calls* of the surrounding procedure. Indeed, the first fully instancewise analysis for programs with—possibly recursive—procedure calls is presented in this thesis.

The next section introduces program transformations useful to parallelization. Most of these transformations will be studied in more detail in the rest of this thesis. Of course, they are based on *instancewise* and *elementwise* analysis of program properties.

1.2 Program Transformations for Parallelization

Dependences are known to hamper parallelization of imperative programs and their efficient compilation on modern processors or supercomputers. A general method to reduce the number of memory-based dependences is to disambiguate memory accesses in assigning distinct memory locations to independent writes, i.e. to *expand* data structures.

There are many ways to compute *memory expansions*, i.e. to transform memory accesses in programs. Classical ways include renaming scalars, arrays and pointers, splitting or merging data structures of the same type, reshaping array dimensions, including adding new dimensions, converting arrays into trees, changing the degree of a tree, and changing a global variable into a local one.

Read references are also expanded, using instancewise reaching definition information to implement the expanded reference [Fea91]. Figure 1.1 shows three programs with no possible parallel execution because of output dependences (details of the code are omitted when not useful for presentation). Expanded versions are given in the right-hand side of the figure, to illustrate the benefit of memory expansion for parallelism extraction.

Unfortunately, when the control-flow cannot be predicted at compile-time, some run-time computation is needed to preserve the original data flow: ϕ functions may be needed to "merge" data definitions due to several incoming control paths. These ϕ functions are similar—but not identical—to those of the *static single-assignment* (SSA) framework by Cytron et al. [CFR+91], and have been first extended for instancewise expansion schemes

```
int x;                      int x1, x2;
x = ···; ··· = x;           x1 = ···; ··· = x1;
x = ···; ··· = x;           x2 = ···; ··· = x2;
```

After expansion, i.e. renaming `x` in `x1` and `x2`, the first two statements can be executed in parallel with the two others.

```
   int A[10];                      int A1[10], A2[10][10];
   for (i=0; i<10; i++) {          for (i=0; i<10; i++) {
s1   A[0] = ···;               s1    A1[i] = ···;
     for (j=1; j<10; j++) {          for (j=1; j<10; j++) {
s2     A[j] = A[j-1] + ···;    s2      A2[i][j] = { if (j=1) A1[i];
     }                                          else A2[i][j-1]; }+ ···;
                                     }
```

After expansion, i.e. renaming array `A` in `A1` and `A2` then adding a dimension to array `A2`, the `for` loop is parallel. The instancewise reaching definition of the `A[j-1]` reference depends on the values of `i` and `j`, as implemented with a conditional expression.

```
int A[10];                      struct Tree {
void Proc (int i) {               int value; Tree *left, *right;
  A[i] = ···;                   } *p;
  ··· = A[i];                   void Proc (Tree *p, int i) {
  if (···) Proc (i+1);            p->value = ···;
  if (···) Proc (i-1);            ··· = p->value;
}                                 if (···) Proc (p->left, i+1);
                                  if (···) Proc (p->right, i-1);
                                }
```

After expansion, the two procedure calls can be executed in parallel. Memory allocation for the `Tree` structure is not shown.

Figure 1.1. Simple examples of memory expansion

by Collard and Griebl [GC95, Col98]. The argument of a ϕ function is the *set of possible reaching definitions* for the associated read reference.[1] Figure 1.2 shows two programs with some unknown conditional expressions and arrays subscripts. Expanded versions with ϕ functions are given in the right side of the figure.

Notice that memory expansion is not a mandatory step for parallelization; it is yet a general technique to expose parallelism in programs. Now, implementation of a parallel program depends on the target language and architecture. Two main techniques are used.

The first technique takes benefit of *control parallelism*, i.e. parallelism between different statements in the same program block. Its goal is to replace as many sequential executions of statements—denoted with ; in C—by parallel executions. Depending on the language, there are many different syntaxes to code this kind of parallelism, and all these syntaxes may not have the same expressive power. We will prefer the Cilk [MF98] **`spawn/sync`** syntax (similar to OpenMP's syntax) to the *parallel block* notation from Algol 68 or the EARTH-C compiler [HTZ+97]. As in [MF98], synchronizations involve

[1] This interpretation of ϕ functions is very different from their usual semantics in the SSA framework.

```
   int x;                          int x1, x2;
s1 x = ···;                     s1 x1 = ···;
s2 if (···) x = ···;            s2 if (···) x2 = ···;
r  ··· = x;                     r  ··· = φ({s1, s2});
```

After expansion, one may not decide at compile-time what value is read by statement r. One only knows that it may either come from s_1 or from s_2, and the effective value retrieval code is hidden in the $\phi(\{s_1, s_2\})$ function. It checks whether s_2 executed or not, then if it did, it returns the value of `x2`, else it returns the value of `x1`.

```
   int A[10];                      int A1[10], A2[10];
s1 A[i] = ···;                  s1 A1[i] = ···;
s2 A[···] = ···;                s2 A2[···] = ···;
r  ··· = A[i];                  r  ··· = φ({s1, s2});
```

After expansion, one may not decide at compile-time what value is read by statement r, because one does not know which element of array `A` is assigned by statement s_2.

Figure 1.2. Run-time restoration of the flow of data

every asynchronous computation started in the *surrounding program block*, and *implicit synchronizations* are assumed at return points in procedures. For the example in Figure 1.3.a, execution of A, B, C in parallel followed sequentially by D and E has been written in a Cilk-like syntax (each statement would probably be a procedure call).

```
spawn A;
spawn B;
spawn C;
sync;
// wait for A, B and C to complete
D;
E;
```

Figure 1.3.a. Control parallelism

```
// L is the latency of the schedule
for (t=0; t<=L; t++) {
   parallel for (ı ∈ F(t))
     execute instance ı
   // implicit synchronization
}
```

Figure 1.3.b. Data parallel implementation for schedules

Figure 1.3. Exposing parallelism

The second technique is based on *data parallelism*, i.e. parallelism between different instances of the same statement or block. The data parallel programming model has been extensively studied in the case of loop nests [PD96], because it is very well suited to efficient parallelization of numerical algorithms and repetitive operations on large data sets. We will consider a syntax similar to OpenMP parallel loop declaration, where all variables are supposed to be *shared* by default, and an *implicit synchronization* takes place at each parallel loop termination.

The first algorithms to generate data parallel code were based on intuitive loop transformations such as loop fission, loop fusion, loop interchange, loop reversal, loop skewing, loop reindexing and statement reordering. Moreover, dependences abstractions were much less expressive than affine relations. But data parallelism is also appropriate when describing a parallel order with a *schedule*, i.e. giving an execution date for every statement instance. The program pattern in Figure 1.3.b shows the general implementation of such a schedule [PD96]. It is based on the concept of *execution front* $F(t)$ which gathers all instances $\imath$ executing at date t.

The first scheduling algorithm was designed by Allen and Kennedy [AK87], from which many other methods have been designed. These are all based on a rather approximative abstractions of dependences, like dependence levels, vectors and cones. Despite the lack of generality, the benefit of such methods is the low complexity and easy implementation in a industrial parallelizaing compiler; see the work by Banerjee [Ban92] or more recently by Darte and Vivien [DV97] for a survey of these algorithms.

The first general solution to the scheduling problem was proposed by Feautrier [Fea92]. The proposed algorithm is very useful, but its weak point is the lack of help to decide what parameter of the schedule to optimize: is it the latency L, the number of communications (on a distributed memory machine), the width of the fronts?

Eventually, it is well known that control parallelism is more general than data parallelism, meaning that every data parallel program can be rewritten in a control parallel model, without loosing any parallelism. This is especially true for recursive programs, for which the distinction between the two paradigms becomes very unclear, as shown in [Fea98]. However, for practical programs and architectures, it has long been the case that architectures for massively parallel computations were much more suited to data parallelism, and that getting good speed-ups on such architectures was difficult with control parallelism—mainly due to asynchronous task management overhead. But recent advances in hardware and software systems are showing an evolution in this situation: excellent results for parallel recursive programs (game simulations like chess, and sorting algorithms) have been shown with Cilk for example [MF98].

1.3 Thesis Overview

This thesis is organized in four chapters and a final conclusion. Chapter 2 describes a general framework for program analysis and transformation, and presents the formal definitions useful to the following chapters. The main interest of this chapter is to encompass a very large class of programs, from nests of loops with arrays to recursive programs and data structures.

A collection of mathematical results is gathered in Chapter 3; some are rather well known, such as Presburger arithmetcis and formal language theory; some are very uncommon in compiler and parallelism fields, such as rational and algebraic transductions; and the others are mostly contributions, such as left-synchronous transductions and approximation techniques for rational and algebraic transductions.

Chapter 4 addresses instancewise analysis of recursive programs. Based on an extension of the induction variable concept to recursive programs and on new results in formal language theory, it presents two algorithms for dependence and reaching definition analysis. These algorithms are applied to several practical examples.

Parallelization techniques based on memory expansion are studied in Chapter 5. The first three sections present new techniques to expand nested loops with unrestricted conditionals, bounds and array subscripts; the fourth section is a contribution to simultaneous optimization of expansion and parallelization parameters; and the fifth section presents our results about parallelization of recursive programs.

Chapter 2

Framework

The previous introduction and motivation has covered several very different concepts and approaches. Each one has been studied by many authors who have defined their own vocabulary and abstractions. Of course, we would like to keep the same formalism along the whole presentation. This chapter presents a framework for *describing* program analysis and transformation techniques and for *proving* their correctness or theoretical properties. The design of this framework has been governed by three major goals:

1. build on well defined concepts and vocabulary, while keeping the continuity with related works;
2. focus on instancewise properties of programs, and take benefit of this additional information to design new transformation techniques;
3. head for both generality and high precision, minimizing the necessary number of tradeoffs.

This presentation does not compete with other formalisms, some of which are firmly rooted in semantically and mathematically sound theories [KU77, CC77, JM82, KS92]. Because we advocate for *instancewise* analysis and transformations, we primarily focused on establishing convincing results about effectiveness and feasibility. This required leaving for further studies the necessary integration of our techniques in a more traditional analysis theory. We are sure that instancewise analysis can be modeled in a formal framework such as abstract interpretation, even if very few works have addressed this important issue.

We start with a formal presentation of run-time statement *instances* and program *executions* in Section 2.1, then the program model we will consider throughout this study is exposed and motivated in Section 2.2. Section 2.3 proposes mathematical abstractions for these instance and program models. Program analysis and transformation frameworks are addressed in Sections 2.4 and 2.5 respectively.

2.1 Going Instancewise

During program execution, each statement can be executed several times, depending on the surrounding control structures (loops, procedure calls and conditional expressions). To capture data-flow information as precisely as possible, our analysis and transformation techniques should be able to distinguish between the *distinct executions* of a *statement*.

Definition 2.1 (instance) For a statement s, a *run-time instance* of s is some particular execution of s during execution of the program.

For short, a *run-time instance* of a *statement* is called an *instance*. If the program terminates, each statement has a finite number of *instances*.

Consider the two example programs in Figure 2.1. They both display the sum of an array `A` with an unknown number N of elements; one is implemented with a loop and the other with a recursive procedure. Statements B and C are executed N times during execution of each program, but statements A and D are executed only once. The value of variable `i` can be used to "name" each instance of B and C and to distinguish at compile-time between the $2N+2$ run-time instances of statements A, B, C and D: the unique instances of statements A and D are denoted respectively by $\langle A \rangle$ and $\langle C \rangle$, and the N instances of statement B (resp. statement C) associated with some value i of variable `i` are denoted by $\langle B, i \rangle$ (resp. by $\langle C, i \rangle$), $0 \le i < N$. Such an "iteration variable" notation is not always possible, and a general naming scheme will be studied in Section 2.3.

```
   int A[N];
   int c;
A  c = 0;
   for (i=0; i<N; i++) {
B    c = c + A[i];
   }
   printf ("%d", c);
```

```
   int A[N];
   int Sum (int i) {
     if (i<N)
C      return A[i] + Sum (i+1);
     else
D      return 0;
   }
   printf ("%d", Sum (0));
```

Figure 2.1. About run-time instances and accesses

Because of the state of memory and possible interactions with its environment, several executions of the same program may yield different sets of run-time statement instances and incompatible results. We will not formally define this concept of program execution in operational semantics: a very clean framework has indeed been defined by Cousot and Cousot [Cou81] for *abstract interpretation*, but the correctness of our analysis and transformation techniques does not require so many details.

Definition 2.2 (program execution) Let P be a program. A *program execution* e is given by an *execution trace* of P, which is a finite or infinite (when the program does not terminate) sequence of *configurations*—i.e. machine states. The set of all possible program executions is denoted by $\mathbf{E}$.

Now, the set of all run-time instances for a given program execution $e \in \mathbf{E}$ is denoted by $\mathbf{I}_e$. Subscript e denotes a given program execution, but it also recalls that set $\mathbf{I}_e$ is "exact": it is the effective unapproximate set of statement instances executed during program execution e. This formalism will be used in every further definition of execution-dependent concept.

Considering again the two programs in Figure 2.1, the execution of statements B and C is governed by a comparison of variable `i` with the constant N. Without any information on the possible values of N, it is impossible to decide at compile-time whether some instance of B or C executes. In the extreme case of an execution e where N is equal to zero, both statements are never executed, and the set $\mathbf{I}_e$ is equal to $\{\langle A \rangle, \langle D \rangle\}$. In general, $\mathbf{I}_e$ is equal to $\{\langle A \rangle, \langle D \rangle\} \cup \{\langle B, i \rangle, \langle C, i \rangle : 0 \le i < N\}$, the value of N being part of the definition of e.

Of course, each statement can involve several (including zero) *memory references*, at most one of these being a write (i.e. in left-hand side).

Definition 2.3 (access) A pair $(\imath, r)$ of *a statement instance* and a *reference in the statement* is called an *access*.

For a given execution $e \in \mathbf{E}$ of a program, the set of all accesses is denoted by $\mathbf{A}_e$. It can be decomposed into:

- $\mathbf{R}_e$, the set of all *reads*, i.e. accesses performing some load operation from memory;
- and $\mathbf{W}_e$, the set of all *writes*, i.e. accesses performing some store operation into memory.

Due to our syntactical restrictions, no access may be simultaneously a read and a write. Since a statement performing some write in memory involves exactly one reference in left-hand side, its *instances* are often used in place of its *write accesses* (this sometimes simplifies the exposition).

Looking again at our two programs in Figure 2.1:

- statement A has one write reference to variable `c`, the single associated access is denoted by $\langle A, \texttt{c}\rangle$;
- statement B has one write and one read reference to variable `c`, since both references are identical, the associated accesses are both denoted by $\langle B, i, \texttt{c}\rangle$, $0 \le i < N$;
- statement B has one read reference to array `A`, the associated accesses are denoted by $\langle B, i, \texttt{A[i]}\rangle$, $0 \le i < N$;
- statement C has one read reference to array `A`, the associated accesses are denoted by $\langle C, i, \texttt{A[i]}\rangle$, $0 \le i < N$;
- statement D has no memory reference, thus no associated access.

2.2 Program Model

Our framework focuses on imperative programs. This section describes the control and data structure syntax we consider. In a preliminary work [CCG96], we defined a toy language—called LEGS—which allowed explicit declaration of complex data structures shapes fitting our program model. Most of the program model restrictions we enumerate in this section were also enforced by the language semantics. We chose yet to define our program model with a C-like syntax (with C++ syntactic sugar facilities): despite the the lack of formal semantics available in C, we hope this choice will ease the understanding of practical examples and the communication of our new ideas.

2.2.1 Control Structures

Procedures are seen as functions returning the `void` type and explicit—typed—pointers are allowed. Multi-dimensional arrays are accessed with syntax `[`i_1`,`$\ldots$`,`i_n`]`—not C syntax—for better understanding.

Definition 2.4 (statement and block) A program *statement* is any C expression ended with "`;`" or "`}`". A program *block* is a special kind of statement that starts with "`{`", a function declaration, a loop or a conditional expression, and surrounding one or more sub-statements.

To simplify the exposition, the only *control structures* that may appear in the right-hand side of an assignment, in a function call or in a loop declaration are *conditional statements*. Moreover, multiple expressions separated by `,` are not allowed, and loops are supposed to follow some minimal "code of ethics": each loop variable is affected by a single loop and its value is not used outside of this loop; as a consequence, each loop variable must be initialized.

This framework is primarily designed for first-order control structures: any function call should be fully specified at compile-time, and "computed" `goto`s are forbidden. But higher-order structures can be handled conservatively, in approximating the possible function calls using external analysis techniques [Cou81, Deu90, Har89, AFL95]. Calls to input/output functions are allowed as well, but completely ignored by analysis and transformation techniques, possibly yielding incorrect parallelizations.

Recursive calls, loops with unrestricted bounds, and conditional statements with unrestricted predicates are allowed. Classical exception mechanisms, `break`s, and `continue`s are supported as well. However, we suppose that `goto`s are removed by well known algorithms for structuring programs [Bak77, Amm92], at the cost of some code duplication in the rare cases where the control flow graph is not reducible [ASU86].

2.2.2 Data Structures

We only consider

- scalars (boolean, integer, floating-point, pointer...);
- records (non-recursive and non-array structure with scalar and record fields);
- arrays of scalars or records;
- trees of scalars or records;
- arrays of trees;
- and trees of arrays.

Records are seen as compound scalars with *unaliased named fields*. Moreover, unrestricted array values in trees and tree elements in arrays are allowed, including recursive nestings of arrays and trees.

Arrays are accessed through the classical syntax, and other data structures are accessed through the use of explicit pointers. However, to simplify the exposition, we suppose that no variable is simultaneously used as a pointer (through operators `*` and `->`) and as an array (through operator `[]`): in particular, explicit array subscripts must be preferred to pointer arithmetic.

By convention, *edge names* in trees are identical to the label of pointer fields in the tree declaration.

In practical implementations, recursive data structures are not made explicit. More precisely, two main problems arise when trying to build an abstract view of data structure definition and usage in C programs.

1. Multiple structure declarations may be relative to the same data structure, without explicit declaration of the shape of the whole object. Moreover, even a single recursive `struct` declaration can describe several very different objects, such as lists, doubly-linked lists, trees, acyclic graphs, general graphs, etc. Building a compile-time abstraction of data structures used in a program is thus a difficult problem, but it is essential to our analysis and transformation framework. It can be achieved in two opposite ways: either "decorating" the C code with *shape descriptions* which guide the compiler when building its abstract view of data structures [KS93, FM97, Mic95, HHN92] or running a compile-time *shape analysis* of pointer-based structures [GH96, SRW96].

2. Two pointer variables may be *aliased*, i.e. they may be two different names for the same memory location. The goal of alias analysis [Deu94, CBC93, GH95] (store-less) and points-to analysis [LRZ93, EGH94, Ste96] (store-based) techniques is precisely to disambiguate pointer accesses, when pointer updates are not too complex to be analyzed. In practice, one may expect good results for *strongly typed* programs *without pointer arithmetics*, especially if the goal of the alias analysis is to check whether two pointers refer the same structure or not. Element-wise alias analysis is very costly and still a largely open problem: indeed, no instancewise alias analysis for pointers has been proposed so far, and it could be an interesting future development of our framework.

In the following, we thus suppose that the shape of each data structure has been identified as one of the supported data types, and that each pointer reference has been associated the data structure instance it refers to.

Now, there is one last question about data structures: how are they constructed, modified and destroyed? When dealing with arrays, a compile-time shape declaration is available in most cases; but some programs require *dynamic arrays* whose size is updated dynamically every time an out-of-bound access is detected: this is the case of some *expanded programs* studied in Chapter 5. The problem is more critical with pointer-based data structures: they are most of the time allocated at run-time with explicit `malloc` or `new` operations. This problem has already been addressed by Feautrier in [Fea98] and we consider the same abstraction: all data structures are supposed to by built to their maximal extent—possibly infinite—in a preliminary part of the code. To guarantee that this abstraction is correct regarding data-flow information, we must add an additional restriction to the program model: *any run-time insertion and deletion is forbidden*. In fact there are two exceptions to this very strong rule, but they will be described in the next section after presenting the mathematical abstraction for data structures. Nevertheless, a lot of interesting programs with recursive pointer-based structures perform random insertions and deletions, and these programs cannot be handled at present in our framework. This issue is left for future work.

2.3 Abstract Model

We start with a presentation of a naming scheme for statement instances, and show that execution traces are not suitable to our purpose. Then, we propose a powerful abstraction for memory locations.

2.3.1 Naming Statement Instances

In the following, *every program statement* is supposed to be labeled. The alphabet of ***statement labels*** is denoted by Σ_{CTRL}. Now, loops and conditionals requires special attention.

- Because a loop involves an initialization step, a bound check step, and an iteration step, loops are given *three* labels: the first one represents the loop entry, the second one is the check for termination, and the third one is the loop iteration. Remember that, in C, a bound check is performed immediately after the loop entry and immediately after each increment. The loop *check* is considered as a *block* and a *conditional statement*, and the two other are *non-block* labels.

- An `if` $\cdots$ `then` $\cdots$ `else` $\cdots$ statement is given two labels: one for the condition and the `then` branch, and one for the `else` branch. Both labels are considered as *block* labels.

Consider the program example in Figure 2.2.a. This simple recursive procedure computes all possible solutions to the n-Queens problem, using an array `A` (details of the code are omitted here); it is our running example in this section.

There are two assignment statements: s writes into array `A` and r performs some read access in `A`. Statement I and J are conditionals, and statement Q is a recursive call to procedure `Queens`. Loop statements are divided into three sub-statements which are given distinct labels: the first one denotes the loop *entry*—e.g. A or B—the second one denotes the bound check—e.g. $\textsc{a}$ or $\textsc{b}$—and the third one denotes the loop *iteration*—e.g. a or b. Finally, P is the label of the procedure and F denotes the initial call in `main`.

A primary goal for instancewise analysis and transformation is to name each statement instance. To achieve this, many works in the program analysis field rely on *execution traces*. Their interpretation for program analysis is generally defined as a path from the entry of the *control flow graph* to a given statement.[1] They record every execution of a statement, including return from functions.

For our purpose, these execution traces have three main drawbacks:

1. because of return labels, traces belong to a non-rational language in Σ^*_{CTRL}, as soon as there are recursive function calls;

2. full-length traces are huge and extremely redundant: if an instance executes before another in the same program execution, its trace prefixes the other;

3. a single statement instance may have several execution traces because statement execution is unknown at compile time.

To overcome the first problem, a classical technique relies on a function called $\textsc{Net}$ on Σ^*_{CTRL} [Har89]: intuitively this function collapses all call-return pairs in a given execution trace, yielding compact *rational* sets of execution traces. The third point is much more unpleasant because it forbids to give a unique name to each statement instance. Notice however that different execution traces for the same instance must be associated with distinct executions of the program.

[1] Without notice of conditional expressions and loop bounds.

```
       int A[n];
P      void Queens (int n, int k) {
I        if (k < n) {
A/A/a      for (int i=0; i<n; i++) {
B/B/b        for (int j=0; j<k; j++)
r              ··· = ··· A[j] ···;
J            if (···) {
s              A[k] = ···;
Q              Queens (n, k+1);
             }
           }
         }
       }

       int main () {
F        Queens (n, 0);
       }
```

Figure 2.2.a. Procedure `Queens`

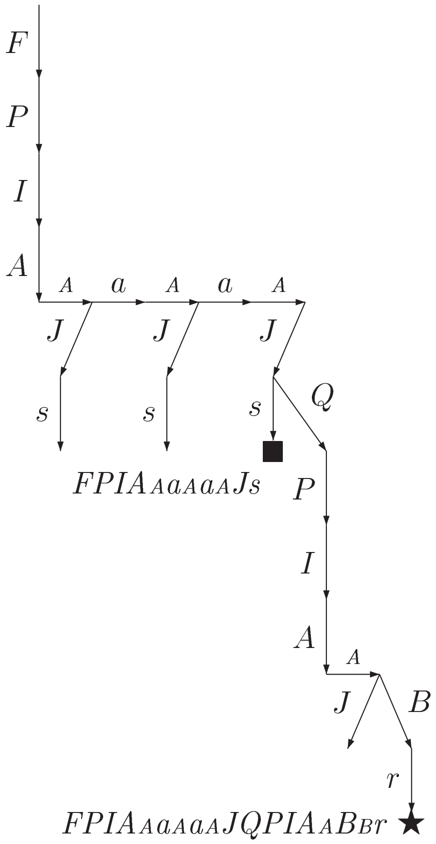


Figure 2.2.b. Control tree

Figure 2.2. Procedure `Queens` *and control tree*

Our solution starts from another representation of the program flow: the intuition behind our naming scheme for instances is to consider some kind of "extended stack states" where loops are seen as special cases of recursive procedures. The dedicated vocabulary for this representation has been defined in parts and with several variations in [CC98, Coh99a, Coh97, Fea98].

Let us start with an example: the first instance of statement s in procedure `Queens`. Depending on the number of iterations of the innermost loop—bounded by k—an execution trace for this first instance can be one of $FPIA\textsc{a}B\textsc{b}Js$, $FPIA\textsc{a}B\textsc{b}b\textsc{b}Js$, $FPIA\textsc{a}B\textsc{b}b\textsc{b}b\textsc{b}Js$, $\ldots$, $FPIA\textsc{a}B\textsc{b}(b\textsc{b})^k Js$. Since we would like to give a unique name to the first instance of s, all B, $\textsc{b}$ and b labels should intuitively be left out. Now, for a given program execution, any statement instance is associated with a *unique* (ordered) list of block enterings, loop iterations and procedure calls leading to it. To each list corresponds a word: the concatenation of statement labels. This is precisely what we get when forgetting about the innermost loop in execution traces of the first instance of statement s: the *single* word $FPIA\textsc{a}Js$. These concepts are illustrated by the tree in Figure 2.2.b, to be defined later. We now formally describe these words and their relation with statement instances.

Definition 2.5 (control automaton and control words) The *control automaton* of the program is a *finite-state automaton* whose states are statements in the program and where a transition from a state q to a state q' express that statement q' occurs in block q. Such a transition is labeled by q'. The initial state is the statement executed at the beginning of program execution, and *all states are final.*

Words accepted by the control automaton are called ***control words***. By construction, they build a *rational language* L_{CTRL} included in Σ^*_{CTRL}.

Lemma 2.1 $\mathbf{I}_e$ being the set of statement instances for a given execution e of a program, there is a natural injection from $\mathbf{I}_e$ to the language L_{CTRL} of control words.

Proof: Any statement instance in a program execution is associated with a unique list of block enterings, loop iterations and procedure calls leading to it. We can thus define a function f from $\mathbf{I}_e$ to $\Sigma^{\mathbb{N}}_{\text{CTRL}}$—lists of statements labels—mapping statement instances to their respective list of block enterings, loop iterations and procedure calls.

Consider an instances $\imath_1$ of a statement s_1 and an instance $\imath_2$ of a statement s_2, and suppose $f(\imath_1) = f(\imath_2) = l$. By definition of f, both statements s_1 and s_2 must be part of the same program block B, and precisely, the last element of l is B. Considering a pair of a statement s and an instance $\imath$ of s, this proves that no other instance $\imath'$ of a statement s' may be such that $(f(\imath), s) = (f(\imath'), s')$.

Consider a function ψ from $\mathbf{I}_e$ to L_{CTRL}—control words—which maps an instance $\imath$ of a statement s to the concatenation of all labels in $f(\imath)$ and s itself. Thanks to the preceding property on pairs $(f(\imath), s)$, function ψ is injective. ■

Theorem 2.1 Let $\mathbf{I}$ be the union of all sets of statement instances $\mathbf{I}_e$ for every possible execution e of a program. There is a natural injection from $\mathbf{I}$ to the language L_{CTRL} of control words.

Proof: Consider two executions e_1 and e_2 of a program. The function defined in the proof of Lemma 2.1 is denoted by ψ_1 for execution e_1 and ψ_2 for execution e_2. If an instance $\imath$ is part of both $\mathbf{I}_{e_1}$ and $\mathbf{I}_{e_2}$ of a program, control words $\psi_1(\imath)$ and $\psi_2(\imath)$ are the same, because the list of block enterings, loop iterations and function calls leading to $\imath$ are unchanged. Lemma 2.1 terminates the proof. ■

We are thus allowed to talk about "the control word of a statement instance". In general, the set $\mathbf{E}$ of possible program executions and the set $\mathbf{I}_e$ for $e \in \mathbf{E}$ are unknown at compile-time, and we may consider all instances that *may* execute during *any* program execution. Eventually, the natural injection becomes a one-to-one mapping when extending the set $\mathbf{I}_e$ with all possible instances associated to "legal" control words. As a consequence, if w is a control word, we will say "instance w" instead of "the instance whose control word is w".

We are also interested in encoding *accesses* themselves with control words. A simple solution consists in considering pairs $(w, \texttt{ref})$, where w is a control word for some instance of a statement s and `ref` is a reference in statement s. But we prefer to encode the full access "inside" the control word: we thus extend the alphabet of statement labels Σ_{CTRL} with letters of the form $s_{\texttt{ref}}$, for all statement $s \in \Sigma_{\text{CTRL}}$ and reference `ref` in s. Of course, extended labels may only take place as the *last letter* in a control word: when the last letter in a control word w is of the form $s_{\texttt{ref}}$, it means that w represents an access instead of an instance. However, when clear from the context, i.e. when there is only one "interesting" reference in a given statement or all references are identical, the reference will be taken out of the control word of accesses. This will be the case in most practical examples.

Eventually, notice that some states in the control automaton have exactly *one* incoming transition and *one* outgoing transition (looping transitions count as both incoming and outgoing). Now, these states do not carry any information about where a statement can be reached from or lead to: in every control word, the label of the outgoing transition follows the label of the incoming one. In practice, we often consider a ***compressed*** control automaton where all states with exactly one incoming transition and one outgoing transition are removed. This transformation has no impact on control words.

Observe that loops in the program are represented by looping transitions in the compressed control automaton, and that cycles involving more than one state are associated with recursive calls.

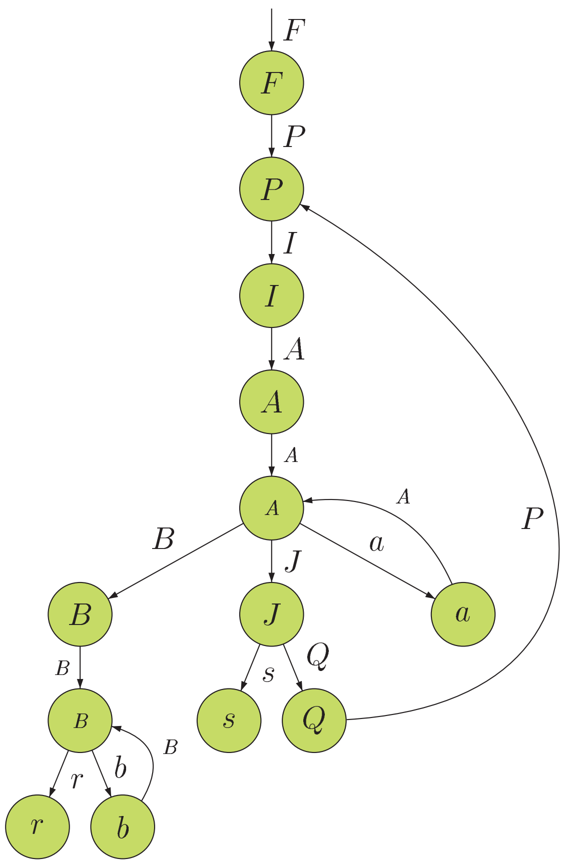


Figure 2.3.a. Control automaton

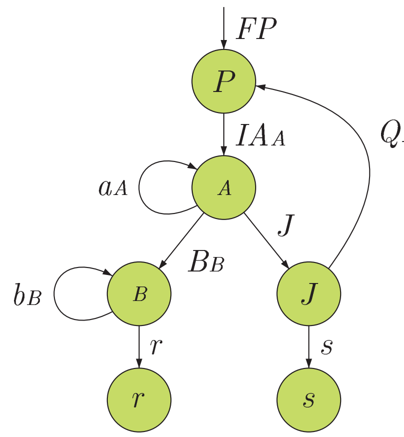


Figure 2.3.b. Compressed control automaton

Figure 2.3. Control automata for program `Queens`

Figure 2.3.a describes the plain control automaton for procedure `Queens`.[2] Since states F, I, A, B, Q, a and b are useless, they are removed along with their *outgoing* edges. The compressed automaton is described in Figure 2.3.b.

As a practical remark, notice that it is often desirable to restrict the language of control words to instances of a particular statement. This is easily achieved in choosing the state associated to this statement as the only final one.

To conclude this presentation of a naming scheme for statement instances, it is possible to compare the execution traces of an instance $\imath$ and the control word of $\imath$. Indeed, the

[2] Every state is final, but this is not made explicit on the figure.

following property is quite natural: it results from the observation that traces of an instance may only differ in labels of statements that are not part of the list of block enterings, loop iterations and function calls leading to this instance.

Proposition 2.1 The control word of a statement instance is a sub-word of every execution trace of this instance.

2.3.2 Sequential Execution Order

The *sequential execution order* of the program defines a total order over instances, call it $<_{\text{SEQ}}$. In English, words are ordered by the *lexicographic order* generated by the alphabet order $a < b < c < \cdots$. Similarly, in any program one can define a partial *textual order* $<_{\text{TXT}}$ over statements: statements in the *same block* are sorted in *apparition order*, and statements appearing in *different blocs* are *mutually incomparable*.

Remember the special case of loops: the *iteration* label executes *after* all the statements inside the loop body, but *entry* and *check* labels are not comparable with these statements. For procedure `Queens` in Figure 2.2.a, we have $B <_{\text{TXT}} J <_{\text{TXT}} a$, $r <_{\text{TXT}} b$ and $s <_{\text{TXT}} Q$.

This textual order generates a lexicographic one on control words, denoted by $<_{\text{LEX}}$:

$$w' <_{\text{LEX}} w \iff \begin{array}{l} \exists x, x' \in \Sigma_{\text{CTRL}}, u, v, v' \in \Sigma^*_{\text{CTRL}} : w = uxv, w' = ux'v', x' <_{\text{TXT}} x \\ \vee \quad \exists v' \in \Sigma^*_{\text{CTRL}} : w = w'v \qquad \text{(a.k.a. prefix order).} \end{array}$$

This order is only partial on Σ^*_{CTRL}. However, by construction of the textual order:

Proposition 2.2 An instance $\imath'$ executes before an instance $\imath$ iff their respective control words w' and w satisfy $w' <_{\text{LEX}} w$.

Notice that the lexicographic order $<_{\text{LEX}}$ is not total on L_{CTRL} because both cases on a conditional are not comparable! This does not yield a contradiction, because the `then` and `else` cases of the same `if` instance are never simultaneously executed in a single execution. In general, the lexicographic order is total on the subset of control words corresponding to instances that *do* execute—in one-to-one mapping with $\mathbf{I}_e$ for some execution $e \in \mathbf{E}$.

Eventually, the language of control words is best understood as an infinite tree, whose root is named ε and every edge is labeled by a statement. Each node then corresponds to the control word equal to the concatenation of edge labels starting from the root. Consider a control word ux, $u \in \Sigma^*_{\text{CTRL}}$ and $x \in \Sigma_{\text{CTRL}}$; every downward edge from a node whose control word is ux corresponds to an outgoing transition from state x in the control automaton. To represent the lexicographic order, downward edges are ordered from left to right according to the textual order. Such a tree is usually called a *call tree* in the functional languages community, but *control tree* is more adequate in the presence of loops and other non-functional control structures. One may talk about plain and *compressed* control trees, dependending on the control automaton which defines them.

A partial control tree for procedure `Queens` is shown in Figure 2.2.b (a compressed one will be studied later in Figure 4.1 page 124). Control word $FPIAAaAaAJQPIAABBr$ is a possible run-time instance of statement r—depicted by a star in Figure 2.2.b, and control word $FPIAAaAaAJs$—depicted by a black square—is a possible run-time instance of statement s.

2.3.3 Adressing Memory Locations

A large number of data structure abstractions have been designed for the purpose of program analysis. This presentation can be seen as an extension of several frameworks we already proposed [CC98, Coh99a, Coh97, Fea98] some of which in collaboration with Griebl [CCG96], but it is also highly relevant to previous work by Alabau and Vauquelin [Ala94], by Giavitto, Michel and Sansonnet [Mic95], by Deutsch [Deu92] and by Larus and Hilfinger [LH88].

With no surprise, array elements are addressed by integers, or vectors of integers for multi-dimensional ones. Tree adresses are concatenation of edge names (see Section 2.2.2) starting from the root. The address of the root is simply ε, the zero-length word. For example, the name of node `root->l->r` in a binary tree is lr. The set of edge names is denoted by Σ_{DATA}. The layout of trees in memory is thus described by a *rational language* $L_{\text{DATA}} \subset \Sigma_{\text{DATA}}^*$ over edge names.

For the purpose of dependence analysis, we are looking for a mathematical abstraction which captures *relations between integer vectors*, *between words*, and *between the two*. Dealing with trees only, Feautrier proposed to use *rational transductions* between free monoids in [Fea98]. We will formally define such transductions in Section 3.3, and then show how the same idea can can be extended to more general classes of transductions and monoids, to handle arrays and nested trees and arrays as well.

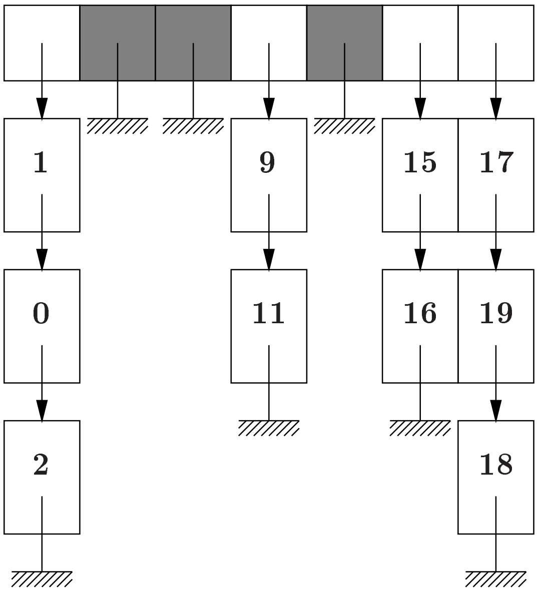
Extending the Data Structure Model

Some interesting structures are basically tree structures enhanced with traversal edges. In many cases, these traversal edges have a very regular structure. Most usual cases are reference to the parent and links between nodes at the same height in a tree. Such traversal edges are often used to facilitate special-purpose traversal algorithms. There is some support for such structures when traversal edges are known functions of the generators of the tree structure [KS93, FM97, Mic95], i.e. the "back-bone" spanning tree of the graph. In such a case, traversal edges are merely an "algorithmical sugar" for better performance. But even though, our support is limited since recursion and iteration over traversal edges is not supported. We will not study this extension any further because a full chapter would be necessary and our support for traversal edges does not include recursion and iteration.

Abstract Memory Model

The key idea to handle both arrays and trees is that they share a common mathematical abstraction: the monoid. For a quick recall of monoid definitions and properties, see Section 3.2. Indeed rational languages (tree addresses) are subsets of *free monoids* with word concatenation, and sets of integer vectors (array subscripts) are *free commutative monoids* with vector addition. The monoid abstraction for a data structure will be denoted by M_{DATA}, and the subset of this monoid corresponding to valid elements of the structure will be denoted by L_{DATA}.

The case of nested arrays and trees is a bit more complex but reveals the expressiveness of monoid abstractions. Our first example is the hash-table structure described in Figure 2.4. It defines an array whose elements are pointers to lists on integers. A monoid abstraction M_{DATA} for this structure is generated by $\mathbb{Z} \cup \{\mathtt{n}\}$, and its binary operation $\bullet$




```
struct key {
  // value of key
  int value;
  // next key
  key *n;
};

key *hash[7];
```

Figure 2.4. Hash-table declaration

is defined as follows:

$$\begin{aligned}
& & \mathtt{n} \bullet \mathtt{n} &= \mathtt{nn} & & (2.1)\\
\forall i \in \mathbb{Z} &: & i \bullet \mathtt{n} &= i\mathtt{n} & & (2.2)\\
\forall i \in \mathbb{Z} &: & \mathtt{n} \bullet i &= \mathtt{n}i & \text{(never used for the hash-table)} & (2.3)\\
\forall i, j \in \mathbb{Z} &: & i \bullet j &= i + j. & & (2.4)
\end{aligned}$$

The set $L_{\text{DATA}} \subset M_{\text{DATA}}$ of valid memory locations in this structure is thus

$$L_{\text{DATA}} = \mathbb{Z}\mathtt{n}^*.$$

Check that the third case in the definition of operation $\bullet$ is never used in L_{DATA}.

Our second example is the structure described in Figure 2.5. It defines an array whose elements are references to other arrays or integers. Each array is either *terminal* with integer elements or *intermediate* with array reference elements. This definition is very similar to file-system storage structures, such as UNIX's `inodes`. The monoid abstraction M_{DATA} for this structure is the same as the hash-table one. However, the set $L_{\text{DATA}} \subset M_{\text{DATA}}$ of valid memory locations in this structure is now

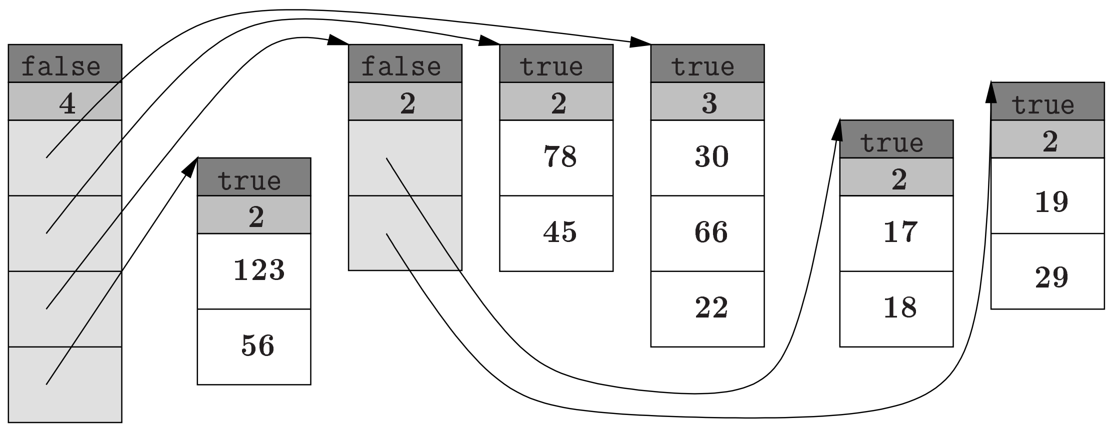
$$L_{\text{DATA}} = (\mathbb{Z}\mathtt{n})^*\mathbb{Z}.$$

Now the definition of operation $\bullet$ is the same as for the hash-table structure, see (2.1).

In the general case of nested arrays and trees, the monoid abstraction is generated by the union of node names in trees and integer vectors. Its binary operation $\bullet$ is defined as word concatenation with additional commutations between vectors of the same dimension. The result is called a free partially commutative monoid [RS97b]:

Definition 2.6 (free partially commutative monoid) A *free partially commutative monoid* M with binary operation $\bullet$ is defined as follows:

- generators of M are letters in an alphabet A and all vectors from a finite union of free commutative monoids of the form $\mathbb{Z}^n$;




```
struct inode {
  // true means terminal array of integers
  // false means intermediate array of pointers
  boolean terminal
  // array size
  int length
  union {
    // array of block numbers
    int a[];
    // array of inode pointers
    inode *n[];
  }
} quad;
```

Figure 2.5. An `inode` declaration

- operation $\bullet$ coincides with word concatenation on A^*, $\forall x, y \in A : x \bullet y = xy$;
- for a given integer n, operation $\bullet$ coincides with vector addition on $\mathbb{Z}^n$, $\forall x, y \in \mathbb{Z}^n : x \bullet y = x + y$.

This framework clearly supports *recursively* nested trees and arrays.

In the following, we abstract any data structure as a subset L_{DATA} of the monoid M_{DATA} with binary operation $\bullet$. ($\bullet$ denotes word concatenation for trees and usual sum for arrays.)

Eventually, we have required in the previous section that *no run-time insertion or deletion* appeared in the program. This rule is indeed too conservative, and two exceptions can be handled by our framework.

1. Because it makes no difference for the flow of data whether the insertion is done before the program or during execution—only assignment of the value does matters—*insertions* at a *list's tail* or *tree's leaf* are supported.
2. The abstraction is still correct when *deletions* at a list's tail or tree's leaf are supported, but may lead to overly conservative results. Indeed, suppose an insertion

follows a deletion at the tail of a list. Considering words in the free monoid abstraction of the list, the memory location of the tail node before deletion will be aliased with the new location of the inserted one.

2.3.4 Loop Nests and Arrays

The case of nested loops with scalar and array operations is very important. It applies to a wide range of numerical, signal-processing, scientific, and multi-media codes. A large amount of work has been devoted to such programs (or program fragments), and very powerful analysis and transformation techniques have been crafted. While the framework above easily captures such programs, it seems both easier and more natural to use another framework for memory addressing and instance naming. Indeed, we prefer the natural addressing scheme in arrays, using integers and integer vectors, because $\mathbb{Z}$-modules have a much richer structure than plain commutative monoids.

To ensure consistency of the control word and integer vector frameworks, we show how control words can be embedded into vectors. This embedding is based on the following definition, introduced by Parikh [Par66] to study properties of algebraic subsets of free commutative monoids:

Definition 2.7 A Parikh mapping over alphabet $\Sigma_{\textsc{ctrl}}$ is a function from words over $\Sigma_{\textsc{ctrl}}$ to integer vectors in $\mathbb{N}^{\mathrm{Card}(\Sigma_{\textsc{ctrl}})}$, such that each word w is mapped to the vector of occurrence count of every label in w.

There is no specific order in which labels are mapped to dimensions, but we are interested in a particular mapping where dimensions are ordered from the label of the outer loop to the label of the inner one.

The loop nest structure is non-recursive, hence the only cycles in the control automaton are transitions looping on the same state. As a result, the language of control words is in one-to-one mapping with its set of Parikh vectors. The following mapping is computed for the loop nest in Figure 2.6:

$$\begin{array}{rcl} A\textsc{a}(a\textsc{a})^*\big(B\textsc{b}(b\textsc{b})^*s + C\textsc{c}(c\textsc{c})^*r\big) & \longrightarrow & \mathbb{N}^{11} \\ w & \longmapsto & \big(|w|_A, |w|_{\textsc{a}}, |w|_a, |w|_B, |w|_{\textsc{b}}, |w|_b, \\ & & |w|_C, |w|_{\textsc{c}}, |w|_c, |w|_s, |w|_r\big). \end{array}$$

Respective Parikh vectors of instances $A\textsc{a}a\textsc{a}a\textsc{a}a\textsc{a}a\textsc{a}B\textsc{b}b\textsc{b}b\textsc{b}s$ and $A\textsc{a}a\textsc{a}a\textsc{a}C\textsc{c}c\textsc{c}c\textsc{c}c\textsc{c}r$ are $(1,5,4,1,2,2,0,0,0,1,0)$ and $(1,4,3,0,0,0,1,4,3,0,1)$.

```
A/A/a for (i=0; i<100; i++) {
B/B/b   for (j=0; j<100; j++)
s         A[i,j] = ···
C/C/c   for (k=0; k<100; k++)
r         ··· = A[i,k] ···
      }
```

Figure 2.6. Computation of Parikh vectors

From Parikh vectors, we build *iteration vectors* by removing all labels of non-iteration statements and collapsing all loops at the same nesting level in the same dimension. Doing this, there is a one-to-one mapping between Parikh vectors and pairs built of iteration vectors and statement labels. Indeed, the statement label captures both the last non-zero component of the Parikh vector—i.e. the identity of the statement—and the identity of the surrounding loops—i.e. which dimension corresponds to which loop.

Continuing the example in Figure 2.6, the only remaining labels are a, b and c—i.e. labels of iteration statements—and labels b and c are collapsed together into the second dimension.

- Iteration vector of instance $A{\scriptstyle A}a{\scriptstyle A}a{\scriptstyle A}a{\scriptstyle A}aB{\scriptstyle B}b{\scriptstyle B}b{\scriptstyle B}s$ of statement s is $(4, 2)$.
- Iteration vector of instance $A{\scriptstyle A}a{\scriptstyle A}a{\scriptstyle A}C{\scriptstyle C}c{\scriptstyle C}c{\scriptstyle C}c{\scriptstyle C}r$ of statement r is $(2, 3)$.

In this process, the lexicographic order $<_{\text{LEX}}$ on control words is replaced by the lexicographic order on iteration vectors (the first dimensions having a higher priority than the last).

As a conclusion, Parikh mappings show that *iteration vectors*—the classical framework for naming instances in loop nests—are a special case of our general control word framework.

Because a statement instance cannot be reduced to an iteration vector, we introduce the following notations (these notations generalize the intuitive ones at the end of Section 2.1):

- $\langle S, x\rangle$ stands for the instance of statement S whose iteration vector is x;
- $\langle S, x, \mathtt{ref}\rangle$ stands for the access built from instance $\langle S, x\rangle$ and reference `ref`.

This does not imply that control words are a case of overkill when studying loop nests. In particular, they may still be useful when `goto`s and non-recursive function calls are considered. However, most interesting loop nest transformation techniques are rooted too deeply in the linear algebraic model to be rewritten in terms of control words. Further comparison is largely open, but some ideas and results are pointed out in Section 4.7.

2.4 Instancewise Analysis

Because our execution model is based on control words instead of execution traces, the previous Definition 2.2 of a program execution is not very practical. For our purpose, a *sequential execution* $e \in \mathbf{E}$ of a program is seen as a pair $(<_{\text{SEQ}}, f_e)$, where $<_{\text{SEQ}}$ is the sequential order over all *possible statement instances* (associated to the language of control words) and f_e maps every *access* to the memory location it either reads or writes. Notice that $<_{\text{SEQ}}$ is not dependent on the execution: it is defined as the order between all *possible* statement instances for all executions, which is legal because sequential execution is *deterministic*. Order $<_{\text{SEQ}}$ is thus *partial*, but its restriction to a set of instances $\mathbf{I}_e$ for a given execution $e \in \mathbf{E}$ is a *total* order. However, f_e clearly depends on the execution e, and its domain is exactly the set $\mathbf{A}_e$ of *accesses*.

Function f_e is the ***storage mapping*** for execution e of the program [CFH95, Coh99b, CL99]—it is also called *access function* [CC98, Fea98]. Storage mapping gathers the effect of every statement instance, for a given execution of the program. It is a function from the *exact* set $\mathbf{A}_e$ of accesses (see Definition 2.3) that actually *execute* into the set of memory locations.

In practice, the sequential execution order is explicitly defined by the program syntax, but it is not the case of the storage mapping. Some analysis has to be performed, either to compute $f_e(a)$ for all executions e and accesses a, or to compute approximations of f_e.

Eventually, $(<_{\text{SEQ}}, f_e)$ has been defined as a view of a specific program execution e, but it can also be seen as a function mapping $e \in \mathbf{E}$ to pairs $(<_{\text{SEQ}}, f_e)$. For the sake of simplicity, such a function—which defines all possible executions of a program—will be referred as "program $(<_{\text{SEQ}}, f_e)$" in the following.

2.4.1 Conflicting Accesses and Dependences

Many analysis and transformation techniques require some information on "conflicts" between memory accesses.

Definition 2.8 (conflict) Two accesses a and a' are in *conflict* if they access—either read or write—the same memory location: $f_e(a) = f_e(a')$.

This vocabulary is inherited from the cache analysis framework and its *conflict misses* [TD95]. Analysis of conflicting accesses is also very similar to *alias analysis* [Deu94, CBC93]. The *conflict relation* is the relation between conflicting accesses, and is denoted by κ_e for a given execution $e \in \mathbf{E}$. An exact knowledge of f_e and κ_e is impossible in general, since f_e may depend on the initial state of memory and/or input data. Thus, *analysis of conflicting accesses* consists in building a *conservative approximation* κ of the conflict relation, compatible with any execution of the program: $v \,\kappa\, w$ must hold when there is an execution e such that $v, w \in \mathbf{A}_e$ and $f_e(v) = f_e(w)$, i.e.

$$\forall e \in \mathbf{E}, \forall v, w \in \mathbf{A}_e : \quad \big(f_e(v) = f_e(w) \implies v \,\kappa\, w\big). \tag{2.5}$$

This condition is the only requirement on relation κ, but a precise approximation is generally hoped for. For most program analysis purposes, this relation only needs to be computed on writes, or between reads and writes, but other problems such as cache analysis [TD95] require a full computation.

Consider the example in Figure 2.7 where `FirstIndex` and `SecondIndex` are external functions on which no information is available. Because the sign of v is unknown at compile-time, the set of statement instances $\mathbf{I}_e$ can be either statement S or statement T (statements coincides with statement instances since they are not surrounded by any loop or procedure call), depending on the execution. Since the results of `FirstIndex` and `SecondIndex` are unpredictable too, no exact storage mapping can be computed at compile-time. The only available compile-time information is that S and T may execute, and then they may also yield conflicting accesses, i.e.

$$\langle S, \texttt{A[FirstIndex ()]}\rangle \,\kappa\, \langle T, \texttt{A[SecondIndex ()]}\rangle.$$

However, another information is that executions of S and T are mutually exclusive (due to the `if` $\cdots$ `then` $\cdots$ `else` $\cdots$ construct syntax), and then S and T cannot be conflicting accesses:

$$\nexists e \in \mathbf{E} : \quad S \in \mathbf{A}_e \wedge T \in \mathbf{A}_e.$$

This example shows the need for computing approximative results about data-flow properties such as conflicting accesses, and it also shows how complex it is to achieve precise results.

```
  int v, A[10];
  scanf ("%d", &v);
  if (v > 0)
S   A[FirstIndex ()] = ···
  else
T   A[SecondIndex ()] = ···
```

Figure 2.7. Execution-dependent storage mappings

For the purpose of parallelization, we need sufficient conditions to allow two accesses to execute in any order. Such conditions can be expressed in terms of *dependences*:

Definition 2.9 (dependence) An access a depends *on another* access a' if at least one is a write (i.e. $a \in \mathbf{W}_e$ or $a' \in \mathbf{W}_e$), if they are in conflict—i.e. $f_e(a) = f_e(a')$—and if a' executes before a—i.e. $a' <_{\text{SEQ}} a$.

The *dependence relation* for an execution e is denoted by δ_e: a depends on a' is written $a' \,\delta_e\, a$:

$$\forall e \in \mathbf{E}, \forall a, a' \in \mathbf{A}_e : \quad a' \,\delta_e\, a \overset{\text{def}}{\Longleftrightarrow} (a \in \mathbf{W}_e \vee a' \in \mathbf{W}_e) \wedge a' <_{\text{SEQ}} a \wedge f_e(a) = f_e(a'). \tag{2.6}$$

Once again, an exact knowledge of δ_e is impossible in general. Thus, *dependence analysis* consists in building a conservative approximation δ, i.e.

$$\forall e \in \mathbf{E}, \forall a, a' \in \mathbf{A}_e : \quad \big(a' \,\delta_e\, a \Longrightarrow a' \,\delta\, a\big). \tag{2.7}$$

Eventually, Bernstein's conditions tell that two accesses can be executed in any order—e.g. in parallel—if they are not dependent.

2.4.2 Reaching Definition Analysis

Some techniques require more precision than is available through dependence analysis: given a read access in memory, they need to identify the statement instance that produced the value. Then the read access is called the *use* and the instance that produced the value is called the "definition" that "reaches" the use, or *reaching definition*. The reaching definition is indeed the *last instance*—according to the execution order—on which the use depends.

We thus define function σ_e, mapping every read access to its reaching definition:

$$\forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e : \quad \sigma_e\,(u) = \max_{<_{\text{SEQ}}} \big\{ v \in \mathbf{W}_e : v \,\delta_e\, u \big\}; \tag{2.8}$$

or, replacing max with its definition:

$$\forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e, v \in \mathbf{W}_e : \quad v = \sigma_e\,(u) \overset{\text{def}}{\Longleftrightarrow} \\ v \,\delta_e\, u \wedge \big(\forall w \in \mathbf{W}_e : u <_{\text{SEQ}} w \vee w <_{\text{SEQ}} v \vee \neg(w \,\delta\, u)\big).$$

or, replacing δ_e with its definition (2.6):

$$\forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e, v \in \mathbf{W}_e : \quad v = \sigma_e(u) \overset{\text{def}}{\Longleftrightarrow} v <_{\text{SEQ}} u \wedge \big(\forall w \in \mathbf{W}_e : u <_{\text{SEQ}} w \vee w <_{\text{SEQ}} v \vee f_e(v) \neq f_e(w)\big).$$

So definition v reaches use u if it executes before the use, if both refer to the same memory location, and if no intervening write w kills the definition.

When a read instance u has no reaching definition, either u reads an uninitialized value (hinting at a programming error) or the analyzed program is only a part of a larger program. To cope with this problem, we add a virtual statement instance $\perp$ which executes before all instances in the program and assigns every memory location. Then, each read instance u has a *unique reaching definition*, which may be $\perp$.

Because no exact knowledge of σ_e can be hoped for in general, *reaching definition analysis* computes a conservative approximation σ. It is preferably seen as a relation, i.e.

$$\forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e, v \in \mathbf{W}_e : \quad \big(v = \sigma_e(u) \implies v\,\sigma\,u\big). \tag{2.9}$$

One may also use σ as a function from reads to sets of writes, and we talk about sets of *possible reaching definitions*. One must be very careful in the distinction between a set of effective instances $\mathbf{S} \subset \mathbf{I}_e$ and the set $\mathbf{S} \cup \{\perp\}$: if $\perp \notin \sigma(u)$ then it says that u reads a value produced by some instance in $\mathbf{S}$, but if $\perp \in \sigma(u)$ then u may read a value produced before executing the program. The fact that $\perp$ appears in a set of possible reaching definitions is the key to program checking techniques, since it may correspond to uninitialized values.

2.4.3 An Example of Instancewise Reaching Definition Analysis

This section is an overview of fuzzy array dataflow analysis (FADA); which was first presented in [CBF95]. The program model is restricted to loop nests with unrestricted conditionals, loop bounds and array subscripts. The aim of this short presentation is to allow comparison with our own analysis for recursive programs, and because the results of an instancewise reaching definition analysis for loop nests are extensively used in Chapter 5.

Intuitive Flavor

According to (2.8), the exact reaching definition of some read access u—$\sigma_e(u)$—is defined as the maximum of the set of writes in $\delta_e(u)$ (for a given program execution $e \in \mathbf{E}$). As soon as the program model includes conditionals, `while` loops, and `do` loops with non-linear bounds, we have to cope with a conservative approximation of the dependence relation. In the case of nested loops, one usually look for an *affine* relation, and non-affine constraints in (2.6) are approximated using additional analyses on variables and array subscripts.

But then, and with the exception of very special cases, computing the maximum of an approximate set of dependences has no meaning: the very execution of instances in $\delta(u)$ is not guaranteed. One solution is to take the entire set $\delta(u)$ as an approximation of the reaching definition. Can we do better than that? Let us consider an example. Notice first that, for expository reasons, only scalars are considered. The method, however, applies to arrays with any subscript.

```
for (i=0; i<N; i++) {
```

```
      if (···)
S1      x = ···;
      else
S2      x = ···;
   }
R  ··· = ··· x ···;
```

Assuming that $N \geq 1$, what is the reaching definition of reference `x` in statement R? Since all instances of S_1 and S_2 are in dependence with $\langle R \rangle$, it seems like we cannot do better that approximating $\sigma(\langle R \rangle)$ with $\{\langle S_1, 1\rangle, \ldots, \langle S_1, N\rangle, \langle S_2, 1\rangle, \ldots, \langle S_1, N\rangle\}$.

Let us introduce a new boolean function $b_e(i)$ which represents the outcome of the test at iteration i, for a program execution $e \in \mathbf{E}$. This allows to compute the *exact* dependence relation δ_e at compile-time:

$$
\begin{aligned}
&\forall e \in \mathbf{E}, \forall v \in \mathbf{W}_e: \\
&\qquad v\,\delta_e\,\langle R\rangle \iff \exists i \in \{1, \ldots, N\} : (v = \langle S_1, i\rangle \wedge b_e(i)) \vee (v = \langle S_2, i\rangle \wedge \neg b_e(i)),
\end{aligned}
$$

which can also be written

$$\forall e \in \mathbf{E}: \quad \delta_e(\langle R \rangle) = \{\langle S_1, i\rangle : 1 \leq i \leq N \wedge b_e(i)\} \cup \{\langle S_2, i\rangle : 1 \leq i \leq N \wedge \neg b_e(i)\}.$$

Since the above result is not approximate, the exact reaching definition $\sigma_e(\langle R\rangle)$ of $\langle R\rangle$ is the maximum of $\delta_e(\langle R\rangle)$.

Suppose $\sigma_e(\langle R\rangle)$ is an instance $\langle S_1, \beta_e^1\rangle$ for some execution $e \in \mathbf{E}$. Because $b_e(i) \vee \neg b_e(i)$ is equal to true for all $i \in \{1, \ldots, N\}$, any value produced by an instance $\langle S_1, i\rangle$ or $\langle S_2, i\rangle$ with $i < N$ is overwritten either by $\langle S_1, N\rangle$ or by $\langle S_2, N\rangle$. This proves that β_e^1 must be equal to N. Conversely, supposing $\sigma_e(\langle R\rangle)$ is an instance $\langle S_2, \beta_e^2\rangle$, the same reasoning proves that β_e^2 must be equal to N. Then, we have the following result for function σ_e:

$$\forall e \in \mathbf{E}: \quad \sigma_e(\langle R\rangle) = \{\langle S_1, N\rangle : b_e(N)\} \cup \{\langle S_2, N\rangle : \neg b_e(N)\}. \tag{2.10}$$

We may now replace b_e and $\neg b_e$ by their conservative approximations:

$$\sigma(\langle R\rangle) = \{\langle S_1, N\rangle, \langle S_2, N\rangle\}. \tag{2.11}$$

Notice here the high precision achieved.

To summarize these observations, our method will be to give new names to the result of maxima calculations in the presence of non-linear terms. These names are called *parameters* and are not arbitrary: as shown in the example, some properties on these parameters can be derived. More generally, one can find relations on non-linear constraints—like b_e—by a simple examination of the syntactic structure of the program or by more sophisticated techniques. These relations imply relations on the parameters, which are then used to increase the accuracy of the reaching definition. In some cases, these relations may be so precise as to reduce the "fuzzy" reaching definition to a singleton, thus giving an exact result. See [BCF97, Bar98] for a formal definition and handling of these parameters.

The general result computed by FADA is the following: the instancewise reaching definition relation σ is a *quast*, i.e. a nested conditional in which predicates are tests for the positiveness of quasi-affine forms (which include integer division), and leaves are either sets of instances whose iteration vector components are again quasi-affine, or $\perp$. See Section 3.1 for details about quasts.

Improving Accuracy

To improve the accuracy of our analysis, properties on non-affine constraints involved in the description of the dependences can be integrated in the data-flow analysis. As shown in the previous example, these properties imply properties on the *parameters* introduced in our computation.

Several techniques have been proposed to find properties on the variables of the program or on non-affine functions (see [CH78, Mas93, MP94, TP95] for instance). They use very different formalisms and algorithms, from pattern-matching to abstract interpretation. However, the relations they find can be written as first order formulas of additive arithmetic (a.k.a. Presburger arithmetics, see Section 3.1) on the variables and non-affine functions of the program. This general type of property makes the data-flow analysis algorithm independent of the practical technique involved to find properties.

How the properties are taken into account in the analysis is detailed in [BCF97, Bar98]. The quality of the approximation is defined w.r.t. the ability of the analysis to integrate (fully or partially) these properties. In general, the analysis cannot find the smallest set of possible reaching definitions [Bar98]. This is due to decidability reasons; but for some kind of properties, such as properties implied by the program structure, the best approximation can be found.

2.4.4 More About Approximations

Until then, every set of instances or accesses considered was exact and dependent on the execution. However, as hinted before, we will mostly consider approximative sets and relations in the following. For this reason, we need the following conservative approximations:

I, the set of all possible statement instances for every possible execution of a given program,

$$\forall e \in \mathbf{E}: \quad \big(\imath \in \mathbf{I}_e \Longrightarrow \imath \in \mathbf{I}\big);$$

A, the set of all possible accesses,

$$\forall e \in \mathbf{E}: \quad \big(a \in \mathbf{A}_e \Longrightarrow a \in \mathbf{A}\big);$$

R, the set of all possible reads,

$$\forall e \in \mathbf{E}: \quad \big(a \in \mathbf{R}_e \Longrightarrow a \in \mathbf{R}\big);$$

W, the set of all possible writes,

$$\forall e \in \mathbf{E}: \quad \big(a \in \mathbf{W}_e \Longrightarrow a \in \mathbf{W}\big).$$

They can be very conservative or be the result of a very precise analysis. In practice, the precision of these sets is not critical because they are rarely directly used in algorithms (but they are widely used in theoretical frameworks associated with these algorithms). Most of the time, they are implicitly present as domains or images of every relation over instances and accesses, which have their own dedicated analysis and approximation.

Sets **I**, **A**, **R**, **W** and relations κ, $\not\kappa$, δ, σ are the key to program analysis and transformation techniques. In our framework, no other instancewise information is available at compile-time. In particular, when we present an *optimality result* for some algorithm it means *optimality according to this information*: nobody can do a better job if his only informations are the sets and relations above.

2.5 Parallelization

With the model defined in Section 2.4, *parallelization* of some program $(<_{\text{SEQ}}, f_e)$ means construction of a program $(<_{\text{PAR}}, f_e^{\text{EXP}})$, where $<_{\text{PAR}}$ is a *parallel execution order*: a partial order and a sub-order of $<_{\text{SEQ}}$. Building a new storage mapping f_e^{EXP} from f_e is called *memory expansion*.[3] Obviously, $<_{\text{PAR}}$ and f_e^{EXP} must satisfy several properties in order to preserve the sequential program semantics.

Some additional properties that are not mandatory for the expansion correctness, are guaranteed by most practical expansion techniques. For example, the property that they effectively "expand" data structures. Intuitively, a storage mapping f_e^{EXP} is *finer* than f_e when it uses at least as much memory as f_e. More precisely:

Definition 2.10 (finer) For a given execution e of a program, a storage mapping f_e^{EXP} is finer than f_e if

$$\forall v, w \in \mathbf{W}: \quad f_e^{\text{EXP}}(v) = f_e^{\text{EXP}}(w) \implies f_e(v) = f_e(w).$$

2.5.1 Memory Expansion and Parallelism Extraction

Some basic expansion techniques techniques to build a storage mapping f_e^{EXP} have been listed in Section 1.2, they are used implicitly or explicitly in most memory expansion algorithms, such as the ones presented in Chapter 5.

Now, the benefit of memory expansion is to remove spurious dependences due to memory reuse: "the more expansion, the less memory reuse". Then, removing dependences extracts more parallelism: "the less memory reuse, the more parallelism". Indeed, consider the exact dependence relation δ_e^{EXP} for the same execution of the *expanded* program with *sequential execution order* $(<_{\text{SEQ}}, f_e^{\text{EXP}})$:

$$\forall e \in \mathbf{E}, \forall a, a' \in \mathbf{A}_e:$$
$$a' \, \delta_e^{\text{EXP}} \, a \overset{\text{def}}{\iff} (a \in \mathbf{W}_e \vee a' \in \mathbf{W}_e) \wedge a' <_{\text{SEQ}} a \wedge f_e^{\text{EXP}}(a) = f_e^{\text{EXP}}(a'). \quad (2.12)$$

Any parallel order $<_{\text{PAR}}$ (over instances) must be consistent with dependence relation δ_e^{EXP} (over accesses):

$$\forall e \in \mathbf{E}, \forall (\imath_1, r_1), (\imath_2, r_2) \in \mathbf{A}_e: \quad (\imath_1, r_1) \, \delta_e^{\text{EXP}} \, (\imath_2, r_2) \implies \imath_1 <_{\text{PAR}} \imath_2$$

($\imath_1$, $\imath_2$ are instances and r_1, r_2 are references in a statement).

Of course, we want a compile-time description and consider a conservative approximation δ^{EXP} of δ_e^{EXP}. This approximation does not require any specific analysis in general: its computation is induced by the expansion strategy, see Section 5.4.8 for example.

Theorem 2.2 (correctness criterion of parallel execution orders) Given the following condition, the parallel order is correct for the expanded program (it preserves the original program semantics).

$$\forall (\imath_1, r_1), (\imath_2, r_2) \in \mathbf{A}: \quad (\imath_1, r_1) \, \delta^{\text{EXP}} \, (\imath_2, r_2) \implies \imath_1 <_{\text{PAR}} \imath_2. \quad (2.13)$$

An important remark is that δ_e^{EXP} is actually equal to σ_e when the program is converted to single-assignment form (but not SSA): *every* dependence due to memory reuse is removed. We may thus consider $\delta^{\text{EXP}} = \sigma$ to parallelize such codes.

[3] Because most of the time, f_e^{EXP} requires more memory than f_e.

2.5.2 Computation of a Parallel Execution Order

In this section, we recall some classical results about *loop nest* parallelization; recursive programs will be addressed in Section 5.5. We have already presented—in Section 1.2—two main paradigms to generate parallel code. To compute the parallel execution order $<_{\text{PAR}}$, *data parallelism*—the second paradigm—will be assumed.

Extending parallelization techniques to irregular loop nests has already been studied by several authors: [Col95a, Col94b, GC95] to cite only the results nearest to our work. Instead of presenting a novel algorithm for parallelization, we show how most of the existing ones can be integrated in our framework.

Scheduling

Dependence or reaching definition analyses derive a graph where nodes are operations and edges are constraints on the execution order. The problem is now to traverse the graph in a partial order; this order is the execution order for the parallel program. The more partial the order, the higher the parallelism. In general, this partial order cannot be expressed as the list of relation pairs: one needs an expression of the partial order that does not grow with problem size, i.e. a closed form. Additional constraints on the expression of partial orders are: have a high expressive power; be easily found and manipulated; allow optimized code generation.

A suitable solution is to use a *schedule* [Fea92], i.e. a function θ from the set $\mathbf{I}$ of all instances to the set $\mathbb{N}$ of positive integers. In a more general presentation of schedules, vectors of integers can be used: one may then talk about multidimensional "time" and schedules. This issue is studied by Feautrier in [Fea92]. The following definitions consider one-dimensional schedules only, but it makes no fundamental difference with multidimensional ones. From Theorem 2.2, we already know how the correct parallel execution orders are defined from the dependence relation in the expanded program. Rewriting this result for a schedule function, the correctness becomes

$$\forall (\imath_1, r_1), (\imath_2, r_2) \in \mathbf{A} : \quad (\imath_1, r_1)\, \delta^{\text{EXP}}\, (\imath_2, r_2) \implies \theta(\imath_1) < \theta(\imath_2), \tag{2.14}$$

where δ^{EXP} is the dependence relation in the expanded program. (for multidimensional schedules, $<_{\text{LEX}}$ is used to compare vectors). If no expansion has been performed δ^{EXP} is the original dependence relation δ. If the program has been converted to single assignment form, it is the reaching definition relation σ. On the other hand, since θ is integer valued, the constraint above is equivalent to:

$$\forall (\imath_1, r_1), (\imath_2, r_2) \in \mathbf{A} : \quad (\imath_1, r_1)\, \delta^{\text{EXP}}\, (\imath_2, r_2) \implies \theta(\imath_1) + 1 \le \theta(\imath_2). \tag{2.15}$$

This system of functional inequalities, called *causality constraints*, must be solved for the unknown function θ. As it is often true for system of inequalities, it may have many different solutions. One can minimize various objective functions, as e.g. the number of synchronization points or the latency.

Feautrier's Scheduling Algorithm

In the following, notation $\textsc{Iter}(\imath)$ denotes the *iteration vector* of instance $\imath$. Considering (2.15), let us introduce ξ, the vector of all variables in the problem: ξ is obtained by concatenating $\textsc{Iter}(\imath_1)$, $\textsc{Iter}(\imath_2)$, and the vector of symbolic constants in the problem (recall $\text{ITER}(\langle S, x\rangle) = x$). It so happens that, in the context of affine dependence relations, $((\imath_1, r_1)\, \delta^{\text{EXP}}\, (\imath_2, r_2))$ is the disjunction of conjunctions of affine inequalities. In other words, the set $\{(u, v) : u\, \delta^{\text{EXP}}\, v\}$ is a union of convex polyhedra. This result, built for general affine relations, is also true when the dependence relation is approximated in various ways such as dependence cones, direction vectors and dependence levels, see [PD96, Ban92, DV97].

Since the constraints in the antecedent of (2.15) are affine; let us denote them by $C_i(\xi) \geq 0$, $1 \leq i \leq M$. Similarly, let $\psi(\xi) \geq 0$ be the consequent $\theta(v) - \theta(u) - 1 \geq 0$ in (2.15). Then, we can apply the following lemma:

Lemma 2.2 (Affine Form of Farkas' Lemma) An affine function $\psi(\xi)$ from integer vectors to integers is non-negative on a polyhedron $\{\xi : C_i(\xi) \geq 0, 1 \leq i \leq M\}$ if there exists non-negative integers $\lambda_0, \ldots, \lambda_M$ (the Farkas multipliers) such that:

$$\psi(\xi) = \lambda_0 + \sum_{i=1}^{M} \lambda_i \mathbf{C}_i(\xi) \tag{2.16}$$

This relation is valid for all values of ξ. Hence, one can equate the constant term and the coefficient of each variable in each side of the identity, to get a set of linear equations where the unknowns are the coefficients of the schedules and the Farkas multipliers, λ_i. Since the latter are constrained to be positive, the system must be solved by linear programming [Fea88b, Pug92] (see also Section 3.1).

Unfortunately, some loop nests do not have "simple" affine schedules. The reason is that when a loop nest has an affine schedule, it has a large degree of parallelism. However, it is clear that some loop nests have few or even no parallelism, hence no affine schedule. The solution in this case is to use a multidimensional affine schedule, whose domain is $\mathbb{N}^d$, $d > 1$, ordered according to the lexicographic order. Such a schedule can have as low a degree of parallelism as necessary, and can even represent sequential programs. The selection of a multidimensional schedule can be automated by using algorithms from [Fea92]. It can be proved that any loop nest in an imperative program has a multidimensional schedule. Notice that multidimensional schedules are particularly useful in the case of dynamic control programs, since we have in that case to overestimate the dependences and hence to underestimate the degree of parallelism.

Code generation of parallel scheduled programs is simple in theory, but very complex in practice: issues such as polyhedron-scanning [AI91], communication handling, task placement, and low-level optimizations are critical for efficient code generation [PD96] (pages 79–103). Dealing with complex loop bounds and conditionals raises new code generation problems–not talking about allocation of expanded data structures—see [GC95, Col94a, Col95b].

Other Scheduling Techniques

Before the general solution to the scheduling problem proposed by Feautrier, most algorithms were based on classical loop transformation techniques that include loop fission, loop fusion, loop interchange, loop reversal, loop skewing, loop scaling, loop reindexing and statement reordering. Moreover, dependences abstractions were much less expressive than affine relations.

The first algorithm was designed by Allen and Kennedy [AK87], which inspired many other solutions [Ban92]. Several complexity and optimality results have also been discovered by Darte and Vivien [DV97]. Extending previous results, they designed a very powerful algorithm, but its abstraction does not support the full expressive power of affine relations.

Moreover, many optimizations of Feautrier's algorithm have been designed, mainly because of the wide range of objective functions to optimize. For example, Lim and Lam propose in [LL97] a technique to reduce the number of synchronizations induced by a schedule, and they compare their technique with other recent improvements.

Speculative execution is a classical technique to improve scheduling of finite dependence graphs, but it is not for general affine relations. It has been explored by Collard and Feautrier as a way to extract more parallelism from programs with complex loop bounds and conditionals [Col95a, Col94b].

Eventually, all schedule functions computed by these techniques can be captured by affine functions of iteration vectors. The associated parallel execution order is thus an affine relation $<_{\text{PAR}}$, well suited to our formal framework:

$$\forall u, v \in \mathbf{W}: \quad u <_{\text{PAR}} v \iff \theta(u) < \theta(v)$$

for one-dimensional schedules, and

$$\forall u, v \in \mathbf{W}: \quad u <_{\text{PAR}} v \iff \theta(u) <_{\text{LEX}} \theta(v)$$

for multidimensional ones.

Tiling

Despite the good theoretical results and recent achievements, scheduling techniques can lead to very bad performance, mainly because of communication overhead and cache problems. Indeed, fine grain parallelization is not suitable to most parallel architectures.[4] Partitioning run-time instances is thus an important issue: the solution is to group elementary computations in order to take advantage of memory hierarchies and to overlap communications and computations.

The *tiling* technique groups elementary computations into a *tile*, each tile being executed on a processor in an atomic way. It is well suited to nested loops with regular computation patterns [IT88, CFH95, BDRR94]. An important goal of these researches is to find the best tiling strategy respecting measure criteria like the number of communications happening between the tiles. This strategy must be known at compile time to generate efficient code for a particular machine.

Most tiling techniques are limited to perfect loop nests, and dependences are often supposed uniform when evaluating the amount of communications. The most usual tile model has been defined by Irigoin and Triolet in [IT88]; it enforces the following constraints:

- tiles are bounded for local memory requirements;
- tiles are identical by translation to allow efficient code generation and automatic processing;
- tiles are atomic units of computation with synchronization steps at their beginning and at their end.

[4] But it is suitable for instruction-level parallelism.

Many different algorithms have been designed to find an efficient tile shape and then to partition the nest of loops. Scheduling of individual tiles is done using classical scheduling algorithms. However, inner-tile sequential execution is open for a larger scope of techniques, depending on the context. The simplest inner-tile execution order is the original sequential execution of elementary computations, but other execution orders—still compatible with the program dependences—could be more suitable for the local memory hierarchy, or would enable more aggressive storage mapping optimization techniques (see Section 5.3 for details, but further study of this idea is left for future work). A more extensive presentation of tiling can be found in [BDRR94].

We make one hypothesis to handle parallel execution orders produced by tiling techniques in out framework: the *inner-tile execution order* must be *affine*. It is denoted by $<_{\text{INN}}$. Nevertheless, we are not aware of techniques that would not build affine inner-tile execution orders. The tile shape can be any bounded parallelepiped (or part of a parallelepiped on iteration space boundaries), but is often a rectangle in practice. Then, the result of a tiling technique is a pair (T, θ), where the *tiling function* T maps *statement instances* to individual tiles and the *schedule* θ maps tiles to integers or vectors of integers.

Eventually, the result of a tiling technique can be captured by our parallel execution order framework, with an affine relation $<_{\text{PAR}}$:

$$\forall u, v \in \mathbf{W} : \quad u <_{\text{PAR}} v \iff \theta(T(u)) < \theta(T(v)) \vee (T(u) = T(v) \wedge u <_{\text{INN}} v) \tag{2.17}$$

for a one-dimensional schedule of tiles, and

$$\forall u, v \in \mathbf{W} : \quad u <_{\text{PAR}} v \iff \theta(T(u)) <_{\text{LEX}} \theta(T(v)) \vee (T(u) = T(v) \wedge u <_{\text{INN}} v) \tag{2.18}$$

for a multidimensional schedule.

2.5.3 General Efficiency Remarks

When dealing with nest of loops, it is well known that complex loop transformations require complex polytope traversals, which slightly increases execution time. Moreover, even when no run-time restoration of the data flow is required, the right-hand side of statements often grow huge because of nested conditional expressions. Then, the code generated by a straightforward application of parallelization algorithms is very inefficient. Moving conditionals and splitting loops is very useful, as well as polytope scanning techniques [AI91, FB98].

These remarks naturally extend to recursive programs and recursive data structures. The only difference is that most optimization techniques—such as constant propagation, forward substitution, invariant code motion, dead-code elimination [ASU86, Muc97]—are either limited to non-recursive programs or much less effective with complex recursive structures. In this work, indeed, most experimentations with recursive programs have required manual optimizations. This should encourage us to develop more aggressive techniques suitable for recursive programs.

Of course, *shape* and *alias* analyses discussed in Section 2.2.2 are very useful when pointer-based data structures are considered. A single pair of aliased pointers is likely to forbid any further precise analysis or aggressive program transformation, especially when using generic types (such as `void*`).

Induction variable detection [Wol92] and other related symbolic analysis techniques [HP96] are critical for program analysis and transformation. It is especially true for instancewise analyses: computing the value of an integer (or pointer) variable at each instance of a statement is the key information for dependence analysis. We will indeed present a new induction variable detection technique suitable for our recursive program model.

In the following, when no specific contribution has been proposed in this work, we will not address these necessary previous stages and optimizations:

- we will always consider that the required information about data structure shape, aliases or induction variables is available, when this information can be derived by classical techniques;
- we will generate unoptimized transformed programs, supposing that classical optimization techniques can do the job.

We make the hypothesis that our techniques, if implemented in a parallelizing compiler, are preceded and followed by the appropriate analyses and optimizations.

Chapter 3

Formal Tools

Most technical results on mathematical abstractions are gathered in this chapter. Section 3.1 is a general presentation of Presburger arithmetics and algorithms for systems of affine inequalities. Section 3.2 recalls classical results on formal languages and Section 3.3 addresses rational relations over monoids. Contributions to an interesting class of rational relations are found in Section 3.4. Section 3.5 addresses algebraic relations, and also presents some new results. The two last sections are mostly devoted to applicability of formal language theory to our analysis and transformation framework: Section 3.6 discusses intersection of rational and algebraic relations, and approximation of relations is the purpose of Section 3.7.

The reader whose primary interest is in the analysis and transformation techniques may skip all proofs and technical lemmas, to concentrate on the main theorems. Because this chapter is more a "reference manual" for mathematical objects, it can also been read "on demand" when technical information is required in the following chapters.

3.1 Presburger Arithmetics

When dealing with iteration vectors, we need a mathematical abstraction to capture sets, relations and functions. This abstraction must also support classical algebraic operations. Presburger arithmetics is well suited to this purpose, since most interesting questions are decidable within this theory. It is defined by logical formulas build from $\neg$, $\vee$ and $\wedge$, equality and inequality of integer affine constraints, and first order quantifiers $\exists$ and $\forall$. Testing the satisfiability of a Presburger formula is at the core of most symbolic computations involving affine constraints. It is known as *integer linear programming* and is decidable, but NP-complete, see [Sch86] for details. Indeed, all known algorithms are super-exponential in the worst case, such as the Fourier-Motzkin algorithm implemented by Pugh in Omega [Pug92] and the Simplex algorithm with Gomory cuts implemented by Feautrier in PIP [Fea88b, Fea91]. In practice, Fourier-Motzkin is very efficient on small problems, and the Simplex algorithm is more efficient on medium problems, because its complexity is polynomial in the mean. Computing exact solutions to large integer linear programs is an open problem at present, and this is a problem for practical application of Presburger arithmetics to automatic parallelization.

3.1.1 Sets, Relations and Functions

We consider vectors of integers, and sets, functions, and relations thereof. Functions are seen as a special case of relation and relations are also interpreted as functions: a relation on sets A and B can equivalently be described by a function from A to the set $\mathfrak{P}(B)$ of subsets of B. Notice the range and domain of a function or relation may not have the same dimension. Sets of integer vectors are ordered by the lexicographic order $<_{\text{LEX}}$, and the "bottom element" $\perp$ denotes by definition an element which precedes all integer vectors. Strictly speaking, we consider sets, functions and relations described by Presburger formulas on integer vectors extended with $\perp$.

To describe mathematical objects in Presburger arithmetics, we use three types of variables: *bound*, *unknowns* and *parameters*. Bound variables are quantified by $\exists$ and $\forall$ in logical formulas, whereas unknown variables and parameters are *free* variables. Unknown variables appear in input, output or set tuples, whereas parameters are fully unbound and interpreted as *symbolic constants*. Handling parameters is trivial with Fourier-Motzkin, but required a specific extension of the Simplex algorithm, called Parametric Integer Programming (PIP) by Feautrier [Fea88b].

Omega [Pug92] is widely used in our prototype implementations and semi-automatic experiments, and its syntax is very close to the usual mathematical one. Non-intuitive details will be explained when needed in the experimental sections. PIP uses another representation for affine relations called quasi-affine selection tree or ***quast***, where *quasi-affine forms* are an extension of affine forms including integer division and modulo operations with integer constants.

Definition 3.1 (quast) A quasi-affine selection tree (quast) representing an affine relation[1] is a many level conditional, in which

- predicates are tests for the positiveness of quasi-affine forms in the input variables and parameters,
- and leaves are sets of vectors described in Presburger arithmetics extended with $\perp$ — which precedes any other vector for the lexicographic order.

It should be noticed that bound variables in affine relations appear as parameters in quasts called *wildcard variables*. These wildcard variables are not free: they are constrained inside the quast itself. Moreover, quasi-affine forms (with modulo and division operations) in conditionals and leaves can be converted into "pure" affine forms thanks to additional wildcard variables, see [Fea91] for details.

Empty sets are allowed in leaves—they differ from the singleton $\{\perp\}$—to describe vectors that are not in the domain of a relation. Let us give a few examples.

- The function corresponding to integer addition is written

$$\{(i_1, i_2) \to (j) : i_1 + i_2 = j\}$$

and can be represented by the quast

$$\{i_1 + i_2\}$$

[1] In fact, this is an extension of Feautrier's definition to capture unrestricted affine relations and not only affine functions, see [GC95].

- The same function restricted to integers less than a symbolic constant N is written

$$\{(i_1, i_2) \to (j) : i_1 < N \wedge i_2 < N \wedge i_1 + i_2 = j\}$$

  and as a quast

$$\left| \begin{array}{ll} \textbf{\textit{if}}\ i_1 < N & \\ \textbf{\textit{then}} & \left| \begin{array}{l} \textbf{\textit{if}}\ i_2 < N \\ \textbf{\textit{then}}\ \{i_1 + i_2\} \\ \textbf{\textit{else}}\ \varnothing \end{array} \right. \\ \textbf{\textit{else}}\ \varnothing & \end{array} \right.$$

- The relation between even numbers is written

$$\{(i) \to (j) : (\exists \alpha, \beta : i = 2\alpha \wedge j = 2\beta)\}$$

  (we keep the functional notation $\to$ for better understanding, and to be compliant with Omega's syntax) and a quast representation

$$\left| \begin{array}{l} \textbf{\textit{if}}\ i = 2\alpha \\ \textbf{\textit{then}}\ \{2\beta : \beta \in \mathbb{Z}\} \\ \textbf{\textit{else}}\ \varnothing \end{array} \right.$$

  (α is a wildcard variable)

Many other examples of quasts occur in Chapter 5.

A new interface to PIP has been written in Objective Caml, allowing easy and efficient handling of these quasts. Implementation was done by Boulet and Barthou, see [Bar98] for details. The quast representation is neither better nor worse than the classical logical one, but it is very useful to code generation algorithms and very near from the parametric integer programming algorithm.

To conclude this presentation of mathematical abstractions for affine relations, we suppose that MAKE-QUAST is an algorithm to compute a quast representation for any affine relation. (The reverse problem is much easier and not useful to our framework.) Its extensive description is rather technical but we may sketch the principles of the algorithm. The Presburger formula defining the affine relation is first converted to a form with only existential quantifiers, by the way of negation operators (a technique also used in the Skolem transformation of first order formulas); then every bound variable is replaced by a new wildcard variable; unknown variables are isolated from equalities and inequalities to build sets of integer vectors; and eventually the $\wedge$ and $\vee$ operators are rewritten in terms of conditional expressions. Subsequent simplifications, size reductions and canonical form computations are not discussed here, see [Fea88b, PD96, Bar98] for details.

For more details on Presburger arithmetics, integer programming, mathematical representations of affine relations, specific algorithms and applications to compiler technology, see [Sch86, PD96, Pug92, Fea88b].

3.1.2 Transitive Closure

Computing the transitive closure of a relation is a classical technique in computer science, but most algorithms target relations whose graph is *finite*. This hypothesis is obviously not acceptable in the case of affine relations. The problem is that the transitive closure of an affine relation may not be an affine relation; and knowing when it is an affine relation is not even decidable. Indeed, we can encode the multiplication using transitive closure, which is not definable inside Presburger arithmetics:

$$\{(x, y) \to (x + 1, y + z)\}^* = \{(x, y) \to (x', y + z(x' - x)) : x \le x'\}.$$

It should be noted that testing if a relation R is closed by transitivity is very simple: it is equivalent to $R \circ R - R$ being empty.

We are thus left with approximation techniques. Indeed, finding a lower bound is rather easy in theory: the transitive closure R^* of a relation R can be defined as

$$R^* = \bigcup_{k \in \mathbb{N}} R^k,$$

and computing $\bigcup_{k=0}^{n} R^k$ for increasing values of n yields increasingly accurate lower bounds. In some cases, $\bigcup_{k=0}^{n} R^k$ is constant for n greater than some value n_0, and this constant gives the exact result for R^*. But in general, the size of the result grows very quickly without reaching the exact transitive closure. This method can still be used with "reasonable" values of n to compute a lower bound.

Now, the previous iterative technique is unable to find the exact transitive closure of relation $R = \{(i) \to (i+1)\}$, and it is even unable to give any interesting approximation. The transitive closure of R is nevertheless a very simple affine relation: $R^* = \{(i) \to (i') : i \le i'\}$. More clever techniques should thus be used to approximate transitive closures of affine relations. Kelly et al. designed such a method and implemented it in Omega [KPRS96]. It is based on approximating general affine relations in a sub-class where transitive closure can be computed exactly. They coined the term d-form (d for difference) to define this class. Their technique allows computation of both upper bounds—i.e. conservative approximations—and lower bounds, see [KPRS96] for details.

3.2 Monoids and Formal Languages

This section starts with a short review of basic concepts, then we recall formal languages properties interesting to our purpose. See the well known book by Hopcroft and Ullman [HU79], the first two chapters of the book by Berstel [Ber79], and the Handbook of Formal Languages (volume 1) [RS97a] for details.

3.2.1 Monoids and Morphisms

A ***semi-group*** consists of a set M and an *associative* binary operation on M, usually denoted by multiplication. A semi-group which has a neutral element is a ***monoid***. The neutral element of a monoid is unique, and is usually denoted by 1_M or 1 for short. The monoid structure is widely used in this work, with several different binary operations.

Given two subsets A and B of a monoid M, the product of A and B is defined by

$$AB = \{c \in M : (\exists a \in A, \exists b \in B : c = ab)\}.$$

This definition converts $\mathfrak{P}(M)$ into a monoid with unit $\{1_M\}$. A subset A of M is a sub-semi-group (resp. sub-monoid) of M if $A^2 \subset A$ (resp. $A^2 \subset A$ and $1_M \in A$). Given

any subset A of M, the set
$$A^+ = \bigcup_{n \geq 1} A^n$$
is a sub-semi-group of M, and
$$A^* = \bigcup_{n \geq 0} A^n$$
with $A^0 = \{1_M\}$ is a sub-monoid of M. In fact, A^+ (resp. A^*) is the least sub-semi-group (resp. sub-monoid) for the order of set inclusion containing A. It is called the sub-semi-group (resp. sub-monoid) generated by A. If $M = A^*$ for some $A \subset M$, then A is a system of generators of M. A monoid is *finitely generated* if it has a finite set of generators.

For any set A, the *free monoid* A^* generated by A is defined by tuples $(a_1, \ldots, a_n)$ of elements of A, with $n \geq 0$, and with tuple concatenation as binary operation. When A is finite and non-empty, it is called an *alphabet*, tuples are called *words*, elements of A are called letters and the neutral element is called the *empty word* and denoted by ε. A *formal language* is a subset of a free monoid A^*, and the length $|u|$ of a word $u \in A^*$ is the number of letters composing u. By definition, the length of the empty word is 0. For a letter a in an alphabet A, the number of occurrences of a in A is denoted by $|u|_a$. We will also use the classical notions of prefixes, suffixes, word reversal, sub-words and word factors. The product of two languages is also called concatenation.

We also recall the definition of a monoid morphism. If M and M' are monoids, a (monoid) morphism $\mu : M \to M'$ is a function satisfying
$$\mu(1_M) = 1_{M'} \quad \text{and} \quad \forall m_1, m_2 \in M : \quad \mu(m_1, m_2) = \mu(m_1)\mu(m_2).$$
If A and B are subsets of M and $\mu : M \to M'$ is a morphism, then
$$\mu(AB) = \mu(A)\mu(B), \quad \mu(A^+) = \mu(A)^+, \text{and} \quad \mu(A^*) = \mu(A)^*.$$

3.2.2 Rational Languages

This sections recalls basic definitions and results, to set notations and allow reference in later chapters.

Given an alphabet A, a (finite-state) automaton $\mathcal{A} = (A^*, Q, I, F, E)$ consists of a finite set Q of states, a set $I \subset Q$ of initial states, a set $F \subset Q$ of final states, and a finite set of transitions (a.k.a. edges) $E \subset Q \times A^* \times Q$.

Free monoid A^* is often removed for comodity, when clear from the context: we write $\mathcal{A} = (Q, I, F, E)$. A transition $(q, x, q') \in E$ is usually written $q \xrightarrow{x} q'$, q is the *departing* state, q' is the *arrival* state, and x is the label of the transition. A transition whose label is ε is called an ε-transition.

A *path* is a word $(p_1, x_1, q_1) \cdots (p_n, x_n, q_n)$ in E^* such as $q_i = p_{i+1}$ for all $i \in \{1, \ldots, n-1\}$, and $x_1 \cdots x_n$ is called the *label* of the path. An *accepting path* goes from an initial state to a final one. An automaton is *trim* when all its states are accessible and may be part of an accepting path.

An automaton is *deterministic* when it has a single initial state, every transition label is a single letter or ε, at most one transition may share the same departing state and label, and a state with departing ε-transition may not have departing labeled transitions.

The language $|\mathcal{A}|$ *realized* by a finite-state automaton $\mathcal{A}$ is defined by $u \in |\mathcal{A}|$ iff u labels an accepting path of $\mathcal{A}$. A *regular language* is a language realized by some finite-state automaton.

Any regular language can be realized by a finite-state automaton without ε-transitions and where all transition labels are single letters. Any regular language can be realized by a deterministic finite-state automaton.

The family of *rational languages* over an alphabet A is equal to the least family of languages over A containing the empty set and singletons, and closed under union, concatenation and the star operation.

The following well known theorem is at the core of formal language theory.

Theorem 3.1 (Kleene) Let A be an alphabet. The family of rational and regular languages over A coincides.

Beyond the closure properties included in the definition, rational languages are closed under the plus operation, intersection, complementation, reversal, morphism and inverse morphism.

Proposition 3.1 The following problems are *decidable* for rational languages: membership in linear time, emptiness, finiteness, emptiness of the complement, finiteness of the complement, inclusion, equality.

3.2.3 Algebraic Languages

We recall a few basic facts about algebraic languages and push-down automata. See [HU79, Ber79] for an extensive introduction.

An *algebraic grammar*—a.k.a. *context-free grammar*—$G = (A, V, P)$ consists of an alphabet A of *terminal letters*, an alphabet V of *variables*—also known as *non-terminals*—distinct from A, and a finite set $P \subset V \times (V \cup A)^*$ of *productions*.

When clear from the context, the alphabet is removed from the grammar definition, and we write $G = (V, P)$. A production $(\xi, \alpha) \in P$ is usually written in the form $\xi \to \alpha$, and if $\xi \to \alpha_1, \alpha_2, \ldots, \xi \to \alpha_n$ are productions of G having the same left-hand side ξ, they are grouped together using notation $\xi \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$.

Let A be an alphabet and let $G = (V, P)$ be an algebraic grammar. We define the *derivation relation* as an extension of the production notation $\to$:

$$f \to g \iff \exists \xi \in V, \exists u, \alpha, v \in (V \cup A)^* : \xi \to \alpha \in P \wedge f = u\xi v \wedge g = u\alpha v.$$

Then, for any $p \in \mathbb{N}$, $\xrightarrow{p}$ is the p^{th} iteration of $\to$, and $\xrightarrow{+}$ and $\xrightarrow{*}$ are defined as usual.

In general, grammars are presented with a distinguished non-terminal S called the *axiom*. This allows to define the language L_G generated by a grammar $G = (V, P)$ by

$$L_G = \{u \in A^* : S \xrightarrow{*} u\}.$$

A language L_G generated by some algebraic grammar G is an *algebraic language*—a.k.a. *context-free language*.

Most expected closure properties hold for algebraic languages, but *not intersection*. Indeed, algebraic languages are closed under union, concatenation, star and plus operations, reversal, morphism, inverse morphism, and intersection *with rational languages*.

Although the most natural definition of algebraic languages comes from the grammar model, we prefer in this work another representation.

Given an alphabet A, a *push-down automaton* $\mathcal{A} = (A^*, \Gamma, \gamma_0, Q, I, F, E)$ consists of a *stack alphabet* Γ, a *non-empty* word γ_0 in Γ^+ called the *initial stack word*, a finite set Q

of states, a set $I \subset Q$ of initial states, a set $F \subset Q$ of final states, and a finite set of transitions (a.k.a. edges) $E \subset Q \times A^* \times \Gamma \times \Gamma^* \times Q$.

Free monoid A^* is often removed for commodity, when clear from the context. A transition $(q, x, g, \gamma, q') \in E$ is usually written $q \overset{x:g \to \gamma}{\longrightarrow} q'$, the finite-state automata vocabulary is inherited, and g is called the *top stack symbol*. An empty stack word is denoted by ε.

A *configuration* of a push-down automaton is a triple (u, q, γ), where u is the word to be read, q is the current state and $\gamma \in \Gamma^*$ is the word composed of symbols in the stack. The *transition* between two configurations $c_1 = (u_1, q_1, \gamma_1)$ and $c_2 = (u_2, q_2, \gamma_2)$ is denoted by relation $\mapsto$ and defined by $c \longmapsto c'$ iff there exist $(a, g, \gamma, \gamma') \in A^* \times \Gamma \times \Gamma^* \times \Gamma^*$ such that

$$u_1 = au_2 \wedge \gamma_1 = \gamma' g \wedge \gamma_2 = \gamma' \gamma \wedge (q_1, a, g, \gamma, q_2) \in E.$$

Then $\overset{p}{\longmapsto}$ with $p \in \mathbb{N}$, $\overset{+}{\longmapsto}$ and $\overset{*}{\longmapsto}$ are defined as usual.

A push-down automaton $\mathcal{A} = (\Gamma, \gamma_0, Q, I, F, E)$ is said to *realize* the language L *by final state*, when $u \in L$ iff there exist $(q_i, q_f, \gamma) \in I \times F \times \Gamma^*$ such that

$$(u, q_i, \gamma_0) \overset{*}{\longmapsto} (\varepsilon, q_f, \gamma).$$

A push-down automaton $\mathcal{A} = (\Gamma, \gamma_0, Q, I, F, E)$ is said to *realize* the language L *by empty stack*, when $u \in L$ iff there exist $(q_i, q_f) \in I \times F$ such that

$$(u, q_i, \gamma_0) \overset{*}{\longmapsto} (\varepsilon, q_f, \varepsilon).$$

Notice that realization by empty stack implies realization by finite state: q_f is still required to be in the set of final states.

Theorem 3.2 The family of languages realized by *final state* or by *empty stack* by push-down automata is the family of algebraic languages.

Unlike finite-state automata, the deterministic property for push-down automata imposes some restrictions on the expressive power and brings an interesting closure property. A push-down automaton is *deterministic* when it has a single initial state, every transition label is a single letter or ε, at most one transition may share the same departing state, label and top stack symbol, and a state with departing ε-transition may not have departing labeled transitions.

It is straightforward that any algebraic language can be realized by a push-down automaton whose transition labels are either ε or a single letter. The family of languages realized by *final state* by deterministic push-down automata is called the family of *deterministic algebraic languages*. It should be noticed that this family is also known as $LR(1)$ (which is equal to $LR(k)$ for $k \geq 1$) in the syntactical analysis framework [ASU86].

Proposition 3.2 The family of languages realized by *empty stack* by deterministic push-down automata is the family of deterministic algebraic languages with *prefix property*.

Recall that a language L has the prefix property when a word uv belonging to L forbids u to belong to L, for all words u and non-empty words v. The interesting closure property is the following:

Proposition 3.3 The family of deterministic algebraic languages is closed under complementation.

However, closure of deterministic algebraic languages under union and intersection are not available. Decidability of deterministic algebraic languages among algebraic ones is unknown, despite the number of tries and related works [RS97a].

Proposition 3.4 The following problems are *decidable* for algebraic languages: membership, emptiness, finiteness.

These additional problems are *decidable* for *deterministic* algebraic languages: membership in linear time, emptiness of the complement, finiteness of the complement.

The following problems are *undecidable* for algebraic languages: being a rational language, emptiness of the complement, finiteness of the complement, inclusion (open problem for deterministic algebraic languages), equality (idem).

We conclude this section with a simple algebraic language example whose properties are frequently observed in our analysis framework [Coh99a]. The Lukasiewicz language Ł over an alphabet $\{a, b\}$ is the language generated by axiom ξ and the grammar with productions

$$\xi \longrightarrow a\xi\xi \mid b.$$

The Lukasiewicz language is apparented to Dyck languages [Ber79] and is the simplest of a family of languages constructed in order to write arithmetic expressions without parentheses (prefix or "polish" notation): the letter a represents a binary operation and b represents the operand. Indeed, the first words of Ł are

$$b, abb, aabbb, ababb, aaabbbb, aababbb, \ldots$$

Proposition 3.5 Let $w \in \{a, b\}^*$. Then $w \in$ Ł iff $|w|_a - |w|_b = -1$ and $|u|_a - |u|_b \geq 0$ for any proper left factor u of w (i.e. $\exists v \in \{a, b\}^+ : w = uv$). Moreover, if $w, w' \in$ Ł, then

$$|ww'|_a - |ww'|_b = |w|_a - |w|_b + |w'|_a - |w'|_b.$$

This implies that Ł has the prefix property, see [Ber79] for details. A graphical representation may help understand intuitively the previous proposition and properties of Ł: drawing the graph of function $u \mapsto |u|_a - |u|_b$ as u ranges over the left factors of $w = aabaabbabbabaaabbb$ yields Figure 3.1.a.

Eventually, Figure 3.1.b shows a push-down automaton which realizes the Lukasiewicz language by *empty stack*. It has a single state, which is both initial and final, a single stack symbol I. The initial stack word is also I, it is denoted as $\rightarrow I$ on the initial state. The push-down automaton in Figure 3.1.c realizes Ł by *final state*. Two states are necessary, as well as two stack symbols Z and I, the initial stack word being Z.

Important remark. In the following, every push-down automaton will implicitly accept words by *final state*.

3.2.4 One-Counter Languages

An interesting sub-class of algebraic languages is called the class of one-counter languages. It is defined through push-down automata. A classical definition is the following: A push-down automaton is a *one-counter automaton* if its stack alphabet contains only one letter.

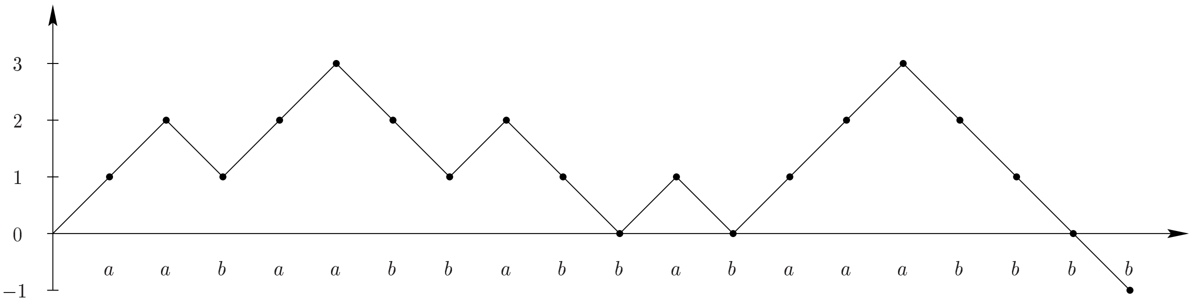


Figure 3.1.a. Evolution of occurrence count differences

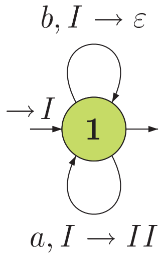


Figure 3.1.b. Push-down automaton accepting by empty stack

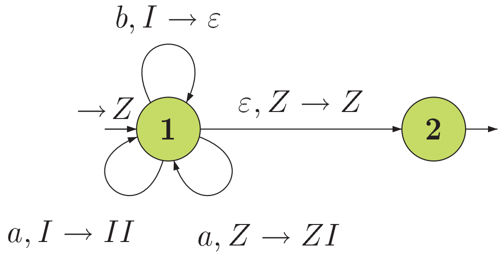


Figure 3.1.c. Push-down automaton accepting by final state

Figure 3.1. Studying the Lukasiewicz language

An algebraic language is a *one-counter language* if it is realized by a one-counter automaton (by final state).

However, we prefer a definition which is more suitable to our practical usage of one-counter languages. This definition is a bit more technical.

Definition 3.2 (one-counter automaton and language) A push-down automaton is a *one-counter automaton* if its stack alphabet contains three letters, Z (for "zero"), I (for "increment") and D (for "decrement") and if the stack word belongs to the (rational) set $ZI^* + ZD^*$. An algebraic language is a *one-counter language* if it is realized by a one-counter automaton (by final state).

It is easy to show that Definition 3.2 describes the same family of languages as the preceding classical definition: the idea is to replace all stack symbols by I and to "remember" the original symbol in the state name. Intuitively, if n is a positive integer, stack word ZI^n stands for counter value n, stack word ZD^n stands for counter value $-n$, and stack word Z stands for counter value 0.

The family of one-counter languages is *strictly* included in the family of algebraic languages, and appears as a natural abstraction in our program analysis framework. The Lukasiewicz language is a simple example of one-counter language, Figure 3.2 shows a one-counter automaton realizing it. This example introduces specific notations to simplify the presentation of one-counter automata:

$\to n$ stands for initialization of the stack word to ZI^n is n is positive, ZD^n if n is negative, and Z if n is equal to zero;

$+n$ for $n \geq 0$ stands for pushing I^n onto the stack if the stack word is in ZI^*, and if the stack word is ZD^k its stands for removing $\max(n,k)$ symbols then, if $n > k$, pushing back I^{n-k} onto the stack;

$+n$ for $n < 0$ stands for $-(-n)$;

$-n$ for $n \geq 0$ stands for pushing D^n onto the stack if the stack word is in ZD^*, and if the stack word is ZI^k its stands for removing $\max(n,k)$ symbols then, if $n > k$, pushing back D^{n-k} onto the stack;

$-n$ for $n < 0$ stands for $+(-n)$;

$=0$ stands for testing if the top stack symbol is Z;

$\neq 0$ stands for testing if the top stack symbol is not Z;

>0 stands for testing if the top stack symbol is I;

<0 stands for testing if the top stack symbol is D;

≥ 0 stands for testing if the top stack symbol is Z or I;

≤ 0 stands for testing if the top stack symbol is Z or D.

These operations are the only available means to check and update the counter. Moreover, tests for 0 can be applied *before* additions or subtractions: $<0\,,-1$ stands for allowing the transition and decrementing the counter when the counter is negative, and $\varepsilon,+1$ stands for incrementing the counter in all cases. See also the transition labeled by b on Figure 3.2.

The general form for a one-counter automaton is thus $(A^*, c_0, Q, I, F, E)]$, where A is an alphabet (removed when clear from the context), c_0 is the initial value of the counter, and $E \subset Q \times A^* \times \{\varepsilon, =0\,, \neq 0\,, >0\,, <0\,, \geq 0\,, \leq 0\} \times \mathbb{Z} \times Q$.

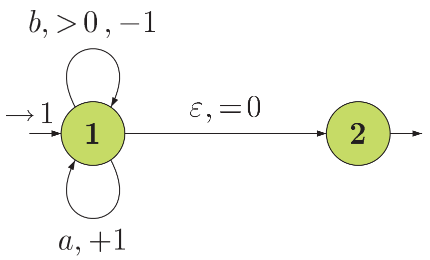


Figure 3.2. One-counter automaton for the Lukasiewicz language

After this short presentation of one-counter languages, one would expect a generalization to multi-counter languages, also called Minsky machines [Min67]. The general form of n-counter automata is $(A^*, c_0^1, \ldots, c^n + 0, Q, I, F, E)$, where c_0^k is the initial value of the k^{th} counter and E is defined on the product of all stacks. However, it has been shown that two-counter automata have the same expressive power as Turing machines—which is a stronger result than the well known equivalence of Turing machines and two-stack automata. Most interesting questions thus become undecidable for multi-counter languages. However, a few additional restrictions on this family of languages have recently been proven to enable several decidability results, as for the emptiness problem. Studying the applicability of these new results to our program analysis framework is left for future work, but most interesting applications would probably arise from work by Comon and Jurski [CJ98].

3.3 Rational Relations

We start with definition and basic properties of recognizable and rational relations, then introduce the machines realizing rational transductions. After studying some examples, we review decision problems and closure properties. This section recalls classical results, see [Eil74, Ber79, AB88] for details.

3.3.1 Recognizable and Rational Relations

We recall the definition and a useful characterization of recognizable sets in finitely generated monoids.

Definition 3.3 (recognizable set) Let M be a monoid. A subset R of M is a *recognizable set* if there exist a finite monoid N, a morphism α from M to N and a subset P of N such that $\alpha(R) = P$.

Recognizable sets can be seen as a generalization of rational (a.k.a. regular) languages to non-free monoïds which preserves the structure of *boolean algebra*:

Proposition 3.6 Let M be a monoid, both $\varnothing$ and M are recognizable sets in M. Recognizable sets are closed under union, intersection and complementation.

Although recognizable sets are closed under concatenation, they are not closed under the *star* operation. But it is the case of *rational* sets, which extend recognizable ones. Their definition is borrowed from rational languages:

Definition 3.4 (rational set) Let M be a monoid. The family of *rational sets* in M is the least family of subsets of M holding $\varnothing$ and singletons $\{m\} \subset M$, closed under union, concatenation and the star operation.

However, rational sets are not closed under complementation and intersection, in general.

When there are two monoids M_1 and M_2 such that $M = M_1 \times M_2$, a recognizable subset of M is called a *recognizable relation*. The following result describes the "structure" of recognizable relations.

Theorem 3.3 (Mezei) A recognizable relation R in $M_1 \times M_2$ is a finite union of sets of the form $K \times L$ where K (resp. L) is a rational set of M_1 (resp. M_2).

When there are two monoids M_1 and M_2 such that $M = M_1 \times M_2$, a rational subset of M is called a *rational relation*. In the following, we will only consider recognizable or rational sets which are *relations* between finitely generated monoids.

The following characterization of rational relations is fundamental: it allows to express rational relations by means of rational languages and monoid morphisms. (The formulation is slightly different from the original theorem by Nivat, see [Ber79] for details.)

Theorem 3.4 (Nivat) Let M and M' be two monoids. Then R is a rational relation over M and M' iff there exist an alphabet A, two morphisms $\mu : A^* \to M$, $\mu' : A^* \to M'$, and a rational language $K \subset A^*$ such that

$$R = \{(\mu(h), \mu'(h)) : h \in K\}.$$

3.3.2 Rational Transductions and Transducers

We recall here a "more functional" view of recognizable and rational relations. From a relation R over M_1 and M_2, we define a *transduction* τ from M_1 into M_2 as a function from M_1 into the set $\mathfrak{P}(M_2)$ of subsets of M_2, such that $v \in \tau(u)$ iff uRv. For commodity, τ may also been extended to a mapping from $\mathfrak{P}(M_1)$ to $\mathfrak{P}(M_2)$, and we write $\tau : M_1 \to M_2$.

A transduction $\tau : M_1 \to M_2$ is recognizable (resp. rational) iff its graph is a recognizable (resp. rational) relation over M_1 and M_2. Both recognizable and rational transductions are closed under inversion (i.e. relational symmetry).

In the next sections, we use either relations or transductions, depending on the context. The family we will study lies somewhere between recognizable and rational relations; it retains the boolean algebra structure and the closure under composition.

The following result—due to Elgot and Mezei [EM65, Ber79]—is restricted to *free* monoids.

Theorem 3.5 (Elgot and Mezei) If A, B and C are alphabets, $\tau_1 : A^* \to B^*$ and $\tau_2 : B^* \to C^*$ are rational transductions, then $\tau_2 \circ \tau_1 : A^* \to C^*$ is a rational transduction.

Nivat's theorem can be rewritten for rational transductions:

Theorem 3.6 (Nivat) Let M and M' be two monoids. Then $\tau : M \to M'$ is a rational transduction iff there exist an alphabet A, two morphisms $\mu : A^* \to M$, $\mu' : A^* \to M'$, and a rational language $K \subset A^*$ such that

$$\forall m \in M : \quad \tau(m) = \mu'(\mu^{-1}(m) \cap K).$$

These two theorems are key results for dependence analysis and dependence testing, see Chapter 4.

The "mechanical" representations of rational relations and transductions are called rational transducers; they extend finite-state automata in a very natural way:

Definition 3.5 (rational transducer) A *rational transducer* $\mathcal{T} = (M_1, M_2, Q, I, F, E)$ consists of an input monoid M_1, an output monoid M_2, a finite set of states Q, a set of initial states $I \subset Q$, a set of final states $F \subset Q$, and a finite set of transitions (a.k.a. edges) $E \subset Q \times M_1 \times M_2 \times Q$.

Monoids M_1 and M_2 are often removed for commodity, when clear from the context: we write $\mathcal{T} = (Q, I, F, E)$. Since we only consider finitely generated monoids, the transitions of a transducer can equivalently be chosen in $Q' \times (G_1 \cup \{1_{M_1}\}) \times (G_2 \cup \{1_{M_2}\}) \times Q'$, where G_1 (resp. G_2) is a set of generators for M_1 (resp. M_2) and Q' is some set of states larger than Q.

Most of the time, we will be dealing with *free* monoids—i.e. languages; the empty word is then the neutral element and is denoted by ε.

A ***path*** is a word $(p_1, x_1, y_1, q_1) \cdots (p_n, x_n, y_n, q_n)$ in E^* such as $q_i = p_{i+1}$ for all $i \in \{1, \ldots, n-1\}$, and $(x_1 \cdots x_n, y_1 \cdots y_n)$ is called the ***label*** of the path. A transducer is ***trim*** when all its states are accessible and may be part of an accepting path.

The transduction $|\mathcal{T}|$ ***realized*** by a rational transducer $\mathcal{T}$ is defined by $g \in |\mathcal{T}|(f)$ iff (f, g) labels an accepting path of $\mathcal{T}$. It is a consequence of Kleene's theorem that a subset of $M_1 \times M_2$ is a rational relation iff it is recognized by a rational transducer :

Proposition 3.7 A transduction is rational iff it is realized by a rational transducer.

Let us now present decidability and undecidability results for rational relations.

Theorem 3.7 The following problems are decidable for rational relations: whether two words are in relation (in linear time), emptiness, finiteness.

However, most other usual questions are undecidable for rational relations.

Theorem 3.8 Let R, R' be rational relations over alphabets A and B with at least two letters. It is undecidable whether $R \cap R' = \varnothing$, $R \subset R'$, $R = R'$, $R = A^* \times B^*$, $(A^* \times B^*) - R$ is finite, R is recognizable.

A few questions may become decidable when replacing A^* and B^* by some particular finitely generated monoids, but it is not the case in general.

The following definition will be useful in some technical discussions and proofs in the following. It formalizes the fact that a rational transducer can be interpreted as a finite-state automaton on a more complex alphabet. But beware: both interpretations have different properties in general.

Definition 3.6 Let $\mathcal{T}$ be a rational transducer over alphabets A and B. The ***finite-state automaton interpretation*** of $\mathcal{T}$ is a finite-state automaton $\mathcal{A}$ over the alphabet $(A \times B) \cup (A \times \{\varepsilon\}) \cup (\{\varepsilon\} \times B)$ defined by the same states, initial states, final states and transitions.

3.3.3 Rational Functions and Sequential Transducers

We need a few results about rational transductions that are partial functions.

Definition 3.7 (rational function) Let M_1 and M_2 be two monoids. A ***rational function*** $\psi : M_1 \to M_2$ is a rational transduction which is a partial function, i.e. such that $\mathrm{Card}(\psi(u)) \leq 1$ for all $u \in M_1$.

Most classical results about rational functions suppose that M_1 and M_2 are *free* monoids, but we will see a result about composition of rational functions over non-free monoids in Section 3.5. In the following, however, M_1 and M_2 will be free monoids.

Given two alphabets A and B, it is decidable whether a rational transduction from A^* into B^* is a partial function. However, the first algorithm by Schützenberger was exponential [Ber79]. The following result by Blattner and Head [BH77] shows that it is decidable in polynomial time.

Theorem 3.9 It is decidable in $\mathcal{O}(\mathrm{Card}(Q)^4)$ whether a rational transducer whose set of states is Q implements a rational function.

Rational functions have two additional decidable properties:

Theorem 3.10 Given two rational functions f and f' from A^* to B^*, it is decidable whether $f \subset f'$ and whether $f = f'$.

Among transducers realizing rational functions, we are especially interested in transducers whose output can be "computed online" with its input. Our interpretation for "online computation" is the following: it requires that when a path e leading to a state q is labeled by pair of words (u, v), and when a letter x is read, there is only one state q' and one output letter y such that (ux, vy) labels a path prefixed by e. This is best understood using the following definitions.

Definition 3.8 (input and output automata) The *input automaton* (resp. *output automaton*) of a transducer is obtained by omitting the output label (resp. input label) of each transition.

Definition 3.9 (sequential transducer) Let A and B be two alphabets. A *sequential transducer* is labeled in $A \times B^*$ and its input automaton is deterministic (which enforces that it has a single initial state).

A sequential transducer obviously realizes a rational function; and a function is *sequential* if it can be realized by a sequential transducer. The transducer example in Figure 3.3.a, whose initial state is **1** is sequential. It replaces by a the bs which appear after an odd number of bs.

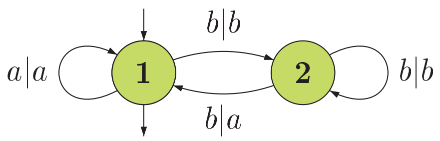


Figure 3.3.a. Sequential transducer

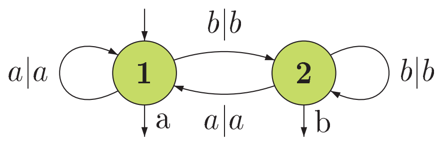


Figure 3.3.b. Sub-sequential transducer

Figure 3.3. Sequential and sub-sequential transducers

Note that a if ψ is a sequential function and $\psi(\varepsilon)$ is defined, then $\psi(\varepsilon) = \varepsilon$. Moreover, when all the states of a sequential transducer are final, the function it realizes is prefix closed, i.e. if uv belongs to its domain then it is the same for u.[2] To a sequential transducer $\mathcal{T} = (A, B^*, Q, I, F, E)$, one may associate a "next state" function $\alpha : Q \times A \to Q$ and a "next output" function $\beta : Q \times A \to B^*$ whose purpose is self-explanatory. Together with the set F of final states, functions α and β are indeed an equivalent characterization of $\mathcal{T}$.

However, the sequential transducer definition is a bit too restrictive regarding our "online computation" property, and we prefer the following extension.

Definition 3.10 (sub-sequential transducer) If A and B are two alphabets, a *sub-sequential transducer* $(\mathcal{T}, \rho)$ over $A^* \times B^*$ is a pair composed of a sequential transducer

[2]In [Ber79, Eil74], all states of a sequential transducer are final.

$\mathcal{T}$ over $A^* \times B^*$ with F as set of final states, and of a function $\rho : F \to B^*$. The function ψ realized by $(\mathcal{T}, \rho)$ is defined as follows: let u be a word in A^*, the value $\psi(u)$ is defined iff there is an accepting path in $\mathcal{T}$ labeled by $(u|v)$ and leading to a final state q; in this case $\psi(u) = v\rho(q)$.

In other words, the function ρ is used to append a word to the output at the end of the computation. A sub-sequential transducer is obviously a rational function; and a function is ***sub-sequential*** if it can be realized by a sequential transducer. A sequential function is sub-sequential: consider $\rho(q) = \varepsilon$ for all final states q.

This definition matches our "online computation" property. The function realized by the sub-sequential transducer in Figure 3.3.b appends to each word its last letter. This function is not sequential because all its states are final and it is not prefix closed.

The following result has been proven by Choffrut in [Cho77].

Theorem 3.11 It is decidable if a function realized by a transducer is sub-sequential, and it is decidable if a sub-sequential function is sequential.

Béal and Carton [BC99b] give two *polynomial-time* algorithms to decide if a rational function is sub-sequential, and if a sub-sequential function is sequential. Two algorithms to build a sub-sequential realization and a sequential realization are also provided, but the first may generate an exponential number of states; as a result, this does not provide a *polynomial-time* algorithm to decide if a rational function is sequential.

Before we conclude this section, notice that the "online computation" property satisfied by sub-sequential transducers is still satisfied for a larger class of rational functions:

Definition 3.11 (online rational transducer) A rational transducer is *online* if it is a rational function and if its input automaton is deterministic. A rational transduction is *online* if it is realized by an online rational transducer.

The only difference with respect to sub-sequential transducers is that ε is allowed in the input automaton, as long as the deterministic property is kept. We are not aware of any result for this class of rational functions, strictly larger than the class of sub-sequential transductions. But if it was decidable among rational functions, it would probably replace every use of sub-sequential functions in the following applications.

In our analysis and transformation framework, we will only use rational and sub-sequential functions, which are decidable in polynomial-time among rational transductions.

3.4 Left-Synchronous Relations

We have seen that rational relations are not closed under intersection, but intersection is critical for dependence analysis. Addressing the undecidable problem of testing whether the intersection of two rational relations is empty or not, Feautrier designed a "semi-algorithm" for dependence testing which sometimes not terminate [Fea98]. Because we would like to effectively compute the intersection, and not only testing its emptiness, our approach is different: we are looking for a sub-class of rational relations with a *boolean algebra* structure (i.e. with union, intersection and complementation).

Indeed, the class of recognizable relations is a boolean algebra, but we have found a more expressive one: the class of *left-syncrhonous relations*. We will show that left-synchronous relations are *not* decidable among rational ones, but we could define a precise algorithm to conservatively approximate relations into left-synchronous ones. In fact, this point is even more interesting for us than decidability. Many results presented here have already been published by Frougny and Sakarovitch in [FS93]. However, our work has been done independently and based on a different—more intuitive and versatile—representation of transductions. Proofs are all new, and several unpublished results have also been discovered.

Notice that a larger class with a boolean algebra structure is the class of *deterministic relations* [PS98] defined by Pelletier and Sakarovitch. But some interesting decidability properties are lost and we could not define any precise approximation algorithm for this class, See Section 3.4.7.

This work has been done in collaboration with Olivier Carton (University of Marne-la-Vallée).

3.4.1 Definitions

We recall the definition of synchronous transducers:[3]

Definition 3.12 (synchronism) A rational transducer on alphabets A and B is *synchronous* if it is labeled on $A \times B$.

A rational relation or transduction is synchronous if it can be realized by a synchronous transducer. A rational transducer is *synchronizable* if it realizes a synchronous relation.

Obviously, such a transducer is length preserving; Eilenberg and Schützenberger [Eil74] showed that the reciprocal is true: a length preserving rational transduction is realized by a synchronous transducer.

A first extension of the synchronous property is the δ-synchronous one:

Definition 3.13 (δ-synchronism) A rational transducer on alphabets A and B is *δ-synchronous* if every transition appearing in a cycle of the transducer's graph is labeled on $A \times B$.

A rational relation or transduction is δ-synchronous if it can be realized by a synchronous transducer. A rational transducer is *δ-synchronizable* if it realizes a δ-synchronous relation.

Such a transducer has a bounded length difference; Frougny and Sakarovitch [FS93] showed that the reciprocal is true: a bounded length difference rational transduction is realized by a δ-synchronous transducer. Obviously, the bound is 0 when the transducer is synchronous. Two examples are shown in Figure 3.4. They respectively realize $\{(u,v) \in \{a,b\}^* \times \{a,b\}^* : u = v\}$ and $\{(u,v) \in \{a,b\}^* \times \{c\}^* : |u|_a = |v|_c \wedge |u|_b = 2\}$.

Then, we define two new extensions:

Definition 3.14 (left-synchronism) A rational transducer over alphabets A and B is *left-synchronous* if it is labeled on $(A \times B) \cup (A \times \{\varepsilon\}) \cup (\{\varepsilon\} \times B)$ and only transitions labeled on $A \times \{\varepsilon\}$ (resp. $\{\varepsilon\} \times B$) may *follow* transitions labeled on $A \times \{\varepsilon\}$ (resp. $\{\varepsilon\} \times B$).

A rational relation or transduction is left-synchronous if it is realized by a left-synchronous transducer. A rational transducer is *left-synchronizable* if it realizes a left-synchronous relation.

[3]It appears to be a special case of k,l-synchronous transducers, where $k = l = 1$, see Section 3.4.7.

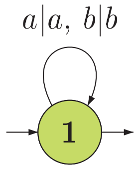


Figure 3.4.a. A synchronous transducer

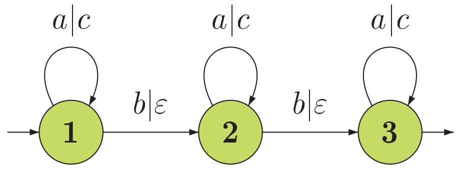


Figure 3.4.b. A δ-synchronous transducer

Figure 3.4. Synchronous and δ-synchronous transducers

Definition 3.15 (right-synchronism) A rational transducer over alphabets A and B is *right-synchronous* if it is labeled on $(A\times B)\cup(A\times\{\varepsilon\})\cup(\{\varepsilon\}\times B)$ and only transitions labeled on $A \times \{\varepsilon\}$ (resp. $\{\varepsilon\} \times B$) may *precede* transitions labeled on $A \times \{\varepsilon\}$ (resp. $\{\varepsilon\} \times B$).

A rational relation or transduction is right-synchronous if it can be realized by a right-synchronous transducer. A rational transducer is *right-synchronizable* if it realizes a right-synchronous relation.

Figure 3.5 shows left-synchronous transducers over an alphabet A realizing two orders (a.k.a. orderings), where $<_{\text{TXT}}$ is some order on A: the prefix order $f <_{\text{PRE}} g \Leftrightarrow \{\exists h \in A^* : f = gh\}$ and the lexicographic order $f <_{\text{LEX}} g \Leftrightarrow \{f <_{\text{PRE}} g \vee (\exists u, v, w \in A^*, a, b \in A : f = uav \wedge g = ubw \wedge a < b)\}$.

In the following transducers, labels x and y stand for $\forall x \in A$ and $\forall y \in A$ respectively.

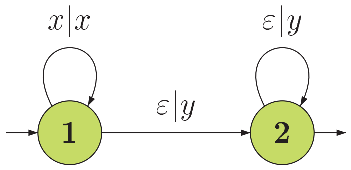


Figure 3.5.a. Prefix order

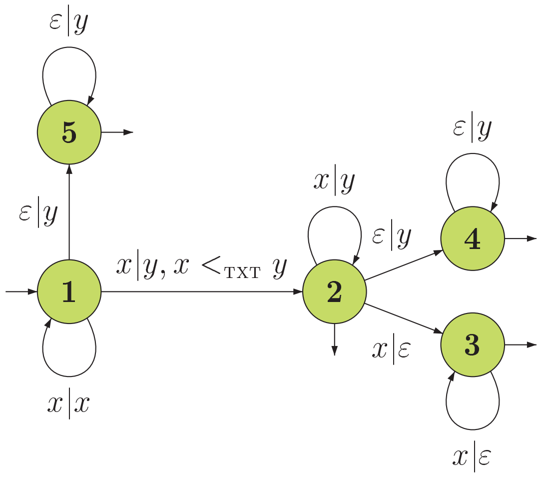


Figure 3.5.b. Lexicographic order

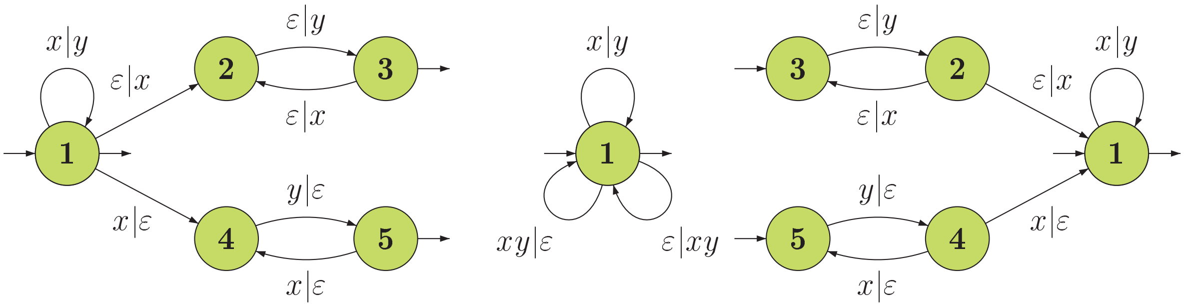
Figure 3.5. Left-synchronous realization of several order relations

The *word-reversal* operation converts a left-synchronous transducer into a right-synchronous one and conversely.[4] The two definitions are not contradictory: some relations are left and right synchronous, such as synchronous ones.

[4] Recognizable, synchronous and δ-synchronous relations are closed under word-reversal.

Figure 3.6 shows a transducer realizing the relation $\tau = \{(u, v) \in A^* \times B^* : |u| \equiv |v| \mod 2\}$. It is neither left-synchronous nor right-synchronous, but the left-synchronous and right-synchronous realizations in the same figure show that τ is left and right synchronous.

In the three following transducers, labels x and y stand for $\forall x \in A$ and $\forall y \in B$.



(left-synchronous) (left and right synchronizable) (right-synchronous)

Figure 3.6. A left and right synchronizable example

In the following we mostly consider left-synchronous transducers, because all results extend to right-synchronous through the word-reversal operation and most interesting transducers are left-synchronous.

3.4.2 Algebraic Properties

It is well known that synchronous and δ-synchronous relations are closed under union, complementation, intersection. We show that it is the same for left-synchronous relations.

Lemma 3.1 (Union) The class of left-synchronous relations is closed under union.

Proof: Let $\mathcal{T} = (Q, I, F, E)$ and $\mathcal{T}' = (Q', I', F', E')$ be left-synchronous transducers. Q and Q' can be supposed disjoint without loss of generality; and then $(Q \cup Q', I \cup I', F \cup F', E \cup E')$ realizes $|\mathcal{T}| \cup |\mathcal{T}'|$. ■

The proof is constructive: given two left-synchronous realizations, one may compute a left-synchronous realization of the union.

Here is a direct application:

Theorem 3.12 Recognizable relations are left-synchronous.

Proof: Let R be a recognizable relation in $A^* \times B^*$. From Theorem 3.3, there exists an integer n, $A_1, \ldots, A_n \in A^*$, and $B_1, \ldots, B_n \in B^*$ such that $tau = A_1 \times B_1 \cup \cdots \cup A_n \times B_n$. Let $i \in \{1, \ldots, n\}$, $\mathcal{A}_A = (Q_A, I_A, F_A, E_A)$ accepting A_i, and $\mathcal{A}_B = (Q_B, I_B, F_B, E_B)$ accepting B_i. We suppose Q_A and Q_B are disjoint sets—without loss of generality—and define a transducer $\mathcal{T} = (Q, I, F, E)$, where $Q = (Q_A \times Q_B) \cup Q_A \cup Q_B$, $I = I_A \times I_B$, $F = F_A \times F_B \cup F_A \cup F_B$, and E is defined as follows:

1. All transitions in E_A and E_B are also in E;

2. If $q_A \xrightarrow{x} q'_A \in E_A$ and $q_B \xrightarrow{y} q'_B \in E_B$, then $(q_A, q_B) \xrightarrow{x|y} (q'_A, q'_B) \in E$;
3. If q_A (resp. q'_B) is a final state and $q_B \xrightarrow{y} q'_B \in E_B$ (resp. $q_A \xrightarrow{x} q'_A \in E_A$), then $(q_A, q_B) \xrightarrow{\varepsilon|y} q'_B \in E$ (resp. $(q_A, q_B) \xrightarrow{x|\varepsilon} q'_A \in E$).

By construction, $\mathcal{T}$ is left-synchronous, its input is A_i and its output is B_i. Moreover, it accepts any combination of input words in A_i and output words in B_i. Lemma 3.1 terminates the proof. ■

The proof is constructive: given a decomposition of a recognizable relation into products of rational languages, one may build a left-synchronous transducer.

Another application is this useful decomposition result for left-synchronous relations:

Proposition 3.8 Any left-synchronous relation can be decomposed into a union of relations of the form SR, where S is synchronous and R has either no input or no output (R is thus recognizable).

Proof: Consider a relation $U \in A^* \times B^*$ realized by a left-synchronous transducer $\mathcal{T}$, and consider an accepting path e in $\mathcal{T}$. The restriction of $\mathcal{T}$ to the states and transitions in e yields a transducer $\mathcal{T}_e$, such as $|\mathcal{T}_e| \subset |\mathcal{T}|$. Morover, $\mathcal{T}_e$ can be divided into transducers $\mathcal{T}_s$ and $\mathcal{T}_r$, such as the (unique) final state of the first is the (unique) initial state of the second, $\mathcal{T}_s$ is synchronous and $\mathcal{T}_r$ has either no input or no output. Therfore, $\mathcal{T}_e$ realizes a left-synchronous relation of the form SR, where S is synchronous and R has either no input or no output. Since the number of "restricted" transducers $\mathcal{T}_e$ is finite, closure under union terminates the proof. ■

The proof is constructive if the left-synchronous relation to be decomposed is given by a left-synchronous realization.

To study complementation and intersection, we need two more definitions: unambiguity and completion.

Definition 3.16 (unambiguity) A rational transducer $\mathcal{T}$ over A and B is *unambiguous* if any couple of words over A and B labels at most one path in $\mathcal{T}$. A rational relation is unambiguous if it is realized by an unambiguous transducer.

This definition coincides with the one in [Ber79] Section IV.4 for *rational functions*, but differs for general rational transductions.

Definition 3.17 (completion) A rational transducer $\mathcal{T}$ is *complete* if every pair of words labels at least one path in $\mathcal{T}$ (accepting or not).

It is obviously not always possible to complete a transducer in a *trim* one. From these two definitions, let us recall a very general result.

Theorem 3.13 The class of a complete unambiguous rational relations is closed under complementation.

Proof: Let R be a complete unambiguous relation realized by transducer $\mathcal{T} = (Q, I, F, E)$. We define a transducer $\mathcal{T}' = (Q, I, Q - F, E)$ such that an accepting path in $\mathcal{T}$ cannot be one of $\mathcal{T}'$. The completion of $\mathcal{T}$ and the uniqueness of accepting paths in $\mathcal{T}$ shows that the complementation of R is realized by $\mathcal{T}'$. ■

The proof is constructive.

Now, we specialize this result for left-synchronous relations.

Lemma 3.2 A left-synchronous relation is realized by an unambiguous left-synchronous transducer.

Proof: Let $\mathcal{T}$ be a left-synchronous transducer over A and B realizing a relation R. Let $\mathcal{A}$ be the finite-state automaton interpretation of $\mathcal{T}$—over the alphabet $(A \times B) \cup (A \times \{\varepsilon\}) \cup (\{\varepsilon\} \times B)$—and let $\mathcal{A}'$ be a deterministic finite-state automaton accepting the same language as $\mathcal{A}$. Let f, g two words such that $|\mathcal{T}|(f) = g$, and let e and e' be two accepting paths in $\mathcal{T}$.

Suppose e differs from e'. By the determinim property, the words w and w' they accept in $\mathcal{A}'$ also differs; let (x, y) and (x', y') be the first difference. If $x = \varepsilon$ and $x' \neq \varepsilon$, the definition of left-synchronous transducers imposes that w to be labeled in $\{\varepsilon\} \times B$ after (x, y), then e and e' accept different inputs in $\mathcal{T}$. The same reasoning applies to the three other cases—$y = \varepsilon$ and $y' \neq \varepsilon$, $x' = \varepsilon$ and $x \neq \varepsilon$, $y' = \varepsilon$ and $y \neq \varepsilon$—and yields different inputs or outputs for paths e and e'. This contradicts the definition of e and e'.

Thus f and g are accepted by a unique path in the rational transducer interpretation $\mathcal{T}'$ of $\mathcal{A}'$. Since $\mathcal{A}'$ is the determinization of $\mathcal{A}$, a transition labeled on $A \times \{\varepsilon\}$ (resp. $\{\varepsilon\} \times B$) may only be followed by another transition labeled on $A \times \{\varepsilon\}$ (resp. $\{\varepsilon\} \times B$). Eventually, $\mathcal{T}'$ is unambiguous and left-synchronous, and it realizes R. ■

The proof is constructive.

Proposition 3.9 A left-synchronous relation is realized by a *complete unambiguous* left-synchronous transducer.

Proof: Let R be a left-synchronous relation. We use Lemma 3.2 to compute an unambiguous left-synchronous transducer $\mathcal{T} = (Q, I, F, E)$ which realizes R. We define a transducer $\mathcal{T}' = (Q', I, F, E')$, where q_i, q_o and q_{io} are three new states, $Q' = Q \cup \{q_i, q_o, q_{io}\}$, and E' is defined as follows:

1. All transitions in E are also in E'.
2. For all $(x, y) \in A \times B$, $q_{io} \xrightarrow{x|y} q_{io} \in E'$.
3. For all $x \in A$, $q_{io} \xrightarrow{x|\varepsilon} q_i \in E'$ and $q_i \xrightarrow{x|\varepsilon} q_i \in E'$.
4. For all $y \in B$, $q_{io} \xrightarrow{\varepsilon|y} q_o \in E'$ and $q_o \xrightarrow{\varepsilon|y} q_o \in E'$.
5. If $q \in Q$ is such that $\forall (x', q') \in A \times Q : q' \xrightarrow{x'|\varepsilon} q \notin E$, then $\forall (y'', q'') \in B \times Q$: $q \xrightarrow{\varepsilon|y''} q'' \notin E \Rightarrow q \xrightarrow{\varepsilon|y''} q_o \in E'$.
6. If $q \in Q$ is such that $\forall (y', q') \in B \times Q : q' \xrightarrow{\varepsilon|y'} q \notin E$, then $\forall (x'', q'') \in A \times Q$: $q \xrightarrow{x''|\varepsilon} q'' \notin E \Rightarrow q \xrightarrow{x''|\varepsilon} q_i \in E'$.
7. If $q \in Q$ is such that $\forall (x', q') \in A \times Q : q' \xrightarrow{x'|\varepsilon} q \notin E$ and $\forall (y', q') \in B \times Q : q' \xrightarrow{\varepsilon|y'} q \notin E$, then $\forall (x'', y'', q'') \in A \times B \times Q :\quad q \xrightarrow{x''|y''} q'' \notin E \Rightarrow q \xrightarrow{x''|y''} q_{io} \in E'$.

The resulting transducer is left-synchronous, complete, and realizes relation R. Moreover, the three last cases have been carefully designed to preserve the unambiguous property: no transition departing from a state q is added if its label is already the one of an existing transition departing from q. ■

The proof is constructive.

Theorem 3.14 (Complementation and Intersection) The class of left-synchronous relations is closed under complementation and intersection.

Proof: As a corollary of Theorem 3.13 and Proposition 3.9, we have the closure under complementation. Together with closure under union, this proves closure under intersection. ■

Eventually, we have proven that the class of left-synchronous relations is a *boolean algebra*, which will be of great help for dependence and reaching definition analysis, see Section 4.3.

Synchronous and δ-synchronous relations are obviously closed under concatenation, but it is not true for left-synchronous ones. However, we have the following result:

Proposition 3.10 Let S, T and R be rational relations.

(i) If S is synchronous and T is left-synchronous, then ST is left-synchronous.

(ii) If T is left-synchronous and R is recognizable, then TR is left-synchronous.

Proof: Proof of (i) is a straightforward application of the definition of left-synchronous transducers (see Proposition 3.12 for a generalization).

We use Proposition 3.8 to partition T into $S_1R_1, \ldots, S_nR_n$ where S_i is synchronous and R_i is recognizable for all $1 \leq i \leq n$. Now, R_iR is recognizable, hence left-synchronizable from Theorem 3.12. Application of (i) shows that S_iR_iR is left-synchronizable. Closure under union terminates the proof of (ii). ■

The proof is constructive when a left-synchronous realization of T is provided, thanks to Proposition 3.8. A generalization of (i) is given in Section 3.4.5.

To close this section about algebraic properties, one should notice that the finite-state automaton interpretation (see Definition 3.6) of a left-synchronous transducer $\mathcal{T}$ has exactly the same properties as $\mathcal{T}$ itself, regarding computation of the complementation and intersection. Indeed, by definition of left-synchronous relations, applying classical algorithms from automata theory to the finite-state automaton interpretation yields correct results on the transducer. This remark shows that algebraic operations for left-synchronous transducers have the same complexity as for finite-state automata in general.

3.4.3 Functional Properties

Synchronous and δ-synchronous transductions are closed under inversion (i.e. relational symmetry) and composition. Clearly, the class of left-synchronous transductions is also closed under inversion.

Combined with the boolean algebra structure, the following result is useful for reaching definition analysis (to solve (4.17) in Section 4.3.3).

Theorem 3.15 The class of left-synchronous transductions is closed under composition.

Proof: Consider three alphabets A, B and C, two transductions $\tau_1 : A^* \to B^*$ and $\tau_1 : B^* \to C^*$, and two left-synchronous transducers $\mathcal{T}_1 = (Q_1, I_1, F_1, E_1)$ realizing τ_1 and $\mathcal{T}_2 = (Q_2, I_2, F_2, E_2)$ realizing τ_2. We suppose Q_1 and Q_2 are disjoint sets—without loss of generality—and define $\mathcal{T} = (Q_1 \times Q_2 \cup Q_1 \cup Q_2, I_1 \times I_2, F_1 \times F_2 \cup F_1 \cup F_2, E)$ as

1. All transitions in E_1 and E_2 are also in E;
2. If $q_1 \xrightarrow{x|y} q_1' \in E_1$ and $q_2 \xrightarrow{y|z} q_2' \in E_2$, then $(q_1, q_2) \xrightarrow{x|z} (q_1', q_2') \in E$;
3. If $q_1 \xrightarrow{x|\varepsilon} q_1' \in E_1$ and $q_2 \xrightarrow{\varepsilon|z} q_2' \in E_2$, then $(q_1, q_2) \xrightarrow{x|z} (q_1', q_2') \in E$;
4. If $q_1 \xrightarrow{\varepsilon|y} q_1' \in E_1$ and $q_2 \xrightarrow{y|\varepsilon} q_2' \in E_2$, then $(q_1, q_2) \xrightarrow{\varepsilon|\varepsilon} (q_1', q_2') \in E$;
5. If $q_1 \xrightarrow{x|y} q_1' \in E_1$ and $q_2 \xrightarrow{y|\varepsilon} q_2' \in E_2$, then $(q_1, q_2) \xrightarrow{x|\varepsilon} (q_1', q_2') \in E$;
6. If $q_1 \xrightarrow{\varepsilon|y} q_1' \in E_1$ and $q_2 \xrightarrow{y|z} q_2' \in E_2$, then $(q_1, q_2) \xrightarrow{\varepsilon|z} (q_1', q_2') \in E$;
7. If $q_1 \xrightarrow{x|\varepsilon} q_1' \in E_1$, then $\forall q_2 \in F_2 : (q_1, q_2) \xrightarrow{x|\varepsilon} q_1' \in E$;
8. If $q_2 \xrightarrow{\varepsilon|z} q_2' \in E_2$, then $\forall q_1 \in F_1 : (q_1, q_2) \xrightarrow{\varepsilon|z} q_2' \in E$.

First, consider an accepting path e in $\mathcal{T}$ for a couple of words (f, h). We may write $e = e_{12}e'$, where e_{12} is the $Q_1 \times Q_2$ part of e. By construction of $\mathcal{T}$, the end state of e_{12} is a final state of $\mathcal{T}_1$ and e' is a path of $\mathcal{T}_2$, or it is the opposite. Considering the projection of states in e_{12} on Q_1, e_{12} accepts a couple of words (f, g) in $\mathcal{T}_1$ such as $h \in \tau_2(g)$. Hence $h \in \tau_2 \circ \tau_1(f)$.

Second, consider three words f, g, h such as $g \in \tau_1(f)$ and $h \in \tau_2(g)$. Let e_1 be an accepting path for (f, g) in $\mathcal{T}_1$ and e_2 be one for (g, h) in $\mathcal{T}_2$. Suppose $|e_1| > |e_2|$. Build a path e_{12} in $\mathcal{T}$ from the product of states and labels of the first $|e_2|$ transitions in e_1 and e_2; its end state is (q_1, q_2) with $q_1 \in Q_1$ and $q_2 \in F_2$. Now, the last $|e_1| - |e_2|$ transitions in e_1 can be written $(q_1, x, \varepsilon, q_1').e_1'$, hence $e_{12}.((q_1, q_2), x, \varepsilon, q_1').e_1'$ is an accepting path for (f, h) in $\mathcal{T}$.

Eventually, we have shown that $\mathcal{T}$ realizes $\tau_2 \circ \tau_1$. Now, using the classical $\varepsilon|\varepsilon$-transition removal algorithm for finite-state automata, we define transducer $\mathcal{T}'$. It is left-synchronous because $\mathcal{T}_1$ and $\mathcal{T}_2$ are, and transitions involving states of Q_1 or Q_2—labeled on $A \times \{\varepsilon\}$ or $\{\varepsilon\} \times C$—are never followed by transitions involving states of $Q_1 \times Q_2$. ■

The proof is constructive.

Before showing an important application of this result, we need an additional definition:

Definition 3.18 (α-selection) Let $\tau : A^* \to B^*$ be a rational transduction, and α be a rational order on B^*—i.e. a rational relation which is reflexive, anti-symmetric and transitive. The ***α-selection*** of τ is a *partial function* τ_α defined by

$$\forall u, v \in A^* \times B^* : \quad v = \tau_\alpha(u) \iff v = \min_\alpha \tau(u).$$

Proposition 3.11 Let $\tau : A^* \to B^*$ be a left-synchronous transduction, and α be a left-synchronous order on B^*. The α-selection of τ is a left-synchronous function.

Proof: Let ι be the identity rational function on B^*. If τ_α is the α-selection of τ, the proof comes from the fact that $\tau_\alpha = \tau - ((\alpha - \iota) \circ \tau)$ ■

The most interesting application of this to our framework appears when choosing the *lexicographic order* for α, see Section 4.3.3. For more details on α-selection, also known as uniformization, see [PS98].

3.4.4 An Undecidability Result

It is well known that the *recognizability* of a transduction is undecidable. This is proved by Berstel in [Ber79] Theorem 8.4, and we use a similar technique to show that it is the same for left-synchronous relations. We start with a preliminary result.

Lemma 3.3 Let K be a positive integer, let $A = \{a, b\}$, let B be any alphabet, and let $u_1, u_2, \ldots, u_p \in B^*$. Define

$$U = \{(ab^K, u_1), (ab^{2K}, u_2), \ldots, (ab^{pK}, u_p)\}.$$

Then, U and U^+ are rational relations, and relation $(A^* \times B^*) - U^+$ is also rational.

Proof: Relation U is finite, hence rational, and U^+ is rational by closure under concatenation and the star operation.

Usually, the class of rational relations is not closed under complementation, so we have to prove something here. This is done the same way as in [Ber79] Lemma 8.3, with the only substitution of b by b^K. ■

Theorem 3.16 Let A and B be alphabets with at least two letters. Given a rational relation R over A and B, it is undecidable whether R is left-synchronous.

Proof: We may assume that A contains exactly two letters, and set $A = \{a, b\}$. Consider two sequences $u_1, u_2, \ldots, u_p$ and $v_1, v_2, \ldots, v_p$ of *non-empty* words over B, and let K be their maximum length. Define

$$U = \{(ab^K, u_1), \ldots, (ab^{pK}, u_p)\} \quad \text{and} \quad V = \{(ab^K, v_1), \ldots, (ab^{pK}, v_p)\}.$$

From Lemma 3.3, U, V, U^+, V^+, $\bar{U} = (A^* \times B^*) - U^+$ and $\bar{V} = (A^* \times B^*) - V^+$ are rational relations.

Let $R = \bar{U} \cup \bar{V}$. Since left-synchronous transductions are closed under complementation, R is left-synchronous iff $(A^* \times B^*) - R = U^+ \cap V^+$ is.

Assume $U^+ \cap V^+$ is non-empty and realized by a left-synchronous transducer $\mathcal{T}$. Consider $(m, u) \in U^+ \cap V^+$. We may write $m = fg$ with $|f| = |u|$ and $|g| > 0$. Left-synchronism requires that (g, ε) labels a path in $\mathcal{T}$. Moreover, $((fg)^k, u^k) \in U^+ \cap V^+$ for all $k \geq 1$, hence the path labeled by (g, ε) must be part of a cycle:

$$\exists g' : \forall k : (fg(g'g)^k, u) \in U^+ \cap V^+.$$

However, because $u_1, \ldots, u_p$ and $v_1, \ldots, v_p$ are non-empty, the ratio between the length of input and output words must be less than or equal to $K + 1$; this is contradictory.

Eventually, R is left-synchronous iff $U^+ \cap V^+$ is empty.[5] Since deciding this emptiness is exactly solving the Post's Correspondence problem for $u_1, \ldots, u_p$ and $v_1, \ldots, v_p$, we have proven that left-synchronism is undecidable. ■

A similar proof shows the following result, which is not a corollary of Theorem 3.16.

Theorem 3.17 Let A and B be alphabets with at least two letters. Given a rational relation R over A and B, it is undecidable whether R is left and right synchronous.

3.4.5 Studying Synchronizability of Transducers

Despite the general undecidability results, we are interested in particular cases where a rational relation can be proved left-synchronous.

Transmission Rate

We recall the following useful notion to give an alternative description of synchronism in transducers. The *transmission rate* of a path labeled by (u, v) is defined as the ratio $|v|/|u| \in \mathbb{Q}^+ \cup \{+\infty\}$.

Eilenberg and Schützenberger [Eil74] showed that the *synchronism* property of a transducer is decidable. Frougny and Sakarovitch [FS93] showed a similar result for *δ-synchronism*, and their algorithm operates directly on the transducer that realizes the transduction. The result is:

Lemma 3.4 A rational transducer is δ-synchronizable iff the transmission rate of all its cycles is 1.

There is no characterization of recognizable transducers through the transmission rate of its cycles, but one may give a sufficient condition:

Lemma 3.5 If the transmission rate of all cycles in a rational transducer is 0 or $+\infty$, then it realizes a recognizable relation.

Proof: Let $\mathcal{T}$ be a rational transducer whose cycles transmission rates are only 0 and $+\infty$. Considering a strongly-connected component, all its cycles must be of the same rate. Hence a strongly-connected component has either no input or no output. This proves that strongly-connected components are recognizable. Closure of recognizable relations by concatenation and by union terminates the proof. ■

There is no characterization of left-synchronizable transducers either. However, as a straightforward application of previous definitions, one may give the following result:

Lemma 3.6 If $\mathcal{T}$ is a left-synchronous transducer, then cycles of $\mathcal{T}$ may only have three different transmission rates: 0, 1 and $+\infty$. All cycles in the same strongly-connected component must have the same transmission rate, only components of rate 0 may follow components of rate 0, and only components of rate $+\infty$ may follow components of rate $+\infty$.

Even if *synchronizable* transducers may not satisfy these properties, some kind of reciprocal is available, see Theorem 3.19.

[5] We have also proven here that U^+ and V^+ are not left-synchronous.

Classes of Transductions

We have shown that left-synchronous transductions extend algebraic properties of recognizable transductions. The following theorem shows that they also extend real-time properties of δ-synchronous transducers.

Theorem 3.18 δ-synchronous transductions are left-synchronous.

Proof: Consider a δ-synchronous transducer $\mathcal{T}$ realizing a relation R over alphabets A and B, and call δ the upper bound on delays between input and output words accepted by $\mathcal{T}$. Taking advantage of closure under intersection, one may partition R into relations R_i of constant delay i, for all $-\delta \leq i \leq \delta$. Let $\mathcal{T}_i$ realize relation R_i: by construction, $v \in |\mathcal{T}_i|(u)$ iff $|u| = |v| + i$.

Let "$\sqcup$" be a new label; if i is non-negative (resp. negative), define $\mathcal{T}_i'$ from $\mathcal{T}_i$ in substituting its final state by a transducer accepting $(\varepsilon, \sqcup^i)$ (resp. $(\sqcup^{-i}, \varepsilon)$). Each $\mathcal{T}_i'$ is length preserving, hence *synchronizable*. Transducer $\mathcal{T}' = \mathcal{T}'_{-\delta} \cup \cdots \cup \mathcal{T}'_\delta$ is thus synchronizable, hence *left-synchronizable*.

Let $\mathcal{P}$ realize relation $\{(u, u\sqcup^a) : u \in A^*, a \geq 0\}$ and $\mathcal{Q}$ realize relation $\{(v\sqcup^b, v) : v \in B^*, b \geq 0\}$, which are both left-synchronizable. Transducer $\mathcal{Q} \circ \mathcal{T}' \circ \mathcal{P}$ realizes the same transduction as $\mathcal{T}$, and it is left-synchronizable from Theorem 3.15. ■

One may go a bit further and give a generalization of Theorems 3.12 and 3.18, based on Lemmas 3.5 and 3.4:

Theorem 3.19 If the transmission rate of each cycle in a rational transducer is 0, 1 or $+\infty$, and if no cycle whose rate is 1 follows a cycle whose rate is not 1, then the transducer is *left-synchronizable*.

Proof: Consider a rational transducer $\mathcal{T}$ satisfying the above hypotheses, and consider an acceptation path e in $\mathcal{T}$. The restriction of $\mathcal{T}$ to the states and transitions in e yields a transducer $\mathcal{T}_e$, such as $|\mathcal{T}_e| \subset |\mathcal{T}|$. Moreover, $\mathcal{T}_e$ can be divided into transducers $\mathcal{T}_s$ and $\mathcal{T}_r$, such as the (unique) final state of the first is the (unique) initial state of the second, and the transmission rate of all cycles is 1 in $\mathcal{T}_s$ and either 0 or $+\infty$ in $\mathcal{T}_r$. From Lemma 3.5, $\mathcal{T}_r$ is recognizable. From Lemma 3.4, $\mathcal{T}_s$ is δ-synchronizable, hence left-synchronizable from Theorem 3.18. Eventually, Proposition 3.10 shows that $\mathcal{T}_e$ is left-synchronizable. Since the number of "restricted" transducers $\mathcal{T}_e$ is finite, closure under union terminates the proof. ■

The proof is constructive.

As an application of this theorem, one may give a generalization of Proposition 3.10.(i):

Proposition 3.12 If σ is δ-synchronous and τ is left-synchronous, then $\sigma.\tau$ is left-synchronous.

Notice that the left and right synchronizable transducer example in 3.6—which is even recognizable—does not satisfy conditions of Theorem 3.19, since the transmission rate of some cycles is 2.

Resynchronization Algorithm

Although left-synchronism is not decidable, one may be interested in a synchronization algorithm that work on a subset of left-synchronizable transducers: the class of transducers satisfying the hypothesis of Theorem 3.19.

Extending an implementation by Béal and Carton [BC99a] of the algorithm in [FS93], it is possible to "resynchronize" our larger class along the lines of the proof of Theorem 3.19. This technique will be used extensively in Sections 3.6 and 3.7, to compute—possibly approximative—intersections of rational relations. Presentation of the full algorithm and further investigations about its complexity are left for future work.

3.4.6 Decidability Results

We first present an extension of the *minimality* concept for finite-state automata to left-synchronous transducers. Let $\mathcal{T} = (Q, I, F, E)$ be a transducer over alphabets A and B. We define the following predicate, for $q \in Q$ and $(u, v) \in A^* \times B^*$:

$$\mathsf{Accept}(q, u, v) \text{ iff } (u, v) \text{ labels an accepting path starting at } q.$$

Nerode's equivalence, noted $\equiv$, is defined by

$$q \equiv q' \text{ iff for all } (u, v) \in A^* \times B^* : \mathsf{Accept}(q, u, v) \iff \mathsf{Accept}(q', u, v).$$

The equivalence class of $q \in Q$ is denoted by $\hat{q}$. Let

$$\mathcal{T}/\equiv = (Q/\equiv, I/\equiv, F/\equiv, \hat{E}),$$

where $\hat{E}$ is naturally defined by

$$(\hat{q}_1, x, y, \hat{q}_2) \in \hat{E} \iff \exists (q_1', q_2') \in \hat{q}_1 \times \hat{q}_2 : (q_1', x, y, q_2') \in E.$$

Using Nerode's equivalence, we extend the concept of minimal automaton to left-synchronous transducers.

Theorem 3.20 Any left-synchronous transduction is realized by a unique minimal left-synchrnonous transducer (up to a renaming of states).

Proof: Let τ be a transduction over alphabets A and B, realized by a left-synchronous transducer $\mathcal{T} = (Q, I, F, E)$. We suppose without loss of generality that $\mathcal{T}$ is trim. By definition of $\equiv$, it is clear that $\mathcal{T}/\equiv$ realizes τ.

Every transition on $\mathcal{T}/\equiv$ is labeled on $A \times B \cup A \times \{\varepsilon\} \cup \{\varepsilon\} \times B$. Consider two states $q, q' \in Q$ such that $q \equiv q'$ and q holds an input transition labeled on $A \times \{\varepsilon\}$ (resp. $\{\varepsilon\} \times B$); and consider $(u, v) \in A^* \times B^*$ such that $\mathsf{Accept}(q, u, v)$ and $\mathsf{Accept}(q', u, v)$. Any output transition from q must be labeled on $A \times \{\varepsilon\}$ (resp. $\{\varepsilon\} \times B$), hence v (resp. u) must be empty. Since this is true for all accepted (u, v), and since $\mathcal{T}$ is trim, any output transition from q' must also be labeled on $A \times \{\varepsilon\}$ (resp. $\{\varepsilon\} \times B$); this proves that $\mathcal{T}/\equiv$ is left-synchronous.

Finally, let $\mathcal{A}$ be the finite-state automaton interpretation of $\mathcal{T}$ (see Definition 3.6). It is well known that $\mathcal{A}/\equiv$ is the unique minimal automaton realizing the same rational language as A (up to a renaming of states). Thus, if $\mathcal{T}'$ is an realization of τ with as many states as $\mathcal{T}/\equiv$, its finite-state automaton interpretation must be $\mathcal{A}/\equiv$ (up to a renaming of states) which is the interpretation of $\mathcal{T}/\equiv$. This proves the unicity of the minimal left-synchronous transducer. ■

As a corollary of closure under complementation and intersection, usual questions become decidable for left-synchronous transductions:

Lemma 3.7 Let R, R' be left-synchronous relations over alphabets A and B. It is decidable whether $R \cap R' = \varnothing$, $R \subset R'$, $R = R'$, $R = A^* \times B^*$, $(A^* \times B^*) - R$ is finite.

These properties are essential for formal reasoning about dependence and reaching definition abstractions in the following chapter.

Eventually, we are still working on decidability of recognizable relations among left-synchronous ones. We have strong arguments to expect a positive result, but no proof at the moment.

3.4.7 Further Extensions

We now consider possible extensions of left-synchronizable relations.

Constant Transmission Rates

An elementary variation on *synchronous* transducers consists in enforcing a single transmission rate in all cycles which is not necessary 1: if k and l are positive integers, a (k, l)-synchronous relation over $A^* \times B^*$ is realized by a transducer whose transitions are labeled in $A^k \times B^l$. Similarly, one may define δ-(k, l)-synchronous and left-(k, l)-synchronous transducers.

When noticing that a change of the alphabet converts a (k, l)-synchronous transducer into a classical synchronous one, it obviously appears that the same properties are satisfied for any k and l, including $k = l = 1$. The only difference is that transmission rates of cycles is now 0, $+\infty$ and k/l. Mixing relations in (k, l)-synchronous classes for different (k, l) is not allowed, of course.

However, most rational transductions useful to our framework, including orders, are left-$(1, 1)$-synchronous, that is left-synchronous... This strongly reduces the usefulness of general left-(k, l)-synchronous transductions.

Deterministic Transducers

Much more interesting is the class of *deterministic* relations introduced by Pelletier and Sakarovitch in [PS98]:

Definition 3.19 (deterministic transducer and relation) Let A and B be two alphabets. A transducer $\mathcal{T} = (A^*, B^*, Q, I, F, E)$ is deterministic if the following conditions hold:

(i) there exists a partition of the set of states $Q = Q_A \cup Q_B$ such that the label of an edge departing from a state in Q_A is in $A \times \{\varepsilon\}$ and the label of an edge departing from a state in Q_B is in $\{\varepsilon\} \times B$;

(ii) for every $p \in Q$ and every $(x, y) \in (A \times \{\varepsilon\}) \cup (\{\varepsilon\} \times B)$, there exists at most one $q \in Q$ such that (p, x, y, q) is in E (i.e. the finite-state automaton interpretation is deterministic);

(iii) there is a single initial state in I.

A deterministic relation is realized by a deterministic transducer.

This class is strictly larger than left-synchronous relations, and keeps most of its good properties: the greatest loss is closure under composition. Moreover, because relation U^+ is deterministic in the proof of Theorem 3.16, it is undecidable whether a deterministic relation is recognizable, left-synchronous or both left and right synchronous.

But the most important reason for us to use left-synchronous relations instead of deterministic ones is that there is no result such as Theorem 3.19 to find a deterministic realization of a relation, or to help approximate a rational relation by a deterministic one.

3.5 Beyond Rational Relations

For the purpose of our program analysis framework, we sometimes require more expressiveness than rational relations: "finite automata cannot count", and we need counting to handle arrays! We thus present an extension of the algebraic—also known as context-free—property to relations between finitely generated monoids. As one would expect, the class of algebraic relations includes rational relations, and retains several decidable properties. This sections ends with a few contributions: Theorems 3.27 and 3.28, and Proposition 3.13.

3.5.1 Algebraic Relations

We define algebraic relations through push-down transducers, defined similarly to push-down automata (see Section 3.2.3).

Definition 3.20 (push-down transducer) Given alphabets A and B, a *push-down transducer* $\mathcal{T} = (A^*, B^*, \Gamma, \gamma_0, Q, I, F, E)$—a.k.a. *algebraic transducer*—consists of a stack alphabet Γ, a *non-empty* word γ_0 in Γ^+ called the initial stack word, a finite set Q of states, a set $I \subset Q$ of initial states, a set $F \subset Q$ of final states, and a finite set of transitions (a.k.a. edges) $E \subset Q \times A^* \times B^* \times \Gamma \times \Gamma^* \times Q$.

Free monoids A^* and B^* are often removed for commodity, when clear from the context. A transition $(q, x, y, g, \gamma, q') \in E$ is usually written $q \xrightarrow{x|y:g \to \gamma} q'$. The push-down automata and rational transducer vocabularies are inherited.

A configuration of a push-down automaton is a quadruple (u, v, q, γ), where (u, v) is the pair of word to be accepted or rejected, q is the current state and $\gamma \in \Gamma^*$ is the word composed of symbols in the stack. The *transition* between two configurations $c_1 = (u_1, v_1, q_1, \gamma_1)$ and $c_2 = (u_2, v_2, q_2, \gamma_2)$ is denoted by relation $\mapsto$ and defined by $c \longmapsto c'$ iff there exist $(x, y, g, \gamma, \gamma') \in A^* \times B^* \times \Gamma \times \Gamma^* \times \Gamma^*$ such that

$$u_1 = xu_2 \wedge v_1 = yv_2 \wedge \gamma_1 = \gamma' g \wedge \gamma_2 = \gamma' \gamma \wedge (q_1, x, y, g, \gamma, q_2) \in E.$$

Then $\overset{p}{\longmapsto}$ with $p \in \mathbb{N}$, $\overset{+}{\longmapsto}$ and $\overset{*}{\longmapsto}$ are defined as usual.

A push-down transducer $\mathcal{T} = (\Gamma, \gamma_0, Q, I, F, E)$ is said to *realize* the relation R, when $(u, v) \in R$ iff there exist $(q_i, q_f, \gamma) \in I \times F \times \Gamma^*$ such that

$$(u, v, q_i, \gamma_0) \overset{*}{\longmapsto} (\varepsilon, \varepsilon, q_f, \gamma).$$

A push-down transducer $\mathcal{T} = (\Gamma, \gamma_0, Q, I, F, E)$ is said to ***realize*** the relation R, when $(u, v) \in R$ iff there exist $(q_i, q_f) \in I \times F$ such that

$$(u, v, q_i, \gamma_0) \stackrel{*}{\longmapsto} (\varepsilon, \varepsilon, q_f, \varepsilon).$$

Notice that realization by empty stack implies realization by finite state: q_f is still required to be in the set of final states.

Definition 3.21 (algebraic relation) The class of relations realized by *final state* or by *empty stack* by push-down transducers is called the class of ***algebraic relations***.

As for rational relations, the following characterization of algebraic relations is fundamental: it allows to express algebraic relations by means of algebraic languages and monoid morphisms. A proof in a much more general case can be found in [Kar92]. (Berstel uses this theorem as a definition for algebraic relations in [Ber79].)

Theorem 3.21 (Nivat) Let A and B be two alphabets. Then R is an algebraic relation over A^* and B^* iff there exist an alphabet C, two morphisms $\phi : C^* \to A^*$, $\psi : C^* \to B^*$, and an algebraic language $L \subset C^*$ such that

$$R = \{(\phi(h), \psi(h)) : h \in L\}.$$

To generalize Section 3.3.2, algebraic transductions are the functional counterpart of algebraic relations.

Nivat's theorem can be formulated as follows for algebraic transductions:

Theorem 3.22 (Nivat) Let A and B be two alphabets. Then $\tau : A^* \to B^*$ is an algebraic transduction iff there exist an alphabet C, two morphisms $\phi : C^* \to A^*$, $\psi : C^* \to B^*$, and an algebraic language $L \subset C^*$ such that

$$\forall w \in A^* : \quad \tau(w) = \psi(\phi^{-1}(w) \cap L).$$

Let us recall some useful properties of algebraic relations and transductions.

Theorem 3.23 Algebraic relations are closed under union, concatenation, and the star operation. They are also closed under composition with rational transductions (similar to Elgot and Mezei theorem). The image of a rational language by an algebraic transduction is an algebraic language (thanks to Nivat's theorem).

The image of an algebraic language by an algebraic transduction may not be algebraic, but there are some interesting exceptions:

Theorem 3.24 (Evey) Given a push-down transducer $\mathcal{T}$, if L is the algebraic language realized by the input automaton of $\mathcal{T}$ (see Definition 3.8), the image $\mathcal{T}(L)$ is an algebraic language.

The following definition will be useful in some technical discussions and proofs in the following. It formalizes the fact that a push-down transducer can be interpreted as a push-down automaton on a more complex alphabet. But beware: both interpretations have different properties in general.

Definition 3.22 Let $\mathcal{T}$ be a push-down transducer over alphabets A and B. The ***push-down automaton interpretation*** of $\mathcal{T}$ is a push-down automaton $\mathcal{A}$ over the alphabet $(A \times B) \cup (A \times \{\varepsilon\}) \cup (\{\varepsilon\} \times B)$ defined by the same stack alphabet, initial stack word, states, initial states, final states and transitions.

Among the usual decision problems, only the following are available for algebraic relations:

Theorem 3.25 The following problems are decidable for algebraic relations: whether two words are in relation (in linear time), emptiness, finiteness.

Important remarks. In the following, every push-down transducer will implicitly accept words by *final state*. Recognizable and rational relations were defined for any finitely generated monoids, but algebraic relations are defined for free monoids only.

Algebraic Functions

There are very few results about algebraic transductions that are partial functions. Here is the definition:

Definition 3.23 (algebraic function) Let A and B be two alphabets. An *algebraic function* $f : A^* \to B^*$ is an algebraic transduction which is a partial function, i.e. such that $\mathrm{Card}(f(u)) \leq 1$ for all $u \in A^*$.

However, we are not aware of any decidability result for an algebraic transduction to be a partial function, and we believe that the most likely answer is negative.

Among transducers realizing algebraic functions, we are especially interested in transducers whose output can be "computed online" with its input. As for rational transducers, our interpretation for "online computation" is based on the determinism of the input automaton:

Definition 3.24 (online algebraic transducer) An algebraic transducer is *online* if it is a partial function and if its input automaton is deterministic. An algebraic transduction is *online* if it is realized by an online algebraic transducer.

Nevertheless, we are not aware of any results for this class of algebraic functions; even decidability of deterministic algebraic languages among algebraic ones is unknown.

3.5.2 One-Counter Relations

An interesting sub-class of algebraic relations is called the class of one-counter relations. It is defined through push-down transducers. A classical definition is the following:

Definition 3.25 A push-down transducer is a *one-counter transducer* if its stack alphabet contains only one letter. An algebraic relation is a *one-counter relation* if it is realized by a one-counter transducer (by final state).

As for one-counter languages, we prefer a definition which is more suitable to our practical usage of one-counter relations.

Definition 3.26 (one-counter transducer and relation) A push-down transducer is a *one-counter transducer* if its stack alphabet contains three letters, Z (for "zero"), I (for "increment") and D (for "decrement") and if the stack word belongs to the (rational) set $ZI^* + ZD^*$. An algebraic relation is a *one-counter relation* if it is realized by a one-counter transducer (by final state).

It is easy to show that Definition 3.26 describes the same family of languages as the preceding classical definition.

We use the same notations as for one-counter languages, see Section 3.2.4. The family of one-counter relations is *strictly* included in the family of algebraic relations.

Notice that using more than one counter gives the same expressive power as Turing machines, as for multi-counter automata, see the last paragraph in Section 3.2.4 for further discussions about this topic.

Now, why are we interested in such a class of relations? We will see in our program analysis framework that we need to compose *rational* transductions over non-free monoids. Indeed, the well known theorem by Elgot and Mezei (Theorem 3.5 in Section 3.3) can be "partly" extended to any finitely generated monoids:

Theorem 3.26 (Elgot and Mezei) If M_1 and M_2 are finitely generated monoids, A is an alphabet, $\tau_1 : M_1 \to A^*$ and $\tau_2 : A^* \to M_2$ are rational transductions, then $\tau_2 \circ \tau_1 : M_1 \to M_2$ is a rational transduction.

But this extension is not interesting in our case, since the "middle" monoid in our transduction composition is *not* free. More precisely, we would like to compute the composition of two *rational* transductions $\tau_2 \circ \tau_1$, when $\tau_1 : A^* \to \mathbb{Z}^n$ and $\tau_2 : \mathbb{Z}^n \to B^*$, for some alphabets A and B and some positive integer n. Sadly, because of the *commutative group* nature of $\mathbb{Z}$, composition of τ_2 and τ_1 is not a rational transduction in general. An intuitive view of this comes from the fact that all "words" on $\mathbb{Z}$ of the form

$$\underbrace{1 + 1 + \cdots + 1}_{k}\underbrace{-1 - 1 - \cdots - 1}_{k}$$

are equal to 0, but do not build a rational language in $\{1, -1\}^*$ (they built a context-free one).

We have proven that such a composition yields a n-counter transduction in general, and the proof gives a constructive way to build a transducer realizing the composition:

Theorem 3.27 Let A and B be two alphabets and let n be a positive integer. If $\tau_1 : A^* \to \mathbb{Z}^n$ and $\tau_2 : \mathbb{Z}^n \to B^*$ are rational transductions, then $\tau_2 \circ \tau_1 : A^* \to B^*$ is a n-counter transduction.

Proof: We first suppose that n is equal to 1. Let $\mathcal{T}_1 = (A^*, \mathbb{Z}, Q_1, I_1, F_1, E_1)$ realize τ_1 and $\mathcal{T}_2 = (\mathbb{Z}, B^*, Q_2, I_2, F_2, E_2)$ realize τ_2. We define a one-counter transducer $\mathcal{T}_1' = (A^*, B^*, 0, Q_1, I_1, F_1, E_1')$—with no output on B^*—from $\mathcal{T}_1$: if $(q, u, v, q') \in E_1$ then $(q, u, \varepsilon, \varepsilon, +v, q') \in E_1'$ (no counter check). Similarly, we define a one-counter transducer $\mathcal{T}_2' = (A^*, B^*, 0, \ldots, c_0^n, Q_2, I_2, F_2, E_2')$—with no input from A^*—from $\mathcal{T}_2$: if $(q, u, v, q') \in E_2$ then $(q, \varepsilon, v, \varepsilon, -u, q') \in E_2'$ (no counter check). Intuitively, the output of $\mathcal{T}_1$ and $\mathcal{T}_2$ are replaced by counter updates in $\mathcal{T}_1'$ and opposite counter updates in $\mathcal{T}_2'$.

Then we define a one-counter transducer $\mathcal{T} = (A^*, B^*, 0, Q_1 \cup Q_2 \cup \{q_F\}, I_1, \{q_F\}, E)$ as a kind of concatenation of $\mathcal{T}_1'$ and $\mathcal{T}_2'$:

- if $e \in E_1'$ then $e \in E$;
- if $e \in E_2'$ then $e \in E$;

- if $q_1 \in F_1$ and $q_2 \in I_2$ then $(q_1, \varepsilon, \varepsilon, \varepsilon, \varepsilon, q_2) \in E$ (neither counter check nor counter update);
- if $q_2 \in F_2$ then $(q_2, \varepsilon, \varepsilon, {=}0, \varepsilon, q_F) \in E$ (no counter update);
- no other transition is in E.

Intuitively, $\mathcal{T}$ accepts pairs of words (u, v) when (u, ε) would be accepted by $\mathcal{T}_1$, (ε, v) would be accepted by $\mathcal{T}_2$ and the counter is zero when reaching state q_F. Then, $\mathcal{T}$ is a one-counter transducer and recognizes $\tau_2 \circ \tau_1$.

Finally, if n is greater than 1, the same construction can be applied to each dimension of $\mathbb{Z}^n$, and the associated counter check and updates can be combined to build a n-counter transducer realizing $\tau_2 \circ \tau_1$. ■

Theorem 3.27 will be used in Section 4.3 to prove properties of the dependence analysis. In practice, we will restrict ourselves to $n = 1$ applying conservative approximations described in Section 3.7, either on τ_1 and τ_2 or on the multi-counter composition.

We now require an additional formalization of the rational transducer "skeleton" of a push-down transducer.

Definition 3.27 (underlying rational transducer) Let $\mathcal{T} = (\Gamma, \gamma_0, Q, I, F, E)$ be a push-down transducer. We can build a rational transducer $\mathcal{T}' = (Q, I, F, E')$ from $\mathcal{T}$ in setting

$$(q, x, y, q') \in E' \iff \exists g \in \Gamma, \gamma \in \Gamma^* : (q, x, y, g, \gamma, q') \in E.$$

The *underlying rational transducer* of $\mathcal{T}$ is the rational transducer obtained in trimming $\mathcal{T}'$ and removing all transitions labeled $\varepsilon|\varepsilon$.

Looking at the proof of Theorem 3.27, there is a very interesting property about transducer $\mathcal{T}$ realizing $\tau_2 \circ \tau_1$: the transmission rate of every cycle in $\mathcal{T}$ is either 0 or $+\infty$. Thanks to Lemma 3.5 in Section 3.4, we have proven the following result:

Proposition 3.13 Let A and B be two alphabets and let n be a positive integer. Let $\tau_1 : A^* \to \mathbb{Z}^n$ and $\tau_2 : \mathbb{Z}^n \to B^*$ be rational transductions and let $\mathcal{T}$ be a n-counter transducer realizing $\tau_2 \circ \tau_1 : A^* \to B^*$ (computed from Theorem 3.27). Then, the *underlying rational transducer* of $\mathcal{T}$ is *recognizable.*

Applications of this result include closure under intersection with *any rational transduction*, thanks to the technique presented in Section 3.6.2.

Eventually, when studying abstract models for data structures, we have seen that nested trees and arrays are neither modeled by free monoids nor by free commutative monoids. Their general structure is called a free partially commutative monoid, see Section 2.3.3. Let A and B be two alphabets, and M be such a monoid with binary operation $\bullet$. We still want to compute the composition of *rational* transductions $\tau_2 \circ \tau_1$, when $\tau_1 : A^* \to M$ and $\tau_2 : M \to B^*$. The following result is an extension of Theorem 3.27, and its proof is still constructive:

Theorem 3.28 Let A and B be two alphabets and let M be a free partially commutative monoid. If $\tau_1 : A^* \to M$ and $\tau_2 : M \to B^*$ then $\tau_2 \circ \tau_1 : A^* \to B^*$ is a multi-counter transduction. The number of counters is equal to the maximum dimension of vectors in M (see Definition 2.6).

Proof: Because the full proof is rather technical while its intuition is very natural, we only sketch the main ideas. Considering two rational transducers $\mathcal{T}_1$ and $\mathcal{T}_2$ realizing τ_1 and τ_2 respectively, we start applying the classical composition algorithm for free monoids to build a transducer $\mathcal{T}$ realizing $\tau_2 \circ \tau_1$. But this time, $\mathcal{T}$ will be multi-counter, every counter is initialized to 0, and transitions generated by the classical composition algorithm simply ignore the counters.

Now, every time a transition of $\mathcal{T}_1$ writes a vector v (resp. $\mathcal{T}_2$ reads a vector v), the "normal execution" of the classical composition algorithm is "suspended", only transitions reading (resp. writing) vectors of the same dimension as v are considered in $\mathcal{T}_2$ (resp. $\mathcal{T}_1$), and v is added to the counters using the technique in Theorem 3.27. When a letter is read or written during the "suspended mode", each counter is checked for zero before "resuming" the "normal execution" of the classical composition algorithm.

The result is a transducer with rational and multi-counter parts, separated by checks for zero. ■

Theorem 3.28 will also be used in Section 4.3.

3.6 More about Intersection

Intersecting relations is a major issue in our analysis and transformation framework. We have seen that this operation neither preserve the rational property nor the algebraic property of a relation; but we have also found sub-classes of relations, closed under intersection. The purpose of this section is to extend these sub-classes in order to support special cases of intersections.

3.6.1 Intersection with Lexicographic Order

For the purpose of dependence analysis, we have already mentioned the need for intersections with the lexicographic order. Indeed, the class of left-synchronous relations includes the lexicographic order and is closed under intersection.

In this section, we restrict ourselves to the case of relations over $A^* \times A^*$ for some alphabet A. We will describe a class larger than synchronous relations over $A^* \times A^*$ which is closed under intersection *with the lexicographic order* only.[6]

Definition 3.28 (pseudo-left-synchronism) Let A be an alphabet. A rational transducer $\mathcal{T} = (A, A, Q, I, F, E)$ (same alphabet A) is *pseudo-left-synchronous* if there exist a partition of the set of states $Q = Q_I \cup Q_S \cup Q_T$ satisfying the following conditions:

(i) any transition between states of Q_I is labeled $x|x$ for some a in A;

(ii) any transition between a state of Q_I and a state of Q_T is labeled $x|y$ for some $x \neq y$ in A;

(iii) the restriction of $\mathcal{T}$ to states in $Q_I \cup Q_S$ is left-synchronous.

A rational relation or transduction is pseudo-left-synchronous if it is realized by a pseudo-left-synchronous transducer. A rational transducer is *pseudo-left-synchronizable* if it realizes a pseudo-left-synchronous relation.

[6]This class is not comparable with the class of deterministic relations proposed in Definition 3.19 of Section 3.4.7.

An intuitive view of this definition would be that a pseudo-left-synchronous transducer satisfies the left-synchronism property everywhere but after transitions labeled $x|y$ with $x \neq y$. The motivation for such a definition comes from the following result:

Proposition 3.14 The class of pseudo-left-synchronous relations is closed under intersection with the lexicographic order.

Proof: Because the non-left-synchronous part is preceded by transitions labeled $x|y$ with $x \neq y$, which are themselves preceded by transitions labeled $x|x$, intersection with the lexicographic order becomes straightforward on this part: if $x < y$ the transition is kept in the intersection, otherwise it is removed. Intersecting the left-synchronous part is done thanks to Theorem 3.14. ■

Another intersecting result is the following:

Proposition 3.15 Intersecting a pseudo-left-synchronous relation with the identity relation yields a *left-synchronous* relation.

Proof: Same idea as the preceding proof, but transitions $x|y$ with $x \neq y$ are now removed every time. ■

Of course, pseudo-left-synchronous relations are closed under union, but not intersection, complementation and composition.

Eventually, the constructive proof of Theorem 3.19 can be modified to look for pseudo-left-synchronous relations: when a transition labeled $x|y$ is found after a path of transitions labeled $x|x$, leave the following transitions unchanged.

3.6.2 The case of Algebraic Relations

What about intersection of algebraic relations? The well known result about closure of algebraic languages under intersection with rational languages has no extension to algebraic relations. Still, it is easy to see that there is a property similar to left-synchronism which brings partial intersection results for algebraic relations.

Proposition 3.16 Let R_1 be an algebraic relation realized by a push-down transducer whose *underlying rational transducer* is left-synchronous, and let R_2 be a left-synchronous relation. Then $R_1 \cap R_2$ is an algebraic relation, and one may compute a push-down transducer realizing the intersection whose underlying rational transducer is left-synchronous.

Proof: Let $\mathcal{T}_1$ be a push-down automaton realizing R_1 whose underlying rational transducer $\mathcal{T}_1'$ is left-synchronous, and let $\mathcal{T}_2$ be a left-synchronous realization of R_2. The proof comes from the fact that intersecting $\mathcal{T}_1'$ and $\mathcal{T}_2$ can be done without "forgetting" the original stack operation associated with each transition in $\mathcal{T}_1$. This is due to the cross-product nature of the intersection algorithm for finite-state automata (which also applies to left-synchronous transducers). ■

Of course, the pseudo-left-synchronism property can be used instead of the left-synchronous one, yielding the following result:

Proposition 3.17 Let A be an alphabet and let R be an algebraic relation over $A^* \times A^*$ realized by a push-down transducer whose underlying rational transducer is pseudo-left-synchronous. Then intersecting R with the lexicographic order (resp. identity relation) yields an algebraic relation, and one may compute a push-down transducer realizing the intersection whose underlying rational transducer is pseudo-left-synchronous (resp. left-synchronous).

3.7 Approximating Relations on Words

This section is a transition between the long study of mathematical tools exposed in this chapter and applications of these tools to our analysis and transformation framework. Remember we have seen in Section 2.4 that exact results were not required for data-flow information, and that our program transformations were based on *conservative approximations* of sets and relations. Studying approximations is rather unusual when dealing with words and relations between words, but we will show its practical interest in the next chapters.

Of course, such conservative approximations must be as precise as possible, and exact results should be looked for every time it is possible. Indeed, approximations are needed only when a question or an operation on rational or algebraic relations is not decidable. Our general approximation scheme for rational and algebraic relations is thus to find a conservative approximation in a smaller class which supports the required operation or for which the required question is decidable.

3.7.1 Approximation of Rational Relations by Recognizable Relations

Sometimes a recognizable approximation of a rational relation may be needed. If R is a rational relation realized by a rational transducer $\mathcal{T} = (Q, I, F, E)$, the simplest way to build a recognizable relation K which is *larger* than R is to define K as the product of input and output languages of R.

A smarter approximation is to consider each pair (q_i, q_f) of initial and final states in $\mathcal{T}$, and to define K_{q_i,q_f} as the product of input and output languages of the relation realized by $(Q, \{q_i\}, \{q_f\}, E)$. Then K is defined as the union of all K_{q_i,q_f} for all $(q_i, q_f) \in I \times F$. This builds a recognizable relation thanks to Mezei's Theorem 3.3.

The next level of precision is achieved in considering each strongly-connected component in $\mathcal{T}$ and approximating it with the preceding technique. The resulting relation K is still recognizable, thanks to Mezei's theorem. This technique will be considered in the following when looking for a recognizable approximation of a rational relation.

3.7.2 Approximation of Rational Relations by Left-Synchronous Relations

Because recognizable approximations are not precise enough in general, and because the class of left-synchronous relations retains most interesting properties of recognizable relations, we will rather approximate rational relations by left-synchronous ones.

The key algorithm in this context is based on the constructive proof of Theorem 3.19 presented in Section 3.4.5. In practical cases, it often returns a left-synchronous transducer and no approximation is necessary. When it fails, it means that some strongly-connected component could not be resynchronized. The idea is then to approximate this strongly connected component by a recognizable relation, and then to restart the resynchronization algorithm.

For better efficiency, all strongly-connected components whose transmission rate is not 0, 1 or $+\infty$ should be approximated this way in a first stage. In the same stage, if a strongly-connected component C whose transmission rate is 1 follows some strongly-connected components $C_1, \dots, C_n$ whose transmission rates are 0 or $+\infty$, then a recognizable approximation K_C of C should be added to the transducer with same outgoing transitions as C, and all paths from $C_1, \dots, C_n$ to C should now lead to K_C. Applying such a first stage guarantees that the resynchronization algorithm will return a left-synchronous approximation of R, thanks to Theorem 3.19.

Eventually, when trying to intersect a rational transducer with the lexicographic order, we are looking for a pseudo-left-synchronous approximation. The same technique as before can then be applied, using the extended version of Theorem 3.19 proposed in Section 3.6.

3.7.3 Approximation of Algebraic and Multi-Counter Relations

There are two very different techniques when approximating algebraic relations. The simplest one is used to give conservative results to a few undecidable questions for algebraic transducers that are decidable for rational ones. It consists in taking the underlying rational transducer as a conservative approximation. Precision can be slightly improved when the stack size is bounded: the finite number of possible stack words can be encoded in state names. This may induce a large increase of the number of states. The second technique is used when looking for an intersection with a left-synchronous relation: it consists in approximating the underlying rational transducer with a left-synchronous (or pseudo-left-synchronous) one without modifying the stack operations. In fact, stack operations can be preserved in the resynchronization algorithm (associated with Theorem 3.19), but they are obviously lost when approximating a strongly-connected component with a recognizable relation. Which technique is applied will be stated every time an approximation of an algebraic relation is required.

Eventually, we have seen that composing two rational transductions over $\mathbb{Z}^n$ yields a n-counter transduction by Theorem 3.27. Approximation by a one-counter transduction then consists in saving the value of bounded counters into new states names, then removing all unbounded counters but one. Smart choices of the remaining counter and attempts to combine two counters into one have not been studied yet, and are left for future work.

Chapter 4

Instancewise Analysis for Recursive Programs

Even though dependence information is at the core of virtually all modern optimizing compilers, recursive programs have not received much attention. When considering instancewise dependence analysis for recursive data structures, less than three papers have been published. Even worse is the state of the art in reaching definition analysis: before our recent results for arrays [CC98], no instancewise reaching definition analysis for recursive programs has been proposed.

Considering the program model proposed in Chapter 2, we now focus on dependence and reaching definition analysis at the run-time instance level. The following presentation is built on our previous work on the subject [CCG96, Coh97, Coh99a, Fea98, CC98], but has been going through several major evolutions. It results in a much more general and mathematically sound framework, with algorithms for automation of the whole analysis process, but also in a more complex presentation. The primary goal of this work is rather theoretical: we look for the highest precision possible. Beyond this important target, we will show in a later chapter (see Section 5.5) how this precise information can be used to outperform current results in parallelization of recursive programs, and also to enable new program transformation techniques.

We start our presentation with a few motivating examples, then discuss induction variable and storage mapping function computation in Section 4.2, the general analysis technique is presented in Section 4.3, with questions specific to particular data structures deferred to the next sections. Eventually, Section 4.7 compares our results with static analyses and with recent works on instancewise analysis for loop nests.

4.1 Motivating Examples

Studying three examples, we present an intuitive flavor of the instancewise dependence and reaching definition analyses for recursive control and data structures.

4.1.1 First Example: Procedure `Queens`

Our first example is still the procedure `Queens`, presented in Section 2.3. It is reproduced here in Figure 4.1.a with a partial control tree.

Studying accesses to array `A`, our purpose is to find dependences between *run-time instances* of program statements. Let us study instance $FPIA\textsc{a}a\textsc{a}a\textsc{a}JQPIA\textsc{a}B\textsc{b}r$ of state-

```
      int A[n];

P     void Queens (int n, int k) {
I       if (k < n) {
A/A/a     for (int i=0; i<n; i++) {
B/B/b       for (int j=0; j<k; j++)
r             ··· = ··· A[j] ···;
J           if (···) {
s             A[k] = ···;
Q             Queens (n, k+1);
            }
          }
        }
      }

      int main () {
F       Queens (n, 0);
      }
```

Figure 4.1.a. Procedure `Queens`

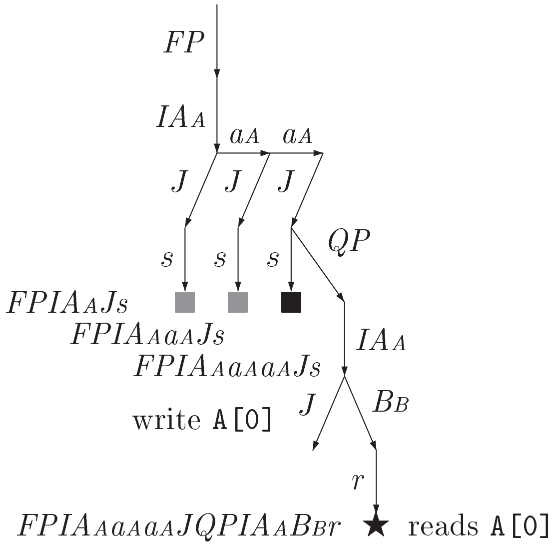


Figure 4.1.b. Compressed control tree

Figure 4.1. Procedure `Queens` *and control tree*

ment r, depicted as a star in Figure 4.1.b. In order to find some dependences, we would like to know which memory location is accessed. Since `j` is initialized to 0 in statement B, and incremented by 1 in statement b, we know that the value of variable `j` at $FPIA_Aa_Aa_AJQPIA_AB_Br$ is 0, so $FPIA_Aa_Aa_AJQPIA_AB_Br$ reads `A[0]`.

We now consider instances of s, depicted as squares: since statement s writes into `A[k]`, we are interested in the value of variable `k`: it is initialized to 0 in `main` (by the first call `Queens(n, 0)`), and incremented at each recursive call to procedure `Queens` in statement Q. Thus, instances such as $FPIA_AJs$, $FPIA_Aa_AJs$ or $FPIA_Aa_Aa_AJs$ write into `A[0]`, and are therefore in dependence with $FPIA_Aa_Aa_AJQPIA_AB_Br$.

Let us now derive which of these definitions *reaches* $FPIA_Aa_Aa_AJQPIA_AB_Br$. Looking again at Figure 4.1.b, we notice that instance $FPIA_Aa_Aa_AJs$—denoted by a black square—is, among the three possible reaching definitions that are shown, the last to execute. And it does execute: since we assume that $FPIA_Aa_Aa_AJQPIA_AB_Br$ executes, then $FPIA_Aa_Aa_AJ$ (hence $FPIA_Aa_Aa_AJs$) has to execute. Therefore, other instances writing in the same array element, such as $FPIA_AJs$ and $FPIA_Aa_AJs$, cannot reach the read instance, since their value is always overwritten by $FPIA_Aa_Aa_AJs$.[1] Noticing that no other instance of s could execute after $FPIA_Aa_Aa_AJs$, we can ensure that $FPIA_Aa_Aa_AJs$ is the reaching definition of $FPIA_Aa_Aa_AJQPIA_AB_Br$. We will show later how this simple approach to computing reaching definitions can be generalized.

[1] $FPIA_Aa_Aa_AJs$ is then called an *ancestor* of $FPIA_Aa_Aa_AJQPIA_AB_Br$, to be formally defined later.

4.1.2 Second Example: Procedure BST

Let us now look at procedure BST, as shown in Figure 4.2. This procedure swaps node values to convert a binary tree into a binary search tree (BST). Nodes of the tree structure are referenced by pointers; `p->l` (resp. `p->r`) denotes the pointer to the left (resp. right) child of the node pointed by `p`; `p->value` denotes the integer value of the node.

```
P    void BST (tree *p) {
I1     if (p->l!=NULL) {
L        BST (p->l);
I2       if (p->value < p->l->value) {
a          t = p->value;
b          p->value = p->l->value;
c          p->l->value = t;
         }
       }
J1     if (p->r!=NULL) {
R        BST (p->r);
J2       if (p->value > p->r->value) {
d          t = p->value;
e          p->value = p->r->value;
f          p->r->value = t;
         }
       }
     }

     int main () {
F      if (root!=NULL) BST (root);
     }
```

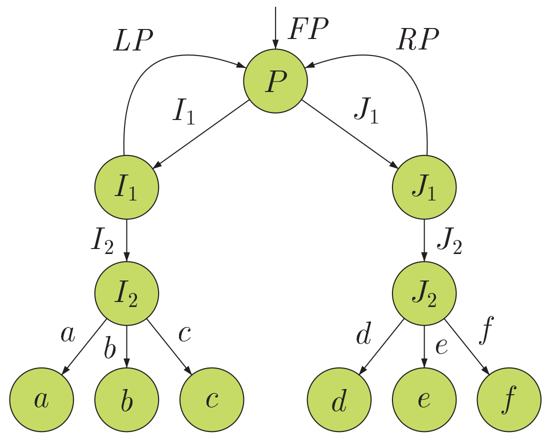


Figure 4.2. Procedure BST *and compressed control automaton*

There are few dependences on program BST. If u is an instance of block I_2, then there are anti-dependences between the first read access in u and instance ub, between the second read access in u and uc, between the read access in instance ua and instance ub, and between the read access in ub and instance uc. It is the same for an instance v of block J_2: there is an anti-dependence between the first read access in u and ue, between the read access in u and uf, between the read access in ud and ue, and between the read access in ue and uf. No other dependences are found. We will show in the following how to compute this result automatically. Eventually, a reaching definition analysis tells that $\perp$ is the unique reaching definiton of each read access.

4.1.3 Third Example: Function Count

Our last motivating example is function Count, as shown in Figure 4.3. It operates on the `inode` structure presented in Section 2.3.3. This function computes the size of a file in blocks, in counting terminal `inodes`.

Since there is no write access to the `inode` structure, there are no dependences on the Count program (not considering the other data structures, such as scalar `c`). How-

```
P      int Count (inode *p) {
I        if (p->terminal)
a          return p->length;
E        else {
b          c = 0;
L/L/l      for (int i=0; i<p->length; i++)
c            c += Count (p->n[i]);
d          return c;
         }
       }

       int main () {
F        Count (file);
       }
```

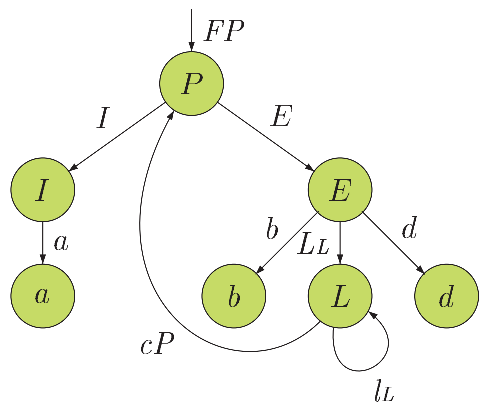


Figure 4.3. Procedure `Count` *and compressed control automaton*

ever, an interesting result for cache optimization techniques [TD95] would be that each memory location is read only once. We will show that this information can be computed automatically by our analysis techniques.

4.1.4 What Next?

In the rest of this chapter, we formalize the concepts introduced above. In Section 4.2, we compute *maps* from instance names to data-element names. Then, the dependence and reaching definitions relation are computed in Section 4.3.

4.2 Mapping Instances to Memory Locations

In Section 2.4, we defined *storage mappings* from *accesses*—i.e. pairs of a run-time instance and a reference in the statement—to memory locations. To abstract the effect of every statement instance, we need to make explicit these functions. This is done through the use of induction variables.

After a few definitions and additional restrictions of the program model, we show that induction variables are described by systems of recurrence equations, we prove a fundamental resolution theorem for such systems, and finally we apply this theorem in an algorithm to compute storage mappings.

To simplify the notations of variables and values, we write "`v`" for the name of an existing program variable, and "v" is an abbreviation for "the *value* of variable "`v`".

4.2.1 Induction Variables

We now extend the classical concept of *induction variable*—strongly connected with nested loops—to recursive programs. To simplify the exposition, we suppose that every integer or pointer variable that is local to a procedure or global to the program has a *unique distinctive name*. This allows quick and non-misleading wordings such as "variable `i`", and has no effect on the generality of the approach. Compared to classical works with nests of loops [Wol92], we have a rather original definition of *induction variables*:

- *integer* arguments of a function that are initialized to a constant or to an *integer* induction variable plus constant (e.g. incremented or decremented by a constant), at each procedure call;
- *integer* loop counters that are incremented (or decremented) by a constant at each loop iteration;
- *pointer* arguments that are initialized to a constant or to a possibly dereferenced *pointer* induction variable, at each procedure call;
- *pointer* loop variables that are dereferenced at each loop iteration;

For example, suppose `i`, `j` and `k` are integer variables, `p` and `q` are pointer variables to a `list` structure with a member `next` of type `list*`, and `Compute` is some procedure with two arguments. In the code in Figure 4.4, reference `2*i+j` appears in a non-recursive function call, hence `i`, `j`, `p` and `q` are considered induction variables. On the opposite, `k` is not an induction variable because it retains its last value at the entry of the inner loop.

```
void Compute (int i, list *p) {
  int j, k;
  list *q;
  ...
  for (q=p, k=0; q!=NULL; q=q->next)
    for (j=0; j<100; j+=2, k++)
      // recursive call
      Compute (j+1, q);
  ...
  printf ("%d", 2*i+j);
}
```

Figure 4.4. First example of induction variables

As a kind of syntactic sugar to increase the versatility of induction variables, some cases of *direct assignments* to induction variables are allowed—i.e. induction variable updates outside of loop iterations and procedure calls. Regarding initialization and increment/decrement/dereference, the rules are the same than for a procedure call, but there are two additional restrictions. These restrictions are those of the *code motion* [KRS94, Gup98] and *symbolic execution* techniques [Muc97] used to move each direct assignment to some loop/procedure block surrounding it. After such a transformation, direct assignments can be interpreted as "executed at the entry of that block", the name of the statement being replaced by the actual name of the block.

Of course, symbolic execution techniques cannot convert all cases of direct assignations into legal induction variable updates, as shown by the following examples. Considering the program in Figure 4.5.a, `i` is an induction variable because the `while` loop can be converted into a `for` loop on `i`, but `j` is not an induction variable since it is not initialized at the entry of the inner `for` loop. Considering the other program in Figure 4.5.b, variable `i` is not an induction variable because s is guarded by a conditional.

```
    int i=0, j=0, k, A[200];
    while (i<10) {
      for (k=0; k<10; k++) {
        j = j + 2;
        ···;
      }
r     A[i] = A[i] + A[j];
s     i = i + 1;
    }
```

Figure 4.5.a. Second example

```
    int i, A[10, 10];
    for (i=0, j=0; i<10; i++) {
      if (···)
s       i = i + 2;
r     A[i, j] = ···;
    }
```

Figure 4.5.b. Third example

Figure 4.5. More examples of induction variables

Additional restrictions to the program model In comparison with the general program model presented in Section 2.2, our analysis requires a few additional hypotheses:

- every data structure subject to dependence or reaching definition analysis must be declared *global* (notice that local variables can be made global using explicit memory allocations and stacks);
- every array subscript must be an affine function of integer *induction variables* (not any integer variable) and symbolic constants;
- every tree access must dereference a pointer *induction variable* (not any pointer variable) or a constant.

4.2.2 Building Recurrence Equations on Induction Variables

Describing conflicts between memory accesses is at the core of dependence analysis. We must be able to associate memory locations to memory references in statement instances (i.e. `A[i]`, `*p`, etc.) by means of *storage mappings*. This analysis is done independently on each data-structure. For each induction veriable, we thus need a function mapping a control word to the associated value of the induction variable. In addition, the next definition introduces a notation for the relation between control words and induction variable values.

Definition 4.1 (value of induction variables) Let α be a program statement or block, and w be an instance of α. The *value of variable* `i` *at instance* w is defined as the value of `i` immediately *after* executing (resp. entering) instance w of statement (resp. block) α. This value is denoted by $[\![\mathtt{i}]\!](w)$.

For a program statement α and an induction variable `i`, we call $[\![\mathtt{i}, \alpha]\!]$ the set of all pairs $(u\alpha, i)$ such that $[\![\mathtt{i}]\!](u\alpha) = i$, for all instances $u\alpha$ of α.

We consider pairs of elements in monoids, and to be consistent with the usual notation for rational sets and relations, a pair (x, y) will be denoted by $(x|y)$.

In general, the value of a variable at a given control word depends on the execution. Indeed, an execution trace keeps all the information about variable updates, but not a control word. However, due to our program model restrictions, *induction variables* are completely defined by *control words*:

Lemma 4.1 Let `i` be an induction variable and u a statement instance. If the value $[\![\mathtt{i}]\!](u)$ depends on the effect of an instance v—i.e. the value depends on whether v executes or not—then v is a *prefix* of u.

Proof: Simply observe that only loop entries, loop iterations and procedure calls may modify an induction variable, and that loop entries are associated with initialisations which "kill" the effect of all preceeding iterations (associated with non-prefix control words). ■

For two program executions $e, e' \in \mathbf{E}$, the consequence of Lemma 4.1 is that storage mappings f_e and $f_{e'}$ coincides on $\mathbf{A}_e \cap \mathbf{A}_{e'}$. This strong property allows to extend the computation of a storage mapping f_e to the whole set $\mathbf{A}$ of *possible* accesses. With this extension, all storage mappings for different executions of a program coincides. We will thus consider in the following a storage mapping f *independent on the execution.*

The following result states that induction variable are described by *recurrence equations*:

Lemma 4.2 Let $(M_{\text{DATA}}, \bullet)$ be the monoid abstraction of the considered data structure. Consider a statement α and an induction variable `i`. The effect of statement α on the value if `i` is captured by one of the following equations:

$$\text{either} \quad \exists \beta \in M_{\text{DATA}}, \mathtt{j} \in \text{INDUC}: \quad \forall u\alpha \in L_{\text{CTRL}} : [\![\mathtt{i}]\!](u\alpha) = [\![\mathtt{j}]\!](u) \bullet \beta \tag{4.1}$$

$$\text{or} \quad \exists \beta \in M_{\text{DATA}}: \quad \forall u\alpha \in L_{\text{CTRL}} : [\![\mathtt{i}]\!](u\alpha) = \beta \tag{4.2}$$

where INDUC is the set of all induction variables in the program, including `i`.

Proof: Consider an edge α in the control automaton. Due to our syntactical restrictions, edge α corresponds to a statement in the program text that can modify `i` in only two ways:

- either there exist an induction variable `j` whose value is $j \in M_{\text{DATA}}$ just before executing instance $u\alpha$ of statement α and a constant $\beta \in M_{\text{DATA}}$ such that the value of `i` after executing instance $u\alpha$ is $j \bullet \beta$—translation from a possibly identical variable;
- or there exist a constant $\beta \in M_{\text{DATA}}$ such that the value of `i` after executing instance $u\alpha$ is β—initialization.

■

Notice that, when accessing arrays, we allow general affine subscripts and not only induction variables. Therefore we also build equations on affine functions `a(i,j,···)` of the induction variables. For example, if `a(i,j,k) = 2*i+j-k` then we have to build equations on $[\![2*\mathtt{i}+\mathtt{j}-\mathtt{k}]\!](u)$ knowing that $[\![2*\mathtt{i}+\mathtt{j}-\mathtt{k}]\!](u) = 2[\![\mathtt{i}]\!](u) + [\![\mathtt{j}]\!](u) - [\![\mathtt{k}]\!](u)$.[2]

[2]We have indeed to generate new equations, since computing $[\![2*\mathtt{i}+\mathtt{j}-\mathtt{k}]\!](u)$ from $[\![\mathtt{i}]\!](u)$, $[\![\mathtt{j}]\!](u)$ and $[\![\mathtt{k}]\!](u)$ is not possible in general: variables `i`, `j` and `k` may have different scopes.

To build systems of recurrent equations automatically, we need two additional notations:

UNDEFINED is a polymorphic value for induction variables, $[\![\mathtt{i}]\!](w) = \textsc{Undefined}$ means that variable `i` has an *undefined value* at instance w; it may also be the case that `i` is not visible at instance w;

$\textsc{Arg}(\mathtt{proc}, num)$ stands for the num^{th} *actual* argument of procedure `proc`.

Algorithm RECURRENCE-BUILD applies Lemma 4.2 in turn for each statement in the program.

RECURRENCE-BUILD (*program*)
program: an intermediate representation of the program
returns a list of recurrence equations

$sys \leftarrow \varnothing$
for each statement α **in** *program*
do for each induction variable `i` **in** α
  do switch
    case $\alpha =$ `for (i=`$init$`; ···; ···)` : *// loop entry*
      $sys \leftarrow sys \cup \{\forall u\alpha \in L_{\text{CTRL}} : [\![\mathtt{i}]\!](u\alpha) = init\}$
    case $\alpha =$ `for (···; ···; i=i+`inc`)` : *// loop iteration*
      $sys \leftarrow sys \cup \{\forall u\alpha \in L_{\text{CTRL}} : [\![\mathtt{i}]\!](u\alpha) = [\![\mathtt{i}]\!](u) \bullet inc\}$
    case $\alpha =$ `for (···; ···; i=i->`inc`)` : *// loop iteration*
      $sys \leftarrow sys \cup \{\forall u\alpha \in L_{\text{CTRL}} : [\![\mathtt{i}]\!](u\alpha) = [\![\mathtt{i}]\!](u) \bullet inc\}$
    case $\alpha = \mathtt{proc}\ (\underbrace{\cdots}_{m-1}, \mathtt{var}, \cdots)$:
      $sys \leftarrow sys \cup \{\forall u\alpha \in L_{\text{CTRL}} : [\![\textsc{Arg}(\mathtt{proc}, m)]\!](u\alpha) = [\![\mathtt{var}]\!](u)\}$
    case $\alpha = \mathtt{proc}\ (\underbrace{\cdots}_{m-1}, \mathtt{var}+cst, \cdots)$:
      $sys \leftarrow sys \cup \{\forall u\alpha \in L_{\text{CTRL}} : [\![\textsc{Arg}(\mathtt{proc}, m)]\!](u\alpha) = [\![\mathtt{var}]\!](u) \bullet cst\}$
    case $\alpha = \mathtt{proc}\ (\underbrace{\cdots}_{m-1}, \mathtt{var}\text{->}cst, \cdots)$:
      $sys \leftarrow sys \cup \{\forall u\alpha \in L_{\text{CTRL}} : [\![\textsc{Arg}(\mathtt{proc}, m)]\!](u\alpha) = [\![\mathtt{var}]\!](u) \bullet cst\}$
    case $\alpha = \mathtt{proc}\ (\underbrace{\cdots}_{m-1}, cst, \cdots)$:
      $sys \leftarrow sys \cup \{\forall u\alpha \in L_{\text{CTRL}} : [\![\textsc{Arg}(\mathtt{proc}, m)]\!](u\alpha) = cst\}$
    case default :
      $sys \leftarrow sys \cup \{\forall u\alpha \in L_{\text{CTRL}} : [\![\mathtt{i}]\!](u\alpha) = [\![\mathtt{i}]\!](u)\}$
  for each procedure p declared $\mathtt{proc}\ (\mathtt{type}_1\ \mathtt{arg}_1, \cdots, \mathtt{type}_n\ \mathtt{arg}_n)$ **in** α
  do for $m \leftarrow 1$ **to** n
    do $sys \leftarrow sys \cup \{\forall up \in L_{\text{CTRL}} : [\![\mathtt{arg}_m]\!](up) = [\![\textsc{Arg}(\mathtt{proc}, m)]\!](u)\}$
return *sys*

Now, suppose that there exist a statement α, two induction variables `i` and `j`, and a constant $\beta \in M_{\text{DATA}}$ such that $[\![\mathtt{i}]\!](u\alpha) = [\![\mathtt{j}]\!](u) \bullet \beta$ is an equation generated by Lemma 4.2. Transposed to $[\![\mathtt{i}, \alpha]\!]$—the set of all pairs $(u\alpha | [\![\mathtt{i}]\!](u\alpha))$—it says that

$$(u|j) \in [\![\mathtt{j}, \alpha']\!] \implies (u\alpha | j \bullet \beta) \in [\![\mathtt{i}, \alpha]\!],$$

for all statements α' that may precede α in a valid control word u. Second, suppose that there exist a statement α, an induction variables `i`, and a constant $\beta \in M_{\text{DATA}}$ such that

$\llbracket \texttt{i} \rrbracket(u\alpha) = \beta$ is an equation generated by Lemma 4.2. Transposed to $\llbracket \texttt{i}, \alpha \rrbracket$, it says that

$$(u|i) \in \llbracket \texttt{i}, \alpha' \rrbracket \implies (u\alpha|\beta) \in \llbracket \texttt{i}, \alpha \rrbracket,$$

for all statements α' that may precede α in a valid control word u. These two observations allow to build a new system involving equations on sets $\llbracket \texttt{i}, \alpha \rrbracket$ from the result of Recurrence-Build. Algorithm to achieve this is called Recurrence-Rewrite: the two conditionals in Recurrence-Rewrite are associated with $u = \varepsilon$, i.e. with recurrence equations of the form $\llbracket \texttt{i} \rrbracket(\alpha) = \llbracket \texttt{j} \rrbracket(\varepsilon)\beta$ ($\llbracket \texttt{j} \rrbracket(\varepsilon)$ is an undefined value) or $\llbracket \texttt{i} \rrbracket(\alpha) = \beta$, and the two loops on α' consider predecessors of α.

Recurrence-Rewrite $(program, system)$
    program: an intermediate representation of the program
    system: a system of recurrence equations produced by Recurrence-Build
    returns a rewritten system of recurrence equations

```
L_CTRL ← language of control words of program
new ← ∅
for each equation ∀uα ∈ L_CTRL : [[i]](uα) = [[j]](u) • β in system
do if α ∈ L_CTRL
      then new ← new ∪ {(α|j • β) ∈ [[i, α]]}
   for each α' such that (Σ*_CTRL α'α ∩ L_CTRL) ≠ ∅
   do new ← new ∪ {∀uα ∈ L_CTRL : (u|j) ∈ [[j, α']] ⇒ (uα|j • β) ∈ [[i, α]]}
for each equation ∀uα ∈ L_CTRL : [[i]](uα) = β in system
do if α ∈ L_CTRL
      then new ← new ∪ {(α|β) ∈ [[i, α]]}
   for each α' such that (Σ*_CTRL α'α ∩ L_CTRL) ≠ ∅
   do new ← new ∪ {∀uα ∈ L_CTRL : (u|i) ∈ [[i, α']] ⇒ (uα|β) ∈ [[i, α]]}
return new
```

Algorithms Recurrence-Build and Recurrence-Rewrite are now applied to procedure `Queens`. There are three induction variables, `i`, `j` and `k`; but variable `i` is not useful for computing storage mapping functions. We get the following equations:

$$\begin{aligned}
\text{From main call } F\text{:} \quad & \llbracket \textsc{Arg}(\texttt{Queens}, 2) \rrbracket(F) = 0 \\
\text{From procedure } P\text{:} \quad & \forall uP \in L_{\text{CTRL}} : \llbracket \texttt{k} \rrbracket(uP) = \llbracket \textsc{Arg}(\texttt{Queens}, 2) \rrbracket(u) \\
\text{From recursive call } Q\text{:} \quad & \forall uQ \in L_{\text{CTRL}} : \llbracket \textsc{Arg}(\texttt{Queens}, 2) \rrbracket(uQ) = \llbracket \texttt{k} \rrbracket(u) + 1 \\
\text{From entry } B \text{ of loop } B/\textsc{b}/b\text{:} \quad & \forall uB \in L_{\text{CTRL}} : \llbracket \texttt{j} \rrbracket(uB) = 0 \\
\text{From iteration } b \text{ of loop } B/\textsc{b}/b\text{:} \quad & \forall ub \in L_{\text{CTRL}} : \llbracket \texttt{j} \rrbracket(ub) = \llbracket \texttt{j} \rrbracket(u) + 1
\end{aligned}$$

All other statements let induction variables unchanged or undefined:

$$\begin{aligned}
& \llbracket \texttt{j} \rrbracket(F) = \textsc{Undefined} \\
\forall uP \in L_{\text{CTRL}} : \quad & \llbracket \texttt{j} \rrbracket(uP) = \textsc{Undefined} \\
\forall uI \in L_{\text{CTRL}} : \quad & \llbracket \texttt{j} \rrbracket(uI) = \textsc{Undefined} \\
\forall uA \in L_{\text{CTRL}} : \quad & \llbracket \texttt{j} \rrbracket(uA) = \textsc{Undefined} \\
\forall u\textsc{a} \in L_{\text{CTRL}} : \quad & \llbracket \texttt{j} \rrbracket(u\textsc{a}) = \textsc{Undefined} \\
\forall ua \in L_{\text{CTRL}} : \quad & \llbracket \texttt{j} \rrbracket(ua) = \textsc{Undefined} \\
\forall u\textsc{b} \in L_{\text{CTRL}} : \quad & \llbracket \texttt{j} \rrbracket(u\textsc{b}) = \llbracket \texttt{j} \rrbracket(u) \\
\forall ur \in L_{\text{CTRL}} : \quad & \llbracket \texttt{j} \rrbracket(ur) = \llbracket \texttt{j} \rrbracket(u) \\
\forall uJ \in L_{\text{CTRL}} : \quad & \llbracket \texttt{j} \rrbracket(uJ) = \llbracket \texttt{j} \rrbracket(u) \\
\forall uQ \in L_{\text{CTRL}} : \quad & \llbracket \texttt{j} \rrbracket(uQ) = \textsc{Undefined} \\
\forall us \in L_{\text{CTRL}} : \quad & \llbracket \texttt{j} \rrbracket(us) = \textsc{Undefined}
\end{aligned}$$

$$
\begin{array}{rl}
 & [\![\mathtt{k}]\!](F) = \textsc{Undefined} \\
\forall uI \in L_{\textsc{ctrl}}: & [\![\mathtt{k}]\!](uI) = [\![\mathtt{k}]\!](u) \\
\forall uA \in L_{\textsc{ctrl}}: & [\![\mathtt{k}]\!](uA) = [\![\mathtt{k}]\!](u) \\
\forall u\textsc{a} \in L_{\textsc{ctrl}}: & [\![\mathtt{k}]\!](u\textsc{a}) = [\![\mathtt{k}]\!](u) \\
\forall ua \in L_{\textsc{ctrl}}: & [\![\mathtt{k}]\!](ua) = [\![\mathtt{k}]\!](u) \\
\forall uB \in L_{\textsc{ctrl}}: & [\![\mathtt{k}]\!](uB) = [\![\mathtt{k}]\!](u) \\
\forall u\textsc{b} \in L_{\textsc{ctrl}}: & [\![\mathtt{k}]\!](u\textsc{b}) = [\![\mathtt{k}]\!](u) \\
\forall ub \in L_{\textsc{ctrl}}: & [\![\mathtt{k}]\!](ub) = [\![\mathtt{k}]\!](u) \\
\forall ur \in L_{\textsc{ctrl}}: & [\![\mathtt{k}]\!](ur) = [\![\mathtt{k}]\!](u) \\
\forall uJ \in L_{\textsc{ctrl}}: & [\![\mathtt{k}]\!](uJ) = [\![\mathtt{k}]\!](u) \\
\forall uQ \in L_{\textsc{ctrl}}: & [\![\mathtt{k}]\!](uQ) = [\![\mathtt{k}]\!](u) \\
\forall us \in L_{\textsc{ctrl}}: & [\![\mathtt{k}]\!](us) = [\![\mathtt{k}]\!](u)
\end{array}
$$

Now, recall that $[\![\mathtt{j},\alpha]\!]$ (resp. $[\![\mathtt{k},\alpha]\!]$) is the set of all pairs $(u\alpha|j)$ (resp. $(u\alpha|k)$) such that $[\![\mathtt{j}]\!](u\alpha) = j$ (resp. $[\![\mathtt{k}]\!](u\alpha) = k$), for all instances $u\alpha$ of a statement α. From equations above, Recurrence-Rewrite yields:

$$
\left\{
\begin{array}{rcl}
 & & (F|\textsc{Undefined}) \in [\![\mathtt{j},F]\!] \\
\forall uP \in L_{\textsc{ctrl}}: (u|j) \in [\![\mathtt{j},F]\!] & \Rightarrow & (uP|\textsc{Undefined}) \in [\![\mathtt{j},P]\!] \\
\forall uP \in L_{\textsc{ctrl}}: (u|j) \in [\![\mathtt{j},Q]\!] & \Rightarrow & (uP|\textsc{Undefined}) \in [\![\mathtt{j},P]\!] \\
\forall uI \in L_{\textsc{ctrl}}: (u|j) \in [\![\mathtt{j},P]\!] & \Rightarrow & (uI|\textsc{Undefined}) \in [\![\mathtt{j},I]\!] \\
\forall uA \in L_{\textsc{ctrl}}: (u|j) \in [\![\mathtt{j},I]\!] & \Rightarrow & (uA|\textsc{Undefined}) \in [\![\mathtt{j},A]\!] \\
\forall u\textsc{a} \in L_{\textsc{ctrl}}: (u|j) \in [\![\mathtt{j},A]\!] & \Rightarrow & (u\textsc{a}|\textsc{Undefined}) \in [\![\mathtt{j},\textsc{a}]\!] \\
\forall u\textsc{a} \in L_{\textsc{ctrl}}: (u|j) \in [\![\mathtt{j},a]\!] & \Rightarrow & (u\textsc{a}|\textsc{Undefined}) \in [\![\mathtt{j},\textsc{a}]\!] \\
\forall ua \in L_{\textsc{ctrl}}: (u|j) \in [\![\mathtt{j},\textsc{a}]\!] & \Rightarrow & (ua|\textsc{Undefined}) \in [\![\mathtt{j},a]\!] \\
\forall uB \in L_{\textsc{ctrl}}: (u|j) \in [\![\mathtt{j},\textsc{a}]\!] & \Rightarrow & (uB|0) \in [\![\mathtt{j},B]\!] \\
\forall u\textsc{b} \in L_{\textsc{ctrl}}: (u|j) \in [\![\mathtt{j},B]\!] & \Rightarrow & (u\textsc{b}|j) \in [\![\mathtt{j},\textsc{b}]\!] \\
\forall u\textsc{b} \in L_{\textsc{ctrl}}: (u|j) \in [\![\mathtt{j},b]\!] & \Rightarrow & (u\textsc{b}|j) \in [\![\mathtt{j},\textsc{b}]\!] \\
\forall ub \in L_{\textsc{ctrl}}: (u|j) \in [\![\mathtt{j},\textsc{b}]\!] & \Rightarrow & (ub|j+1) \in [\![\mathtt{j},b]\!] \\
\forall ur \in L_{\textsc{ctrl}}: (u|j) \in [\![\mathtt{j},\textsc{b}]\!] & \Rightarrow & (ur|j) \in [\![\mathtt{j},r]\!] \\
\forall uJ \in L_{\textsc{ctrl}}: (u|j) \in [\![\mathtt{j},\textsc{a}]\!] & \Rightarrow & (uJ|\textsc{Undefined}) \in [\![\mathtt{j},J]\!] \\
\forall uQ \in L_{\textsc{ctrl}}: (u|j) \in [\![\mathtt{j},J]\!] & \Rightarrow & (uQ|\textsc{Undefined}) \in [\![\mathtt{j},Q]\!] \\
\forall us \in L_{\textsc{ctrl}}: (u|j) \in [\![\mathtt{j},J]\!] & \Rightarrow & (us|\textsc{Undefined}) \in [\![\mathtt{j},s]\!]
\end{array}
\right.
$$

$$
\begin{cases}
\hfill (F|\textsc{Undefined}) \in [\![\mathtt{k}, F]\!] \hfill \\
\forall uP \in L_{\textsc{ctrl}} : (u|x) \in [\![\textsc{Arg}(\mathtt{Queens}, 2), F]\!] \;\Rightarrow\; (uP|x) \in [\![\mathtt{k}, P]\!] \\
\forall uP \in L_{\textsc{ctrl}} : (u|x) \in [\![\textsc{Arg}(\mathtt{Queens}, 2), Q]\!] \;\Rightarrow\; (uP|x) \in [\![\mathtt{k}, P]\!] \\
\forall uI \in L_{\textsc{ctrl}} : (u|k) \in [\![\mathtt{k}, P]\!] \;\Rightarrow\; (uI|k) \in [\![\mathtt{k}, I]\!] \\
\forall uA \in L_{\textsc{ctrl}} : (u|k) \in [\![\mathtt{k}, I]\!] \;\Rightarrow\; (uA|k) \in [\![\mathtt{k}, A]\!] \\
\forall u{\scriptstyle A} \in L_{\textsc{ctrl}} : (u|k) \in [\![\mathtt{k}, A]\!] \;\Rightarrow\; (u{\scriptstyle A}|k) \in [\![\mathtt{k}, {\scriptstyle A}]\!] \\
\forall u{\scriptstyle A} \in L_{\textsc{ctrl}} : (u|k) \in [\![\mathtt{k}, a]\!] \;\Rightarrow\; (u{\scriptstyle A}|k) \in [\![\mathtt{k}, {\scriptstyle A}]\!] \\
\forall ua \in L_{\textsc{ctrl}} : (u|k) \in [\![\mathtt{k}, {\scriptstyle A}]\!] \;\Rightarrow\; (ua|k) \in [\![\mathtt{k}, a]\!] \\
\forall uB \in L_{\textsc{ctrl}} : (u|k) \in [\![\mathtt{k}, {\scriptstyle A}]\!] \;\Rightarrow\; (uB|k) \in [\![\mathtt{k}, B]\!] \\
\forall u{\scriptstyle B} \in L_{\textsc{ctrl}} : (u|k) \in [\![\mathtt{k}, B]\!] \;\Rightarrow\; (u{\scriptstyle B}|k) \in [\![\mathtt{k}, {\scriptstyle B}]\!] \\
\forall u{\scriptstyle B} \in L_{\textsc{ctrl}} : (u|k) \in [\![\mathtt{k}, b]\!] \;\Rightarrow\; (u{\scriptstyle B}|k) \in [\![\mathtt{k}, {\scriptstyle B}]\!] \\
\forall ub \in L_{\textsc{ctrl}} : (u|k) \in [\![\mathtt{k}, {\scriptstyle B}]\!] \;\Rightarrow\; (ub|k) \in [\![\mathtt{k}, b]\!] \\
\forall ur \in L_{\textsc{ctrl}} : (u|k) \in [\![\mathtt{k}, {\scriptstyle B}]\!] \;\Rightarrow\; (ur|k) \in [\![\mathtt{k}, r]\!] \\
\forall uJ \in L_{\textsc{ctrl}} : (u|k) \in [\![\mathtt{k}, {\scriptstyle A}]\!] \;\Rightarrow\; (uJ|k) \in [\![\mathtt{k}, J]\!] \\
\forall uQ \in L_{\textsc{ctrl}} : (u|k) \in [\![\mathtt{k}, J]\!] \;\Rightarrow\; (uQ|k) \in [\![\mathtt{k}, Q]\!] \\
\forall us \in L_{\textsc{ctrl}} : (u|k) \in [\![\mathtt{k}, J]\!] \;\Rightarrow\; (us|k) \in [\![\mathtt{k}, s]\!] \\
\hfill (F|0) \in [\![\textsc{Arg}(\mathtt{Queens}, 2), F]\!] \hfill \\
\forall uQ \in L_{\textsc{ctrl}} : (u|k) \in [\![\mathtt{k}, J]\!] \;\Rightarrow\; (uQ|k+1) \in [\![\textsc{Arg}(\mathtt{Queens}, 2), Q]\!]
\end{cases}
$$

4.2.3 Solving Recurrence Equations on Induction Variables

The following result is at the core of our analysis technique, but it is not limited to this purpose. It will be applied in the next section to the system of equations returned by RECURRENCE-REWRITE.

Lemma 4.3 Consider two monoids L and M with respective binary operations $*$ and $\star$. Let R be a subset of $L \times M$ defined by a system of equations of the form

$$(E_1) \qquad \forall l \in L, m_1 \in M : \quad (l|m_1) \in R_1 \implies (l * \alpha_1 | m_1 \star \beta_1) \in R$$

and

$$(E_2) \qquad \forall l \in L, m_2 \in M : (l|m_2) \in R_2 \implies (l * \alpha_2 | \beta_2) \in R,$$

where $R_1 \subset L \times M$ and $R_2 \subset L \times M$ are some set *variables* constrained in the system (possibly equal to R), α_1, α_2 are *constants* in L and β_1, β_2 are *constants* in M. Then, R is a rational set.

Proof: Our first task is to convert these expressions on unstructured elements of L and M, into expressions in the monoid $L \times M$. Then our second task is to derive *set* expressions in $L \times M$, of the form *set* $\cdot$ *constant* $\subseteq$ *set* or *constant* $\subseteq$ *set* (the induced operation is denoted by "$\cdot$"). Indeed, the right-hand-side of (E_1) can be written

$$(l|m_1) \cdot (\alpha_1|\beta_1) \in R.$$

Thus, (E_1) gives

$$R_1 \cdot (\alpha_1|\beta_1) \subseteq R.$$

The right-hand-side of (E_2) can also been written

$$(l|\varepsilon) \cdot (\alpha_2|\beta_2) \in R$$

but $(l|\varepsilon)$ is neither a variable nor a constant of $L \times M$.

To overcome this difficulty, we call R^ε the set of all pairs $(l|\varepsilon)$ such that $\exists m \in M$: $(l|m) \in R$. It is clear that R^ε satisfies the same equations as R with all right pair members replaced by ε. Now, (E_2) yields two equations:

$$R_2^\varepsilon \cdot (\alpha_2|\varepsilon) \subseteq R^\varepsilon \qquad \text{and} \qquad R^\varepsilon \cdot (\varepsilon|\beta_2) \subseteq R.$$

At last, if the only equations on R are (E_1) and (E_2), we have

$$\begin{aligned} R^\varepsilon &= R_1 \cdot (\alpha_1|\varepsilon) + R_2^\varepsilon \cdot (\alpha_2|\varepsilon) \\ R &= R_1 \cdot (\alpha_1|\beta_1) + R^\varepsilon \cdot (\varepsilon|\beta_2) \end{aligned}$$

More generally, applying this process to R_1, R_2 and to every subset of $L \times M$ described in the system, we get a new system of *regular equations* defining R. It is well known that such equations define a rational subset of $L \times M$. ■

Thanks to classical list operations INSERT, DELETE and MEMBER (systems are encoded as lists of equations), and to string operation CONCAT (equations are encoded as strings), algorithm RECURRENCE-SOLVE gives an automatic way to solve systems of equations of the form (E_1) or (E_2).

RECURRENCE-SOLVE (*system*)
system: a list of recurrence equations of the form (E_1) and (E_2)
returns a list of regular expressions
sets $\leftarrow \varnothing$
for each implication " $(l|m) \in A \Rightarrow (l * \alpha|m \star \beta) \in B$ " **in** *system*
do INSERT $(sets, \{A \cdot (\alpha|\beta) \subseteq B\})$
  INSERT $(sets, \{A^\varepsilon \cdot (\alpha|\varepsilon) \subseteq B^\varepsilon\})$
for each implication " $(l|m) \in A \Rightarrow (l * \alpha|\beta) \in B$ " **in** *system*
do INSERT $(sets, \{B^\varepsilon \cdot (\varepsilon|\beta) \subseteq B\})$
  INSERT $(sets, \{A^\varepsilon \cdot (\alpha|\varepsilon) \subseteq B^\varepsilon\})$
variables $\leftarrow \varnothing$
for each inclusion " $A \cdot (x|y) \subseteq B$ " **in** *sets*
do if MEMBER $(variables, B)$
  then *equation* $\leftarrow$ DELETE $(variables, B)$
    INSERT $(variables,$ CONCAT $(equation,$ " $+A \cdot (x|y)$ "$))$
  else INSERT $(variables,$ " $B = A \cdot (x|y)$ "$)$
variables $\leftarrow$ COMPUTE-REGULAR-EXPRESSIONS (*variables*)
return *variables*

Algorithm COMPUTE-REGULAR-EXPRESSIONS solves a system of *regular equations between rational sets*, then returns a list of regular expressions defining these sets. The system is seen as a *regular grammar* and resolution is done through variable substitution—when the variable in left-hand side does not appear in right-hand side—or Kleene star insertion—when it does. Well known heuristics are used to reduce the size of the result, see [HU79] for details.

4.2.4 Computing Storage Mappings

The main result of this section follows: we can solve recurrence equations in Lemma 4.2 to compute the value of induction variables at control words.

Theorem 4.1 The storage mapping f that maps every possible access in **A** to the memory location it accesses is a rational function from Σ^*_{CTRL} to M_{DATA}.

Proof: Since array subscripts are affine functions of integer induction variables, and since tree accesses are given by dereferenced induction pointers, one may generate a system of equations according to Lemma 4.2 (or Recurrence-Build) for any read or write access.

The result is a system of equations on induction variables. Thanks to Recurrence-Rewrite, this system is rewritten in terms of equations on sets of pairs $(u\alpha | [\![\mathtt{i}]\!](u\alpha))$, where $u\alpha$ is a control word and `i` is an iteration variable, describing the value of `i` for any instance of statement α. We thus get a new system which inductively describes subset $[\![\mathtt{i}, \alpha]\!]$ of $\Sigma^*_{\text{CTRL}} \times M_{\text{DATA}}$. Because this system satisfies the hypotheses of Lemma 4.3, we have proven that $[\![\mathtt{i}, \alpha]\!]$ is a rational set of $\Sigma^*_{\text{CTRL}} \times M_{\text{DATA}}$. Now, for a given memory reference in α, we know that pairs $(w|f(w))$—where w is an instance of α—build a rational set. Hence f is a rational transduction from Σ^*_{CTRL} to M_{DATA}. Because f is also a partial function, it is a rational function from Σ^*_{CTRL} to M_{DATA}. ■

The proof is constructive, thanks to Recurrence-Build and Recurrence-Solve, and Compute-Storage-Mappings is the algorithm to automatically compute storage mappings for a recursive program satisfying the hypotheses of Section 4.2.1. The result is a list of *rational transducers*—converted by Compute-Rational-Transducer from regular expressions—realizing the rational storage mappings for each reference in right-hand side.

Compute-Storage-Mappings (*program*)
program: an intermediate representation of the program
returns a list rational transducers realizing storage mappings
system ← Recurrence-Build (*program*)
new ← Recurrence-Rewrite (*program*, *system*)
list ← Recurrence-Solve (*new*)
newlist ← ∅
for each regular expression *reg* **in** *list*
do *newlist* ← *newlist* ∪ Compute-Rational-Transducer (*reg*)
return *newlist*

Let us now apply Compute-Storage-Mappings on program `Queens`. Starting from the result of Recurrence-Rewrite, we apply Recurrence-Solve. Just before calling Compute-Regular-Expressions, we get the following system of regular equations:

$$
\left\{\begin{array}{rcl}
\llbracket \mathtt{j}, F \rrbracket & = & (F|\textsc{Undefined}) \\
\llbracket \mathtt{j}, P \rrbracket & = & \llbracket \mathtt{j}, F \rrbracket \cdot (P|\textsc{Undefined}) + \llbracket \mathtt{j}, Q \rrbracket \cdot (P|\textsc{Undefined}) \\
\llbracket \mathtt{j}, I \rrbracket & = & \llbracket \mathtt{j}, P \rrbracket \cdot (I|\textsc{Undefined}) \\
\llbracket \mathtt{j}, A \rrbracket & = & \llbracket \mathtt{j}, I \rrbracket \cdot (A|\textsc{Undefined}) \\
\llbracket \mathtt{j}, {\scriptstyle A} \rrbracket & = & \llbracket \mathtt{j}, A \rrbracket \cdot ({\scriptstyle A}|\textsc{Undefined}) + \llbracket \mathtt{j}, a \rrbracket \cdot ({\scriptstyle A}|\textsc{Undefined}) \\
\llbracket \mathtt{j}, a \rrbracket & = & \llbracket \mathtt{j}, {\scriptstyle A} \rrbracket \cdot (a|\textsc{Undefined}) \\
\llbracket \mathtt{j}, B \rrbracket & = & \llbracket \mathtt{j}, B \rrbracket^{\varepsilon} \cdot (\varepsilon|0) \\
\llbracket \mathtt{j}, {\scriptstyle B} \rrbracket & = & \llbracket \mathtt{j}, B \rrbracket \cdot ({\scriptstyle B}|0) + \llbracket \mathtt{j}, b \rrbracket \cdot ({\scriptstyle B}|0) \\
\llbracket \mathtt{j}, b \rrbracket & = & \llbracket \mathtt{j}, {\scriptstyle B} \rrbracket \cdot (b|1) \\
\llbracket \mathtt{j}, r \rrbracket & = & \llbracket \mathtt{j}, {\scriptstyle B} \rrbracket \cdot (r|0) \\
\llbracket \mathtt{j}, J \rrbracket & = & \llbracket \mathtt{j}, {\scriptstyle A} \rrbracket \cdot (J|\textsc{Undefined}) \\
\llbracket \mathtt{j}, Q \rrbracket & = & \llbracket \mathtt{j}, J \rrbracket \cdot (Q|\textsc{Undefined}) \\
\llbracket \mathtt{j}, s \rrbracket & = & \llbracket \mathtt{j}, J \rrbracket \cdot (s|\textsc{Undefined}) \\
\llbracket \mathtt{j}, F \rrbracket^{\varepsilon} & = & (F|\varepsilon) \\
\llbracket \mathtt{j}, P \rrbracket^{\varepsilon} & = & \llbracket \mathtt{j}, F \rrbracket^{\varepsilon} \cdot (P|0) + \llbracket \mathtt{j}, Q \rrbracket^{\varepsilon} \cdot (P|0) \\
\llbracket \mathtt{j}, I \rrbracket^{\varepsilon} & = & \llbracket \mathtt{j}, P \rrbracket^{\varepsilon} \cdot (I|0) \\
\llbracket \mathtt{j}, A \rrbracket^{\varepsilon} & = & \llbracket \mathtt{j}, I \rrbracket^{\varepsilon} \cdot (A|0) \\
\llbracket \mathtt{j}, {\scriptstyle A} \rrbracket^{\varepsilon} & = & \llbracket \mathtt{j}, A \rrbracket^{\varepsilon} \cdot ({\scriptstyle A}|0) + \llbracket \mathtt{j}, a \rrbracket^{\varepsilon} \cdot ({\scriptstyle A}|0) \\
\llbracket \mathtt{j}, a \rrbracket^{\varepsilon} & = & \llbracket \mathtt{j}, {\scriptstyle A} \rrbracket^{\varepsilon} \cdot (a|0) \\
\llbracket \mathtt{j}, B \rrbracket^{\varepsilon} & = & \llbracket \mathtt{j}, {\scriptstyle A} \rrbracket^{\varepsilon} \cdot (B|0) \\
\llbracket \mathtt{j}, {\scriptstyle B} \rrbracket^{\varepsilon} & = & \llbracket \mathtt{j}, B \rrbracket^{\varepsilon} \cdot ({\scriptstyle B}|0) + \llbracket \mathtt{j}, b \rrbracket^{\varepsilon} \cdot ({\scriptstyle B}|0) \\
\llbracket \mathtt{j}, b \rrbracket^{\varepsilon} & = & \llbracket \mathtt{j}, {\scriptstyle B} \rrbracket^{\varepsilon} \cdot (b|0) \\
\llbracket \mathtt{j}, J \rrbracket^{\varepsilon} & = & \llbracket \mathtt{j}, {\scriptstyle A} \rrbracket^{\varepsilon} \cdot (J|0) \\
\llbracket \mathtt{j}, Q \rrbracket^{\varepsilon} & = & \llbracket \mathtt{j}, J \rrbracket^{\varepsilon} \cdot (Q|0)
\end{array}\right.
$$

$$
\left\{\begin{array}{rcl}
\llbracket \mathtt{k}, F \rrbracket & = & (F|\textsc{Undefined}) \\
\llbracket \mathtt{k}, P \rrbracket & = & \llbracket \textsc{Arg}(\mathtt{Queens}, 2), F \rrbracket \cdot (P|0) + \llbracket \textsc{Arg}(\mathtt{Queens}, 2), Q \rrbracket \cdot (P|0) \\
\llbracket \mathtt{k}, I \rrbracket & = & \llbracket \mathtt{k}, P \rrbracket \cdot (I|0) \\
\llbracket \mathtt{k}, A \rrbracket & = & \llbracket \mathtt{k}, I \rrbracket \cdot (A|0) \\
\llbracket \mathtt{k}, {\scriptstyle A} \rrbracket & = & \llbracket \mathtt{k}, A \rrbracket \cdot ({\scriptstyle A}|0) + \llbracket \mathtt{k}, a \rrbracket \cdot ({\scriptstyle A}|0) \\
\llbracket \mathtt{k}, a \rrbracket & = & \llbracket \mathtt{k}, {\scriptstyle A} \rrbracket \cdot (a|0) \\
\llbracket \mathtt{k}, B \rrbracket & = & \llbracket \mathtt{k}, {\scriptstyle A} \rrbracket \cdot (B|0) \\
\llbracket \mathtt{k}, {\scriptstyle B} \rrbracket & = & \llbracket \mathtt{k}, B \rrbracket \cdot ({\scriptstyle B}|0) + \llbracket \mathtt{k}, b \rrbracket \cdot ({\scriptstyle B}|0) \\
\llbracket \mathtt{k}, b \rrbracket & = & \llbracket \mathtt{k}, {\scriptstyle B} \rrbracket \cdot (b|0) \\
\llbracket \mathtt{k}, r \rrbracket & = & \llbracket \mathtt{k}, {\scriptstyle B} \rrbracket \cdot (r|0) \\
\llbracket \mathtt{k}, J \rrbracket & = & \llbracket \mathtt{k}, {\scriptstyle A} \rrbracket \cdot (J|0) \\
\llbracket \mathtt{k}, Q \rrbracket & = & \llbracket \mathtt{k}, J \rrbracket \cdot (Q|0) \\
\llbracket \mathtt{k}, s \rrbracket & = & \llbracket \mathtt{k}, J \rrbracket \cdot (s|0) \\
\llbracket \textsc{Arg}(\mathtt{Queens}, 2), F \rrbracket & = & (F|0) \\
\llbracket \textsc{Arg}(\mathtt{Queens}, 2), Q \rrbracket & = & \llbracket \mathtt{k}, J \rrbracket \cdot (Q|1)
\end{array}\right.
$$

These systems—seen as *regular grammars*—can be solved with COMPUTE-REGULAR-EXPRESSIONS, yielding *regular expressions*. These expressions describe rational functions from Σ^*_{CTRL} to $\mathbb{Z}$, but we are only interested in $\llbracket \mathtt{j}, r \rrbracket$ and $\llbracket \mathtt{k}, s \rrbracket$ (accesses to array `A`):

$$
\llbracket \mathtt{j}, r \rrbracket = (FPIA{\scriptstyle A}|0) \cdot \big((JQPIA{\scriptstyle A}|0) + (a{\scriptstyle A}|0)\big)^* \cdot (B{\scriptstyle B}|0) \cdot (b{\scriptstyle B}|1)^* \cdot (r|0) \qquad (4.3)
$$

$$
\llbracket \mathtt{k}, s \rrbracket = (FPIA{\scriptstyle A}|0) \cdot \big((JQPIA{\scriptstyle A}|1) + (a{\scriptstyle A}|0)\big)^* \cdot (Js|0) \qquad (4.4)
$$

Eventually, we have found the storage mapping function for every reference to the array:

$$\begin{aligned}\big\{(ur|f(ur,\mathtt{A[j]}))\big\} &= (FPIA_A|0)\cdot\big((JQPIA_A|0)+(a_A|0)\big)^*\cdot(B_B|0)\cdot(b_B|1)^*\cdot(r|0) \quad (4.5)\\ \big\{(us|f(us,\mathtt{A[k]}))\big\} &= (FPIA_A|0)\cdot\big((JQPIA_A|1)+(a_A|0)\big)^*\cdot(Js|0) \quad (4.6)\end{aligned}$$

4.2.5 Application to Motivating Examples

We have already applied Compute-Storage-Mappings on program `Queens`, and we repeat the process for the two other motivating examples.

Procedure `BST`

Algorithm Compute-Storage-Mappings is now applied to procedure `BST` in Figure 4.2. The only induction variable is `p`:

$$\begin{aligned}\text{From main call } F\text{:} &\quad [\![\textsc{Arg}(\mathtt{BST},1)]\!](F)=\varepsilon\\ \text{From procedure } \mathtt{BST}\text{:} &\quad \forall uP\in L_{\text{CTRL}}:[\![\mathtt{k}]\!](uP)=[\![\textsc{Arg}(\mathtt{BST},1)]\!](u)\\ \text{From first recursive call } L\text{:} &\quad \forall uL\in L_{\text{CTRL}}:[\![\textsc{Arg}(\mathtt{BST},1)]\!](uL)=[\![\mathtt{p}]\!](u)\mathtt{l}\\ \text{From second recursive call } R\text{:} &\quad \forall uR\in L_{\text{CTRL}}:[\![\textsc{Arg}(\mathtt{BST},1)]\!](uR)=[\![\mathtt{p}]\!](u)\mathtt{r}.\end{aligned}$$

All other statements let the induction variable unchanged. Recall that $[\![\mathtt{p},\alpha]\!]$ is the set of all pairs $(u|p)$ such that $[\![\mathtt{p}]\!](u)=p$, for all instances u of a statement α. From equations above, this set satisfy the following regular equations:

$$\left\{\begin{array}{lcl}[\![\mathtt{p},P]\!] &=& (FP|\varepsilon)+[\![\mathtt{p},I_1]\!]\cdot(LP|l)+[\![\mathtt{p},J_1]\!]\cdot(RP|r)\\ [\![\mathtt{p},I_1]\!] &=& [\![\mathtt{p},P]\!]\cdot(I_1|\varepsilon)\\ [\![\mathtt{p},J_1]\!] &=& [\![\mathtt{p},P]\!]\cdot(J_1|\varepsilon)\\ [\![\mathtt{p},I_2]\!] &=& [\![\mathtt{p},I_1]\!]\cdot(I_2|\varepsilon)\\ [\![\mathtt{p},J_2]\!] &=& [\![\mathtt{p},J_1]\!]\cdot(J_2|\varepsilon)\\ [\![\mathtt{p},a]\!] &=& [\![\mathtt{p},I_2]\!]\cdot(a|\varepsilon)\\ [\![\mathtt{p},b]\!] &=& [\![\mathtt{p},I_2]\!]\cdot(b|\varepsilon)\\ [\![\mathtt{p},c]\!] &=& [\![\mathtt{p},I_2]\!]\cdot(c|\varepsilon)\\ [\![\mathtt{p},d]\!] &=& [\![\mathtt{p},J_2]\!]\cdot(d|\varepsilon)\\ [\![\mathtt{p},e]\!] &=& [\![\mathtt{p},J_2]\!]\cdot(e|\varepsilon)\\ [\![\mathtt{p},f]\!] &=& [\![\mathtt{p},J_2]\!]\cdot(f|\varepsilon)\end{array}\right.$$

This system describes rational functions from Σ^*_{CTRL} to $\mathbb{Z}$, but we are only interested in $[\![\mathtt{p},\alpha]\!]$ for $\alpha\in\{I_2,a,b,c,J_2,d,e,f\}$ (accesses to node values):

$$\begin{aligned}\forall\alpha\in\{I_2,a,b,c\}:[\![\mathtt{p},\alpha]\!] &= (FP|\varepsilon)\cdot\big((I_1LP|l)+(J_1RP|r)\big)^*\cdot(I_1I_2\alpha|\varepsilon) \quad (4.7)\\ \forall\alpha\in\{J_2,d,e,f\}:[\![\mathtt{p},\alpha]\!] &= (FP|\varepsilon)\cdot\big((I_1LP|l)+(J_1RP|r)\big)^*\cdot(J_1J_2\alpha|\varepsilon) \quad (4.8)\end{aligned}$$

Eventually, we can compute the storage mapping function for every reference to the tree:

$$\begin{aligned}&\forall\alpha\in\{I_2,a,b\}:\\ \big\{(u\alpha|f(u\alpha,\mathtt{p\text{->}value}))\big\} &= (FP|\varepsilon)\cdot\big((I_1LP|l)+(J_1RP|r)\big)^*\cdot(I_1I_2\alpha|\varepsilon) \quad (4.9)\\ &\forall\alpha\in\{I_2,b,c\}:\\ \big\{(u\alpha|f(u\alpha,\mathtt{p\text{->}l\text{->}value}))\big\} &= (FP|\varepsilon)\cdot\big((I_1LP|l)+(J_1RP|r)\big)^*\cdot(I_1I_2\alpha|l) \quad (4.10)\\ &\forall\alpha\in\{J_2,d,e\}:\\ \big\{(u\alpha|f(u\alpha,\mathtt{p\text{->}value}))\big\} &= (FP|\varepsilon)\cdot\big((I_1LP|l)+(J_1RP|r)\big)^*\cdot(J_1J_2\alpha|\varepsilon) \quad (4.11)\\ &\forall\alpha\in\{J_2,e,f\}:\\ \big\{(u\alpha|f(u\alpha,\mathtt{p\text{->}r\text{->}value}))\big\} &= (FP|\varepsilon)\cdot\big((I_1LP|l)+(J_1RP|r)\big)^*\cdot(J_1J_2\alpha|r) \quad (4.12)\end{aligned}$$

Function `Count`

Algorithm COMPUTE-STORAGE-MAPPINGS is now applied to procedure `Count` in Figure 4.3. Variable `p` is a tree index and variable `i` is an integer index. Indeed, the `inode` structure is neither a tree nor an array: nodes are named in the language $L_{\text{DATA}} = (\mathbb{Z}\mathtt{n}^*)\mathbb{Z}$. Thus, the effective induction variable should combine both p and i and be interpreted in L_{DATA}, with binary operation $\bullet$ defined in Section 2.3.3. But no such variable appears in the program... The reason is that the code is written in C, in which the `inode` structure cannot be referenced through a uniform "cursor"—like a tree pointer or array subscript.

```
P      int Count (inode &p) {
I        if (p->terminal)
a          return p->length;
E        else {
b          c = 0;
L/L/l      for (int i=0, inode &q=p->n; i<p->length; i++, q=q->1)
c            c += Count (q);
d          return c;
         }
       }

       main () {
F        Count (file);
       }
```

Figure 4.6. Procedure `Count` and control automaton

This would become possible in a higher-level language: we have rewritten the program in a C++-like syntax in Figure 4.6. Now, p is a C++ reference and not a pointer, and operation `->` has been redefined to emulate array accesses.[3] References `p` and `q` are the two induction variables:

$$\begin{aligned}
\text{From main call } F\text{:}\quad & [\![\textsc{Arg}(\mathtt{Count}, 1)]\!](F) = \varepsilon \\
\text{From procedure } P\text{:}\quad & \forall uP \in L_{\text{CTRL}} : [\![\mathtt{p}]\!](uP) = [\![\textsc{Arg}(\mathtt{Count}, 1)]\!](u) \\
\text{From recursive call } c\text{:}\quad & \forall uc \in L_{\text{CTRL}} : [\![\textsc{Arg}(\mathtt{Count}, 1)]\!](uc) = [\![\mathtt{q}]\!](u) \\
\text{From entry } L \text{ of loop } L/\textsc{l}/l\text{:}\quad & \forall uL \in L_{\text{CTRL}} : [\![\mathtt{q}]\!](uL) = [\![\mathtt{p}]\!](u) \bullet \mathtt{n} \\
\text{From iteration } l \text{ of loop } L/\textsc{l}/l\text{:}\quad & \forall ul \in L_{\text{CTRL}} : [\![\mathtt{q}]\!](ul) = [\![\mathtt{q}]\!](u) \bullet 1
\end{aligned}$$

All other statements let induction variables unchanged or undefined. Recall that $[\![\mathtt{p}, \alpha]\!]$ (resp. $[\![\mathtt{q}, \alpha]\!]$) is the set of all pairs $(u|p)$ (resp. $(u|q)$) such that $[\![\mathtt{p}]\!](u) = p$ (resp. $[\![\mathtt{q}]\!](u) = q$), for all instances u of a statement α. From equations above, these sets satisfy

[3] Yes, C++ is both high-level and dirty!

the following regular equations:

$$\left\{\begin{array}{lcl}
\llbracket \mathtt{p}, P \rrbracket & = & (FP|\varepsilon) + \llbracket \mathtt{q}, \text{\textsc{l}} \rrbracket \cdot (cP|\varepsilon) \\
\llbracket \mathtt{p}, I \rrbracket & = & \llbracket \mathtt{p}, P \rrbracket \cdot (I|\varepsilon) \\
\llbracket \mathtt{p}, E \rrbracket & = & \llbracket \mathtt{p}, P \rrbracket \cdot (E|\varepsilon) \\
\llbracket \mathtt{p}, a \rrbracket & = & \llbracket \mathtt{p}, I \rrbracket \cdot (a|\varepsilon) \\
\llbracket \mathtt{p}, b \rrbracket & = & \llbracket \mathtt{p}, E \rrbracket \cdot (b|\varepsilon) \\
\llbracket \mathtt{p}, L \rrbracket & = & \llbracket \mathtt{p}, E \rrbracket \cdot (L|\varepsilon) \\
\llbracket \mathtt{p}, \text{\textsc{l}} \rrbracket & = & \llbracket \mathtt{p}, L \rrbracket \cdot (\text{\textsc{l}}|\varepsilon) + \llbracket \mathtt{p}, \text{\textsc{l}} \rrbracket \cdot (l\text{\textsc{l}}|\varepsilon) \\
\llbracket \mathtt{p}, d \rrbracket & = & \llbracket \mathtt{p}, E \rrbracket \cdot (d|\varepsilon) \\
\llbracket \mathtt{q}, P \rrbracket & = & (F|\text{\textsc{Undefined}}) + \llbracket \mathtt{q}, \text{\textsc{l}} \rrbracket \cdot (cP|\text{\textsc{Undefined}}) \\
\llbracket \mathtt{q}, I \rrbracket & = & \llbracket \mathtt{q}, P \rrbracket \cdot (I|\text{\textsc{Undefined}}) \\
\llbracket \mathtt{q}, E \rrbracket & = & \llbracket \mathtt{q}, P \rrbracket \cdot (E|\text{\textsc{Undefined}}) \\
\llbracket \mathtt{q}, a \rrbracket & = & \llbracket \mathtt{q}, I \rrbracket \cdot (a|\text{\textsc{Undefined}}) \\
\llbracket \mathtt{q}, b \rrbracket & = & \llbracket \mathtt{q}, E \rrbracket \cdot (b|\text{\textsc{Undefined}}) \\
\llbracket \mathtt{q}, L \rrbracket & = & \llbracket \mathtt{p}, E \rrbracket \cdot (L|\mathtt{n}) \\
\llbracket \mathtt{q}, \text{\textsc{l}} \rrbracket & = & \llbracket \mathtt{q}, L \rrbracket \cdot (\text{\textsc{l}}|0) + \llbracket \mathtt{q}, \text{\textsc{l}} \rrbracket \cdot (l\text{\textsc{l}}|1) \\
\llbracket \mathtt{q}, d \rrbracket & = & \llbracket \mathtt{q}, E \rrbracket \cdot (d|\text{\textsc{Undefined}})
\end{array}\right.$$

These systems describe rational functions from Σ^*_{CTRL} to $(\mathbb{Z}\mathtt{n}^*)\mathbb{Z}$, but we are only interested in $\llbracket \mathtt{p}, I \rrbracket$, $\llbracket \mathtt{p}, a \rrbracket$ and $\llbracket \mathtt{p}, L \rrbracket$ (accesses to `inode` values):

$$\begin{aligned}
\llbracket \mathtt{p}, I \rrbracket &= \big\{(uI|f(uI, \mathtt{p\text{->}terminal}))\big\} \\
&= (FP|\varepsilon) \cdot \big((EL\text{\textsc{l}}|\mathtt{n}) \cdot (l\text{\textsc{l}}|1)^* \cdot (cP|\varepsilon)\big)^* \cdot (I|\varepsilon) \qquad (4.13) \\
\llbracket \mathtt{p}, a \rrbracket &= \big\{(ua|f(ua, \mathtt{p\text{->}length}))\big\} \\
&= (FP|\varepsilon) \cdot \big((EL\text{\textsc{l}}|\mathtt{n}) \cdot (l\text{\textsc{l}}|1)^* \cdot (cP|\varepsilon)\big)^* \cdot (Ia|\varepsilon) \qquad (4.14) \\
\llbracket \mathtt{p}, L \rrbracket &= \big\{(uL\text{\textsc{l}}|f(uL\text{\textsc{l}}, \mathtt{p\text{->}length}))\big\} \\
&= (F|\varepsilon) \cdot \big((EL\text{\textsc{l}}|\mathtt{n}) \cdot (l\text{\textsc{l}}|1)^* \cdot (cP|\varepsilon)\big)^* \cdot (EL|\varepsilon) \qquad (4.15)
\end{aligned}$$

4.3 Dependence and Reaching Definition Analysis

When all program model restrictions are satisfied, we have shown in the previous section that storage mappings are *rational transductions*. Based on this result, we will now present a general dependence and reaching definition analysis scheme for recursive programs. Both classical results and recent contributions to formal languages theory will be useful, definitions and details can be found in Chapter 3.

This section tackles the general dependence and reaching definition analysis problem in our program model. See Sections 4.4 (trees), 4.5 (arrays) and 4.6 (nested trees and arrays) for technical questions depending on the data structure context.

4.3.1 Building the Conflict Transducer

In Section 2.4.1, we have seen that analysis of conflicting accesses is one of the first problems arising when computing dependence relations. We thus present a general computation scheme for the *conflict relation*, but technical issues and precise study is left for the next sections.

We consider a program whose set of statement labels is Σ_{CTRL}. Let $L_{\text{CTRL}} \subset \Sigma^*_{\text{CTRL}}$ be the rational language of control words. Let M_{DATA} be the monoid abstraction for a given data structure D used in the program, and $L_{\text{DATA}} \subset M_{\text{DATA}}$ be the rational language of valid data structure elements.

Now because f is used instead of f_e (it is independent on the execution), the exact conflict relation κ_e is defined by

$$\forall e \in \mathbf{E}, \forall u, v \in L_{\text{CTRL}} : \quad u\, \kappa_e\, v \iff (u, v \in \mathbf{A}_e) \wedge f(u) = f(v),$$

which is equivalent to

$$\forall e \in \mathbf{E}, \forall u, v \in L_{\text{CTRL}} : \quad u\, \kappa_e\, v \iff (u, v \in \mathbf{A}_e) \wedge v \in f^{-1}(f(u)).$$

Because f is a rational transduction from Σ^*_{CTRL} to M_{DATA}, f^{-1} is a rational transduction from M_{DATA} to Σ^*_{CTRL}, and M_{DATA} is either a free monoid, or a free commutative monoid, or a free partially commutative monoid, we know from Theorems 3.5, 3.27 and 3.28 that $f^{-1} \circ f$ is either a rational or a multi-counter counter transduction. The result will thus be exact in almost all cases: only multi-counter transductions must be approximated by one-counter transductions.

We cannot compute an exact relation κ_e, since $\mathbf{A}_e$ depends on the execution e. Moreover, guards of conditionals and loop bounds are not taken into account for the moment, and the only approximation of $\mathbf{A}_e$ we can use is the full language $\mathbf{A} = L_{\text{CTRL}}$ of control words. Eventually, the approximate conflict relation we compute is the following:

$$\forall u, v \in L_{\text{CTRL}} : \quad u\, \kappa\, v \overset{\text{def}}{\iff} v \in f^{-1}(f(u)). \tag{4.16}$$

In all cases, we get a transducer realization (rational or one-counter) of transduction κ. This realization is often unapproximate on pairs of control words which are effectively executed.

One may immediately notice that testing for emptiness of κ is equivalent to testing whether two *pointers are aliased* [Deu94, Ste96], and emptimess is decidable for rational and algebraic transductions (see Chapter 3). This is an important application of our analysis, considering the fact that κ is often unapproximate in practice.

Notice also that this computation of κ does not require access functions to be rational *functions*: if a rational *transduction* approximation of f was available, one could still compute relation κ using the same techniques. However, a general approximation scheme for function f has not been designed, and further study is left for future work.

4.3.2 Building the Dependence Transducer

To build the dependence transducer, we need first to restrict relation κ_e to pairs of write accesses or read and write accesses, and then to intersect the result with the lexicographic order $<_{\text{LEX}}$:

$$\forall e \in \mathbf{E}, \forall u, v \in L_{\text{CTRL}} : \quad u\, \delta_e\, v \iff u\Big(\kappa_e \cap ((\mathbf{W} \times \mathbf{W}) \cup (\mathbf{W} \times \mathbf{R}) \cup (\mathbf{R} \times \mathbf{W})) \cap <_{\text{LEX}}\Big) v.$$

Thanks to techniques described in Section 3.6.2, we can always compute a conservative approximation δ of δ_e. Relation δ is realized by a rational transducer in the case of trees and by a one-counter transducer in the case of arrays or nested trees and arrays.

Approximations may either come from the previous approximation κ of κ_e or from the intersection itself. The intersection may indeed be approximate in the case of trees and nested trees and arrays, because rational relations are not closed under intersection (see Section 3.3). But thanks to Proposition 3.13 it will always be exact for arrays. More details in each data structure case can be found in Sections 4.4, 4.5 and 4.6. We can now give a general dependence analysis algorithm for our program model. The Dependence-Analysis algorithm is exactly the same for every kind of data structure, but individual steps may be implemented differently.

Dependence-Analysis (*program*)
    program: an intermediate representation of the program
    returns a dependence relation between all accesses
    $f \leftarrow$ Compute-Storage-Mappings (*program*)
    $\kappa \leftarrow (f^{-1} \circ f)$
    if κ is a multi-counter transduction
        then $\kappa \leftarrow$ one-counter approximation of κ
    if the underlying rational transducer of κ is not left-synchronous
        then $\kappa \leftarrow$ resynchronization with or without approximation of κ
    $\kappa \leftarrow \kappa \cap ((\mathbf{W} \times \mathbf{W}) \cup (\mathbf{W} \times \mathbf{R}) \cup (\mathbf{R} \times \mathbf{W}))$
    $\delta \leftarrow \kappa \cap <_{\text{LEX}}$
    return δ

The result of Dependence-Analysis is limited to dependences on a specific data structure. To get the full dependence relation of the program, it is necessary to compute the union for all the data structures involved.

4.3.3 From Dependences to Reaching Definitions

Remember the formal definition in Section 2.4.2: the exact reaching definition relation is defined as a lexicographic selection of the last write access in dependence with a given read access, i.e.

$$\forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e : \quad \sigma_e(u) = \max_{<_{\text{LEX}}} \{v \in \mathbf{W}_e : v\, \delta_e\, u\}.$$

Clearly, this maximum is unique for each read access u in the course of execution. In the case of an *exact* knowledge of δ_e, and when this relation is *left-synchronous*, one may easily compute an *exact* reaching definition relation, using *lexicographic selection*, see Section 3.4.3.

The problem is that δ_e is not known precisely in general, and the above solution is rarely applicable. Moreover, using the computation scheme above, conditionals and loop bounds have not been taken into account: the result is that many *non-existing* accesses are considered dependent for relation δ. We should thus be looking for a conservative approximation σ of σ_e, built on the available approximate dependence relation δ. Relying on δ makes computation of σ from (4.17) almost impossible, for two reasons: first, a write v may be in dependence with u without being executed by the program, and second, all writes which are not effectively in conflict with u may be considered as possible dependences.

However, we know we can compute an approximate reaching definition relation σ from δ when at least one of the following conditions is satisfied.

- Suppose we can prove that some statement instance *does not* execute, and that this information can be inserted in the original transduction: some flow dependences can be removed. The remaining instances are described by predicate $e_{\text{MAY}}(w)$ (instances that *may* execute).

- On the opposite, if we can prove that some instance w *does* execute, and if this information can be inserted in the original transduction, then writes executing before w are "killed": they cannot reach an instance u such that $w\,\delta\,u$. Instances that are effectively executed are described by predicate $e_{\text{MUST}}(w)$ (instances that *must* execute).

- Eventually, one may have some information $e_{\text{CONDITIONAL}}(v, w)$ about an instances w that *does* execute whenever another instance v does: this "conditional" information is used the same way as the former predicate e_{MUST}.

The more precise the predicates e_{MAY}, e_{MUST} and $e_{\text{CONDITIONAL}}$, the more precise the reaching definition relation. In some cases, one may even compute an exact reaching definition relation.

Now, remember all our work since Section 4.2 has completely ignored guards in conditional statements and loop bounds. This information is of course critical when trying to build predicates e_{MAY}, e_{MUST} and $e_{\text{CONDITIONAL}}$. Retrieving this information can be done using both the results of induction variable analysis (see Section 4.2) and additional analyses of the *value of variables* [CH78, Mas93, MP94, TP95]. Such *external* analyses would for example compute loop and recursion *invariants*.

Another source of information—mostly for predicate $e_{\text{CONDITIONAL}}$—is provided by a simple *structural analysis* of the program, which consists in exploiting every information hidden in the program syntax:

- in a `if` ··· `then` ··· `else` ··· construct, either the `then` or the `else` branch is executed;

- in a `while` construct, assuming some instance of a statement does execute, all instances preceding it in the `while` loop also execute;

- in a sequence of non-guarded statements, all instances of these statements are simultaneously executed or not;

Notice this kind of structural analysis was already critical for nested loops [BCF97, Bar98, Won95].

Another very important structural property is described with the following additional definition:

Definition 4.2 (ancestor) Consider an alphabet Σ_{CTRL} of statement labels and a language L_{CTRL} of control words. We define Σ_{UNCO}: a subset of Σ_{CTRL} made of all *block* labels which are *not* conditionals or loop blocks, and all (unguarded) *procedure call* labels, i.e. blocks whose execution is *unconditional*.

Let r and s be two statements in Σ_{CTRL}, and let u be a *strict* prefix of a control word $wr \in L_{\text{CTRL}}$ (an instance of r). If $v \in \Sigma_{\text{UNCO}}^*$ (without labels of conditional statements) is such that $uvs \in L_{\text{CTRL}}$, then uvs is called an *ancestor* of wr.

The set of ancestors of an instance u is denoted by $\textsc{Ancestors}(u)$.

This definition is best understood on a *control tree*, such as the one in Figure 4.1.b page 124: black square $FPIA\textit{AaAaAJs}$ is an ancestor of $FPIA\textit{AaAaAJQPIAaBBr}$, but not gray squares $FPIA\textit{AaAJs}$ and $FPIA\textit{AJs}$. Now, observe the formal ancestor definition:

1. execution of wr implies execution of u, because it is in the path from the root of the control tree to node wr;

2. execution of u implies execution of uvs, because v is made of declaration blocks only, without conditional statements.

We thus have the following result:

Proposition 4.1 If an instance u executes, then all ancestors of u also execute. This can be written using predicates e_{MUST} and $e_{\text{CONDITIONAL}}$:

$$\forall u \in L_{\text{CTRL}}: \quad e_{\text{CONDITIONAL}}(u, \textsc{Ancestors}(u)),$$
$$\forall u \in L_{\text{CTRL}}: \quad e_{\text{MUST}}(u) \implies e_{\text{MUST}}(\textsc{Ancestors}(u)).$$

At last, we can define a conservative approximation σ of the reaching definition relation, built on δ, e_{MAY}, e_{MUST} and $e_{\text{CONDITIONAL}}$:

$$\forall u \in \mathbf{R}: \quad \sigma(u) = \big\{v \in \delta(u) : e_{\text{MAY}}(v) \wedge \big(\nexists w \in \delta(u) : v <_{\text{LEX}} w \wedge (e_{\text{MUST}}(w) \vee e_{\text{CONDITIONAL}}(v, w) \vee e_{\text{CONDITIONAL}}(u, w))\big\}. \quad (4.17)$$

Predicates e_{MAY}, e_{MUST}, $e_{\text{CONDITIONAL}}$ should define rational sets, in order to compute the algebraic operations involved in (4.17). When, in addition, relation δ is *left-synchronous*, closure under union, intersection, complementation, and composition, allows unaproximate computation of σ with (4.17).

However, designing a general computation framework for these predicates is left for future work, and we will only consider a few "rules" useful in our practical examples.

4.3.4 Practical Approximation of Reaching Definitions

Instead of building automata for predicates e_{MAY}, e_{MUST} and $e_{\text{CONDITIONAL}}$ then computing σ from (4.17), we present a few *rewriting rules* to *refine* the sets of possible reaching definitions, starting from a very conservative approximation of the reaching definition relation: the restriction of dependence relation δ to flow dependences (i.e. from a write to a read access). This technique is less general than solving (4.17), but it avoids complex—and approximate—algebraic operations.

Applicability of the rewriting rules is governed by the compile-time knowledge extracted by external analyses, such as analysis of contitional expressions, detection of invariants, or structural analysis. In the Section 4.5, we will demonstrate practical usage of these rules when applying our reaching definition analysis framework to program `Queens`.

For the moment, we choose a statement s with a write reference to memory, and try to refine sets of possible reaching definitions *among instances of s*. Refining sets of possible reaching definitions which are instances of several statements will be discussed at the end of this section.

The VPA Property (Values are Produced by Ancestors)

This property comes from the common observation about recursive programs that "values are produced by ancestors". Indeed, a lot of sort, tree, or graph-based algorithms perform in-depth explorations where values are produced by ancestors. This behavior is also strongly assessed by scope rules of local variables.

$$\textsc{vpa} \iff \forall e \in \mathbf{E}, u \in \mathbf{R}_e, v \in \mathbf{W}_e : \big(v = \sigma_e(u) \implies v \in \textsc{Ancestors}(u)\big).$$

Since all possible reaching definitions are ancestors of the use, rule VPA consists in removing all transitions producing non-ancestors. Formally, all transitions $\alpha'|\alpha$ s.t. $\alpha' <_{\text{TXT}} \alpha$ and $\alpha' \neq s$ are removed.

We may define one other interesting property useful to automatic property checking; its associated rewriting rule is not given.

The OKA Property (One Killing Ancestor)

If it can be proven that at least one ancestor vs of a read u is in dependence with u, it kills all previous writes since it does execute when u does.

$$\text{OKA} \iff \forall u \in \mathbf{R} : \big(\delta(u) \neq \varnothing \implies (\exists v \in \textsc{Ancestors}(u) : v \in \delta(u))\big).$$

Property Checking

Property OKA can be discovered using invariant properties on induction variables. Checking for property VPA is difficult, but we may rely on the following result: when property OKA holds, checking VPA is equivalent to checking whether an ancestor vs in dependence with us may followed—according to the lexicographic order—by a non-ancestor instance w in dependence with us.

Other properties can be obtained by more involved analyses: the problem is to find a relevant rewriting rule for each one.

Now, remember we restricted ourselves to one assignation statement s when presenting the rewriting rules. Designing rules which handle the global flow of the program is a bit more difficult. When comparing possible reaching definition instances of two writes s_1 and s_2, it is not possible in general to decide whether one may "kill" the other without a specific transducer (rational or one-counter, depending on the data structure). The problem is thus to intersect two rational or algebraic relations, which cannot be done without approximations in general, see Sections 3.6 and 3.7. In many cases, however, storage mappings for s_1 and s_2 are very similar, and exact results can be easily computed.

The REACHING-DEFINITION-ANALYSIS algorithm is a general algorithm for reaching definition analysis inside our program model. Algebraic operations on sets and relations in the second loop of the algorithm may yield approximative results, see Sections 3.4, 3.6 and 3.7. The intersection with $\mathbf{R} \times \{w : e_{\text{MAY}}(w)\}$ in the third line serves the purpose of restricting the domain to read accesses and the image to writes which may execute; it can be computed exactly since $\mathbf{R} \times \{w : e_{\text{MAY}}(w)\}$ is a recognizable relation. The REACHING-DEFINITION-ANALYSIS algorithm is applied to program `Queens` in Section 4.5.

Notice that *all output and anti-dependences are removed* by the algorithm, but some spurious flow dependences may remain when the result is approximate.

Now, there is something missing in this presentation of reaching definition analysis: what about the $\perp$ instance? When predicates $e_{\text{MUST}}(v)$ or $e_{\text{CONDITIONAL}}(u, v)$ are empty for all possible reaching definitions v of a read instance u, it means that an uninitialized value may be read by u, hence that $\perp$ is a possible reaching definition; and the reciprocal is true. In terms of our "practical properties", OKA can be used to determine whether $\perp$ is a possible reaching definition or not. This gives an automatic way to insert $\perp$ when needed in the result of REACHING-DEFINITION-ANALYSIS.

To conclude this section, we have shown a very clean and powerful framework for *instancewise dependence analysis* of recursive programs, but we should also recognize the limits of relying on a list of refinement rules to compute an approximate reaching

Reaching-Definition-Analysis (*program*)
program: an intermediate representation of the program
returns a reaching definition relation between all accesses
compute e_{MAY}, e_{MUST} and $e_{\text{CONDITIONAL}}$ using structural and external analyses
$\delta \leftarrow$ Dependence-Analysis
$\delta \leftarrow \delta \cap (\mathbf{R} \times \{w : e_{\text{MAY}}(w)\})$
for each assignment statement s **in** *program*
do check δ for properties OKA, VPA, and other properties
using external static analyses or asking the user
apply refinement rules on δ accordingly
for each pair of assignment statements (s, t) **in** *program*
do $kill \leftarrow \{(us, w) \in \mathbf{W} \times \mathbf{R} : (\exists vt \in \mathbf{W} : us\, \delta\, w \wedge vt\, \delta\, w \wedge us <_{\text{LEX}} vt$
$\wedge\, (e_{\text{MUST}}(vt) \vee e_{\text{CONDITIONAL}}(us, vt) \vee e_{\text{CONDITIONAL}}(w, vt)))\}$
$\delta \leftarrow \delta - kill$
return δ

definition relation from an approximate dependence relation. Now that the feasibility of *instancewise reaching definition analysis* for recursive programs has been proven, it is time to work on a formal framework to compute predicates e_{MAY}, e_{MUST} and $e_{\text{CONDITIONAL}}$, from which we could expect a powerful reaching definition analysis algorithm.

4.4 The Case of Trees

We will now precise the dependence and reaching definition analysis in the case of a tree structure. Practical computations will be performed on program `BST` presented in 4.2.

The first part of the Dependence-Analysis algorithm consists in computing the storage mapping. When the underlying data structure is a tree, its abstraction is the free monoid $M_{\text{DATA}} = \{\mathtt{l}, \mathtt{r}\}$ and the storage mapping is a rational transduction between free monoids. Computation of function f for program `BST` has already been done in Section 4.2.5. Figure 4.7 shows a rational transducer realizing rational function f. Following the lines of Section 2.3.1 page 68, the alphabet of statement labels has been extended to distinguish between distinct *references* in I_2, J_2, b and e, yielding new labels $I_{2_{\mathtt{p}}}$, $I_{2_{\mathtt{p->l}}}$, $J_{2_{\mathtt{p}}}$, $J_{2_{\mathtt{p->r}}}$, $b_{\mathtt{p}}$, $b_{\mathtt{p->l}}$, $e_{\mathtt{p}}$ and $e_{\mathtt{p->r}}$ (these new labels may only appear as the last letter in a control word).

Computation of κ is done thanks to Elgot and Mezei's theorem, and yields a rational transduction. The result for program `BST` is given by the transducer in Figure 4.8.

When κ is realized by a left-synchronous transducer, the last part of the Dependence-Analysis algorithm does not require any approximation: dependence relation $\delta = \kappa \cap <_{\text{LEX}}$ can be computed exactly (after removing conflict between reads in κ). It is the case for program `BST`, and the exact dependence analysis result is shown in Figure 4.9. In the general case, a conservative left-synchronous approximation of κ must be computed, see Section 3.7.

One may immediately notice that every pair (u, v) accepted by the dependence transducer is of the form $u = wu'$ and $v = wv'$ where $w \in \{F, P, L, R, I_1, J_1\}^*$ and u', v' do not hold any recursive call—i.e. L or R. That means that all dependences lie between instances of the same block I_1 or J_1. We will show in Section 5.5 that this result can be used to run the first `if` block—statement I_1—in parallel with the second—statement J_1.

Eventually, it appears that dependence transduction δ is a *rational function*, and the

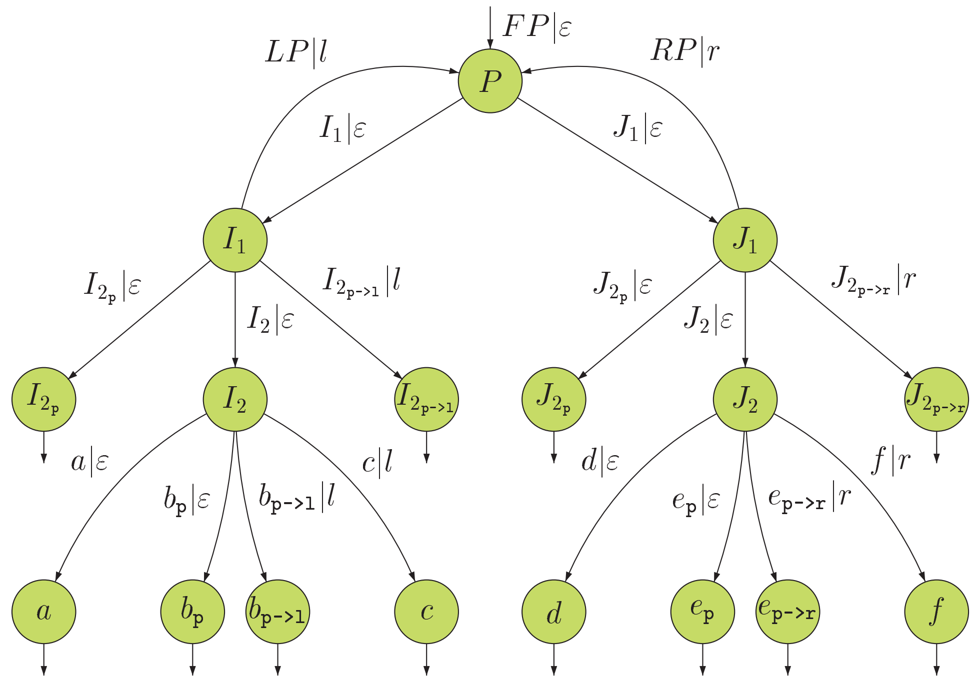


Figure 4.7. Rational transducer for storage mapping f of program `BST`

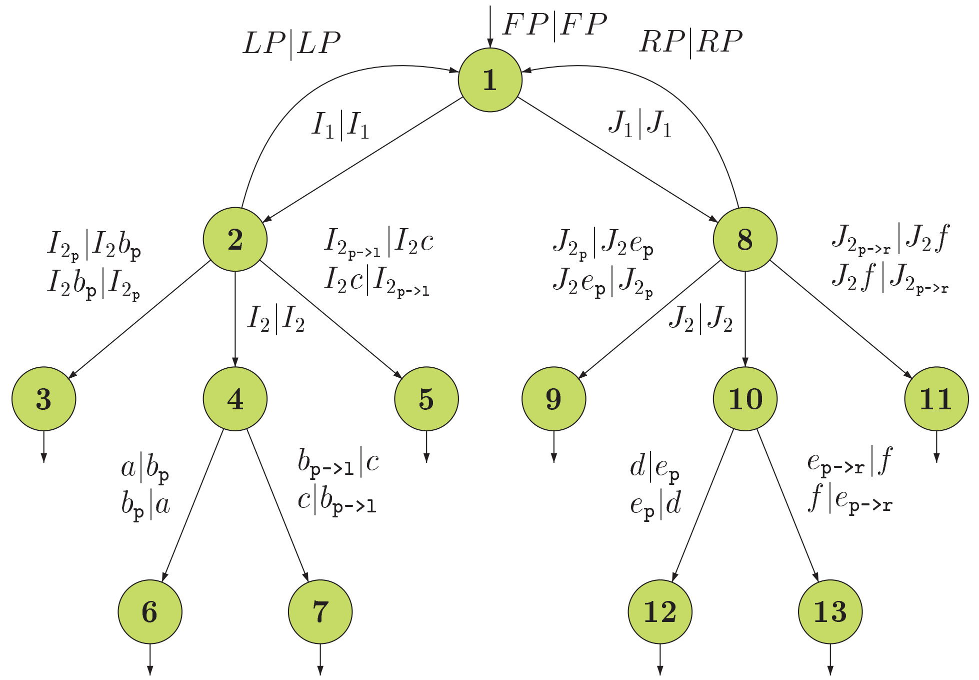


Figure 4.8. Rational transducer for conflict relation κ of program `BST`

restriction of δ to pairs (u, v) of a read u and a write v yields the *empty relation*! Indeed, the only dependences on program `BST` are anti-dependences.

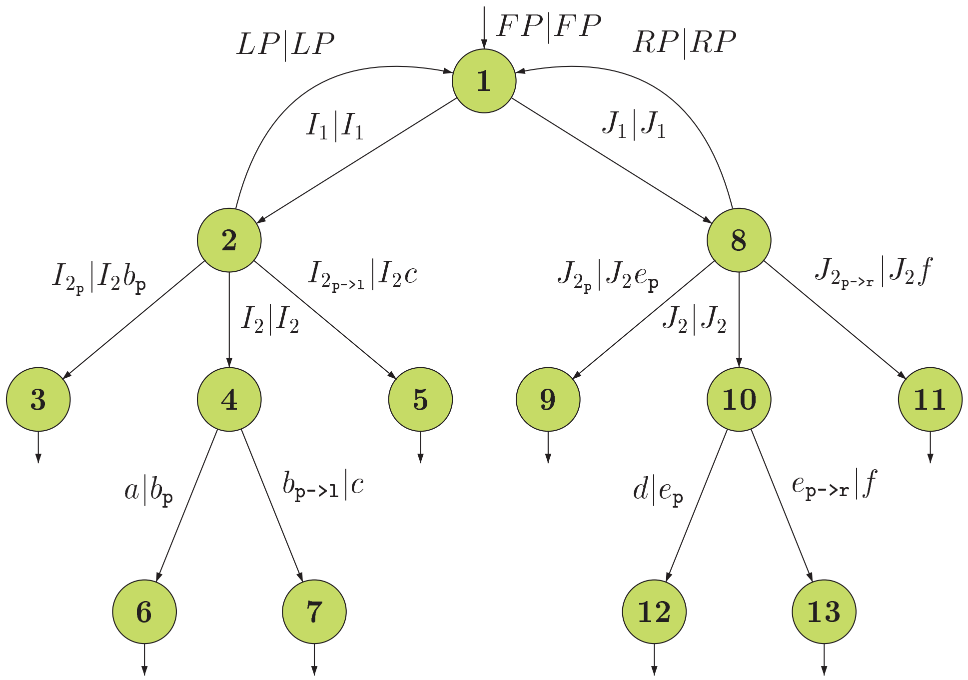


Figure 4.9. Rational transducer for dependence relation δ of program `BST`

4.5 The Case of Arrays

We will now precise the dependence and reaching definition analysis in the case of an array structure. Practical computations will be performed on program `Queens` presented in 4.1.

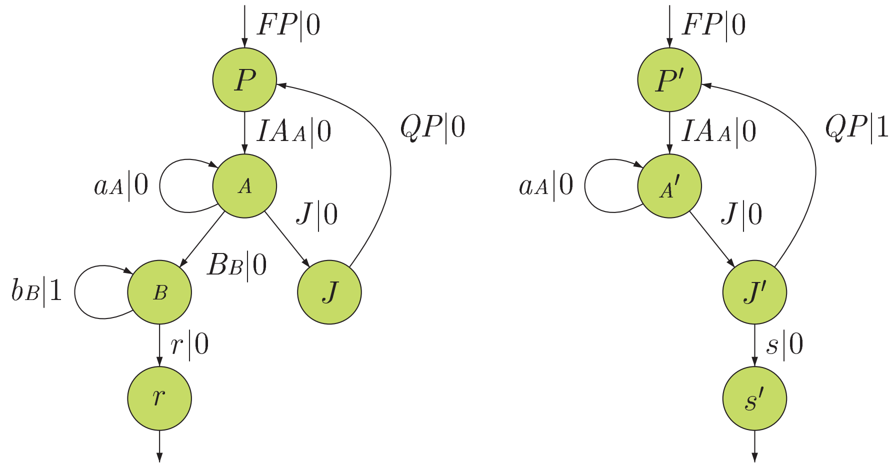


Figure 4.10. Rational transducer for storage mapping f of program `Queens`

The first part of the Dependence-Analysis algorithm consists in computing the storage mapping. When the underlying data structure is an array, its abstraction is the free commutative monoid $M_{\text{DATA}} = \mathbb{Z}$. Computation of function f for program `Queens` has already been done in Section 4.2.5. Figure 4.10 shows a rational transducer realizing rational function $f : \Sigma^*_{\text{CTRL}} \to \mathbb{Z}$. It reflects the combination of regular expressions (4.5) and (4.6).

Computation of κ is done thanks to Theorem 3.27, and yields a one-counter transduction. The result for program `Queens` is given by the transducer in Figure 4.11—with four initial states.

To compute a dependence relation δ, one first restrict κ to pairs of accesses with at least one write, then intersect the result with the lexicographic order. From Proposition 3.13 the underlying rational transducer of κ is *recognizable*, hence *left-synchronous* (from Theorem 3.12) and can thus be resynchronized with the constructive proof of Theorem 3.19 to get a one-counter transducer whose underlying rational transducer is left-synchronous.

Resynchronization of κ has been applied to program `Queens` in Figure 4.12: it is limited to conflicts of the form (us, vr), $us, vr \in L_{\text{CTRL}}$. The lacking three fourths of the transducer have not been represented because they are very similar the the first fourth and not used for reaching definition analysis. The underlying rational transducer is only pseudo-left-synchronous because resynchronization has not been applied completely, see Section 3.6 and Definition 3.28

Intersection with $<_{\text{LEX}}$ is done with Theorem 3.14. As a result, the dependence relation δ can be computed exactly and is realized by a one-counter transducer whose underlying rational transducer is left-synchronous.

This is applied to program `Queens` in Figure 4.13, starting from the pseudo-left-synchronous transducer in Figure 4.12. Knowing that $B <_{\text{TXT}} J <_{\text{TXT}} a$ and $s <_{\text{TXT}} Q$, transitions $J|a$ and $s|Q$ are kept but transitions $a|J$, $a|B$ and $J|B$ are removed (and the transducer is trimmed). This time, only one third of the actual transducer is shown: the transducer realizing flow dependences. Anti and output dependences are realized by very similar transducers, and are not used for reaching definition analysis.

We now demonstrate the Reaching-Definition-Analysis algorithm on program `Queens`. A simple analysis of the inner loop shows that j is always less than k. This proves that for any instance w of r, there exists $u, v \in \Sigma^*_{\text{CTRL}}$ s.t. $w = uQvr$ and $us\,\delta\,uQvr$. Because us is an ancestor of $uQvr$, property OKA is satisfied. Dependence transducer in Figure 4.13 shows that all instances of s executing after us are of the form $uQv's$, and it also shows that *reading* Q increases the counter: the result is that no instance executing after us may be in dependence with w. In combination with OKA, property VPA thus holds. Applying rule VPA, we can remove transition $J|a_A$ which does not yield ancestors. We get the one-counter transducer in Figure 4.14. Notice that the $\perp$ instance (associated with uninitialized values) is not accepted as a possible reaching definition: this is because property OKA ensures that at least an ancestor of every read instance defined a value.

The transducer is "compressed" in Figure 4.15 to increase readability. It is easy to prove that this result is exact: a unique reaching definition is computed for every read instance. However, the general problem of the functionality of an algebraic transduction is "probably" undecidable. As a result, we achieved—in a semi-automated way—the best precision possible. This precise result will be used in Section 5.5 to parallelize program `Queens`.

4.6 The Case of Composite Data Structures

We will now precise the dependence and reaching definition analysis in the case of a nested list and array structure. Practical computations will be performed on program `Count` presented in 4.3.

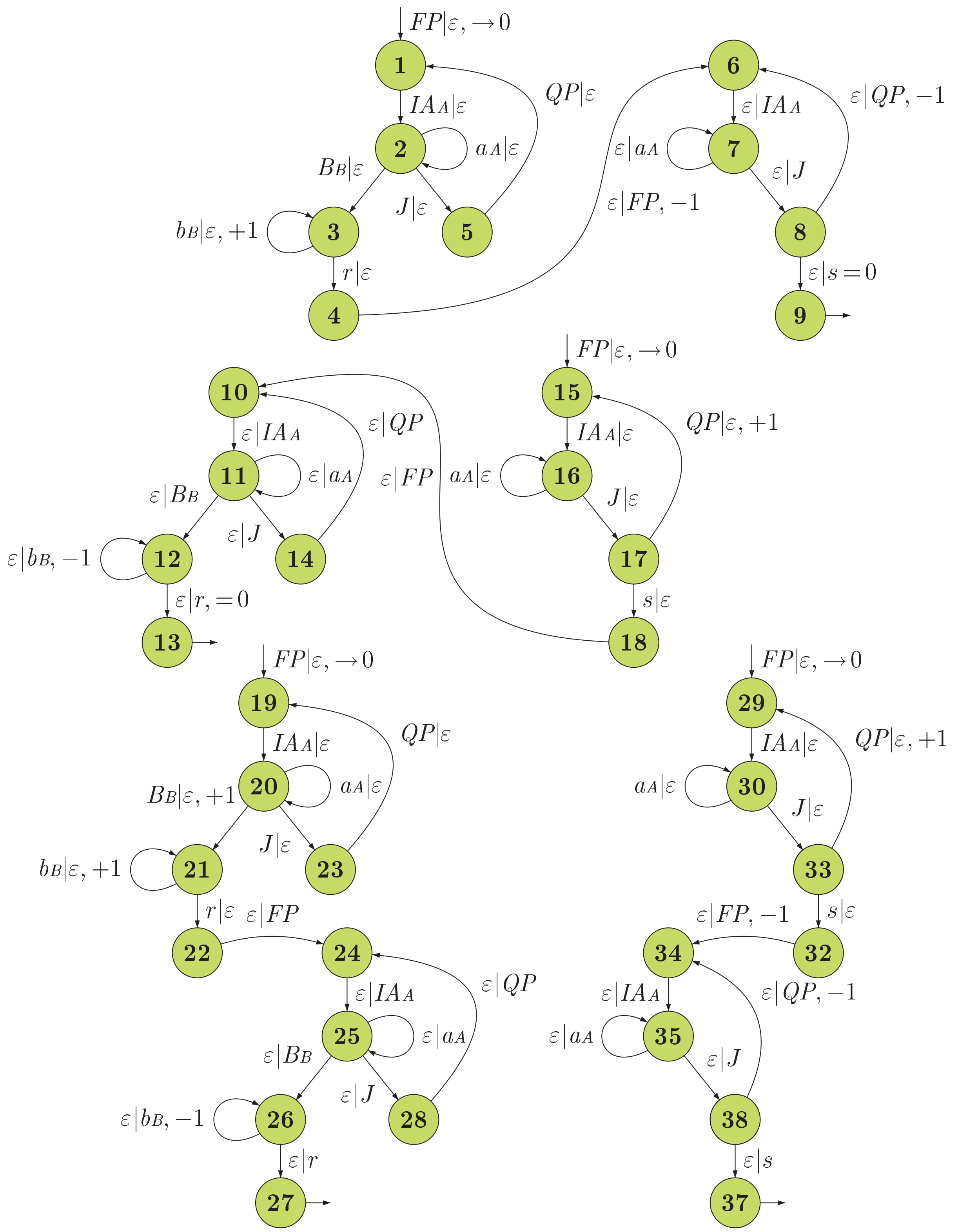


Figure 4.11. One-counter transducer for conflict relation κ of program `Queens`

The first part of the DEPENDENCE-ANALYSIS algorithm consists in computing the storage mapping. When the underlying data structure is built of nested trees and arrays, its abstraction is a free partially commutative monoid M_{DATA}. Computation of function f for program `Count` has already been done in Section 4.2.5.

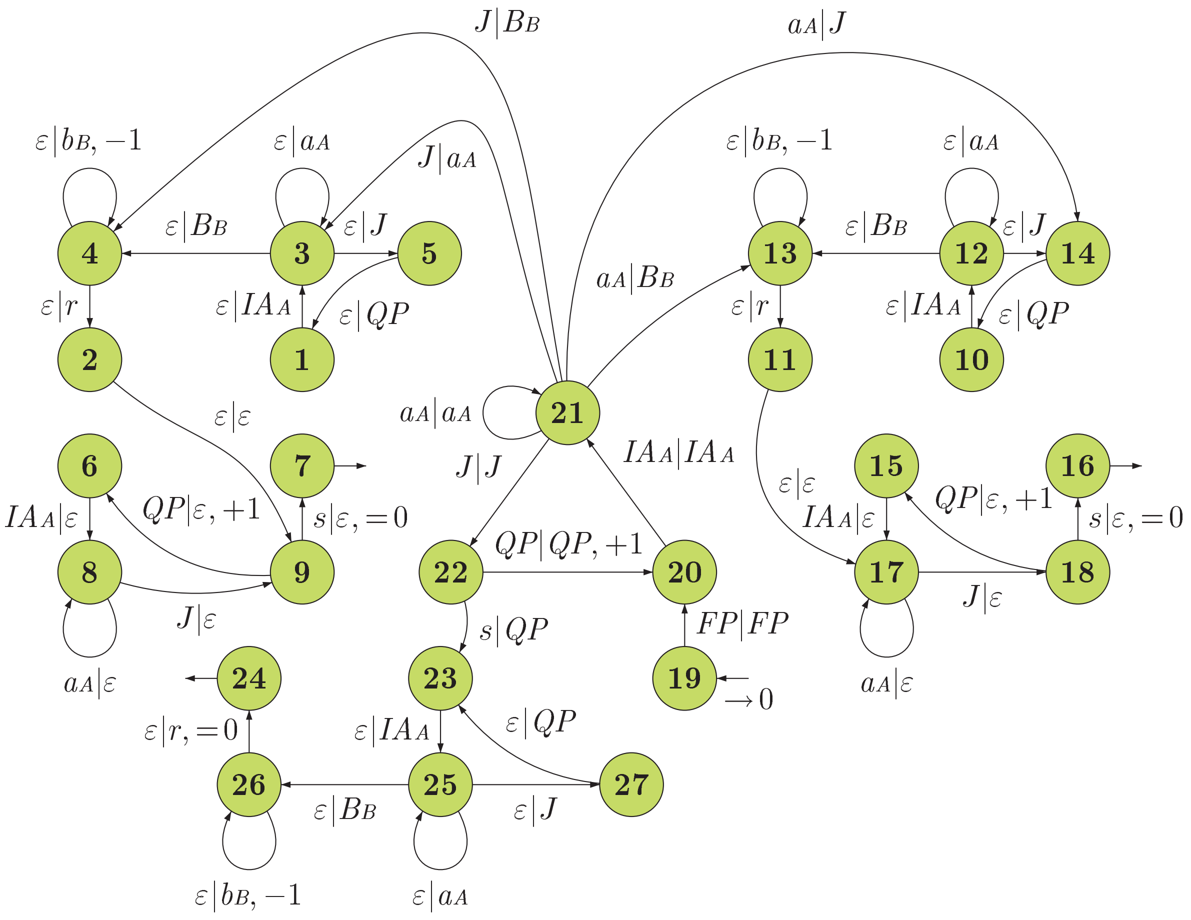


Figure 4.12. Pseudo-left-synchronous transducer for the restriction of κ to $\mathbf{W} \times \mathbf{R}$

Computation of κ is done thanks to Theorem 3.28, and yields a one-counter transduction. On program `Count`, there are no write accesses to the `inode` structure. Now, we could be interested in an analysis of conflict-misses for cache optimization [TD95]. The result $f^{-1} \circ f$ for program `Count` is thus interesting, and it is the identity relation! This proves that the same memory location is never accessed twice during program execution.

Now, when computing a dependence relation in general, Proposition 3.13 does not apply: it is necessary in general to approximate the underlying rational transducer by a left-synchronous one. Eventually, the Reaching-Definition-Analysis algorithm has no technical issues specific to nested trees and arrays.

4.7 Comparison with Other Analyses

Before evaluating our analysis for recursive programs, we summary its program model restrictions. First of all, some restrictions are required to simplify algorithms and should be considered harmless thanks to previous code transformations—see Sections 2.2 and 4.2 for details:

- no function pointers (i.e. higher-order control structures) and no gotos are allowed;

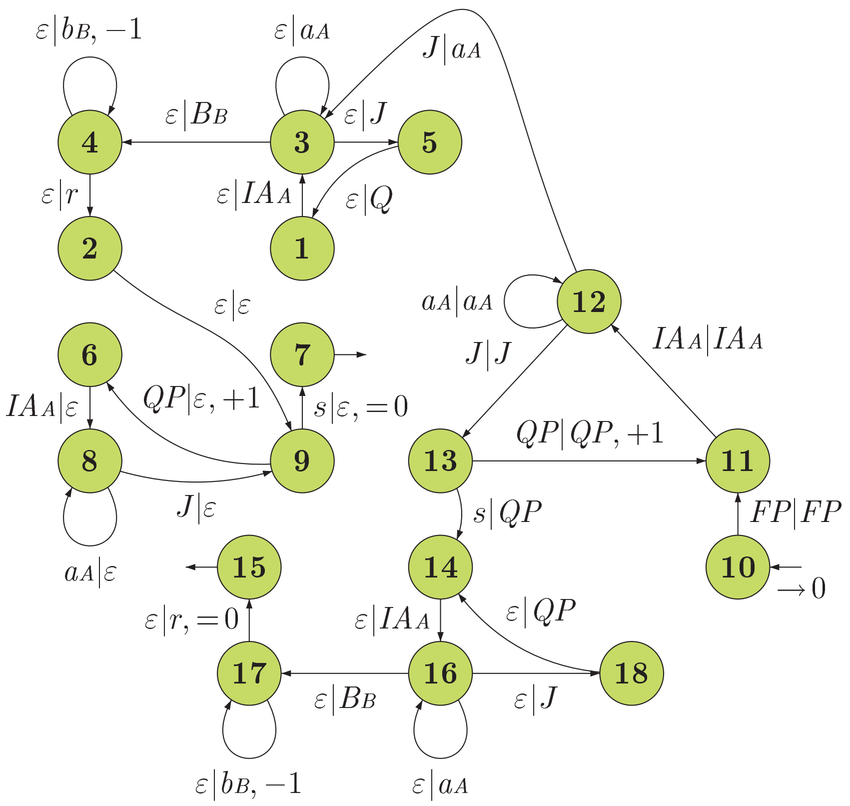


Figure 4.13. One-counter transducer for the restriction of dependence relation δ to flow dependences

- a loop variable is initialized at the loop entry and used only inside this loop;
- expressions in right-hand side may hold conditionals but no function calls and no loops;
- every data structure subject to dependence or reaching definition analysis must be declared global;

Now, some restrictions on the program model cannot been avoided with preliminary program transformations, but should be removed in further versions of the analysis, thanks to appropriate approximation techniques (induction variables are defined in Section 4.2):

- only scalars, arrays, trees and nested trees and arrays are allowed as data structures;
- induction variables must follow very strong rules regarding initialization and update;
- every array subscript must be an affine function of integer induction variables and symbolic constants;
- every tree access must dereference a pointer induction variable or a constant.

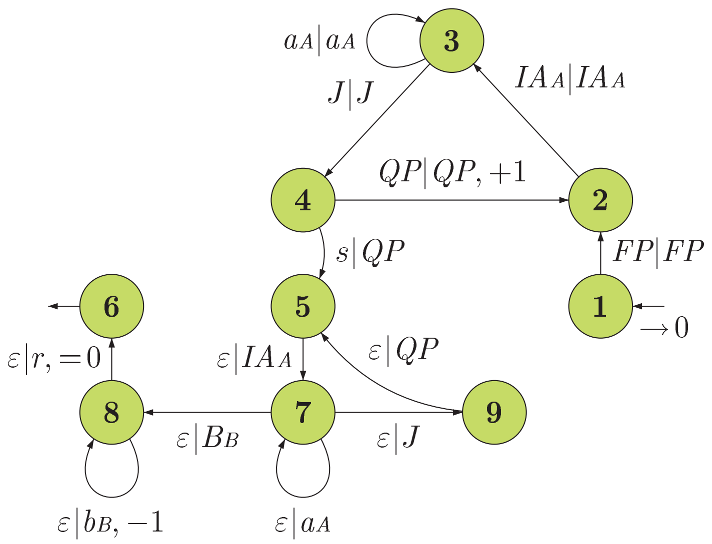


Figure 4.14. One-counter transducer for reaching definition relation σ of program `Queens`

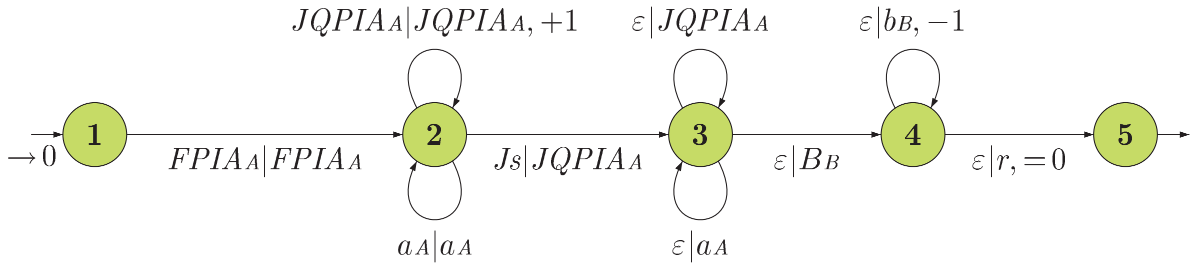


Figure 4.15. Simplified one-counter transducer for σ

Eventually, one restriction is very deeply rooted in the monoid abstraction for tree structures, and we expect no general way to avoid it:

- random insertions and deletions in trees are forbidden (allowed only at trees' leaves).

We are now able to compare the results of our analysis technique with those of classical static analyses—some of which also handle our full program model—and with those of the existing instancewise analyses for loop nests.

Static dependence and reaching definition analyses generally compute the same kind of results, whether they are based on abstract interpretation [Cou81, JM82, Har89, Deu94] or other data-flow analysis techniques [LRZ93, BE95, HHN94, KSV96]. A comprehensive study of static analysis useful to parallelization of recursive programs can be found in [RR99]. Comparison of the results is rather easy: *none* of these static analyses is *instancewise*.[4] None of these static analyses is able to tell which instance of which statement

[4] We think that building an instancewise analysis of practical interest in the data-flow or abstract interpretation framework is indeed possible, but very few works have been made in this direction, see

is in conflict, in dependence, or a possible reaching definition. However, these analyses are very useful to remove a few restrictions in our program model, and they also compute properties useful to instancewise reaching definition analysis. Remember that our own instancewise reaching definition analysis technique makes a heavy use of so called "external" analyses, which precisely are classical static analyses. A short comparison between parallelization from the results of our analysis and parallelization from static analyses will be proposed in Section 5.5, along with some practical examples.

Comparison with instancewise analyses for loop nests is more topical, since our technique was clearly intended to extend such analyses to recursive programs. A simple method to get a fair evaluation consists in running both analyses on their common program model subset. The general result is not surprising: today's most powerful reaching definition analyses for loop nests such as fuzzy array dataflow analysis (FADA) [BCF97, Bar98] and constraint-based array dependence analysis [WP95, Won95] are far more precise than our analysis for recursive programs. There are many reasons for that:

- we do not use conditionals and loop bounds to establish our results, or when it is the case, it is through "external" static analyses;
- multi-dimensional arrays are roughly approximated by one-dimensional ones;
- rational and algebraic transducers have a limited expressive power when dealing with integer parameters (only one counter can be described);
- some critical algebraic operations such as intersection and complementation are not decidable and thus require further approximations.

A major difference between FADA and our analysis for recursive program is deeply rooted the philosophy of each technique.

- FADA is a fully exact process with symbolic computations and "dummy" parameters associated with unpredictable constraints, and only one approximation is performed at the end; this ensures that no precious data-flow information is lost during the computation process (see Section 2.4.3).
- Our technique is not as clever, since many approximation stages can be involved. It is more similar to iterative methods in that sense, and hence it is far from being optimal: some approximations are made even if the mathematical abstraction could have enough expressive power to avoid it.

But the comparison also reveals very positive aspects, in terms of all the information available in the result of our analysis:

- exactness of the result is equivalent to deciding the functionality of a transduction, and is thus polynomial for rational transductions; but it is unknown for algebraic ones, and decidability of the finiteness of a set of reaching definitions can help in some cases;
- emptiness of a set of reaching definitions is decidable, which allows automatic detection of read accesses to uninitialized variables;

[DGS93, Tzo97, CK98].

- in the case of rational transductions, dependence testing can be extended to rational languages of control words, because of Nivat's Theorem 3.6 and the fact that rational languages are closed under intersection; this is very useful for parallelization;
- in the case of algebraic transductions, dependence testing is equivalent to the intersection of an algebraic language and a rational one, because of Nivat's Theorem 3.21 for algebraic transductions and Evey's Theorem 3.24; this is still very useful for parallelization.

We refer to Section 5.5 for additional comparisons between the applicability of our analysis and loop nest analyses to parallelization.

4.8 Conclusion

We presented an application of formal language theory to the automatic discovery of some semantic properties of programs: instancewise dependences and reaching definitions. When programs are recursive and nothing is known about recursion guards, only conservative approximations can be hoped for. In our case, we approximate the relation between reads and their reaching definitions by a rational (for trees) or algebraic (for arrays) transduction. The result of the reaching definition analysis is a transducer mapping control words of read instances to control words of write instances. Two algorithms for dependence and reaching definition analysis of recursive programs were designed. Incidentally, these results showed the use of the new class of left-synchronous transductions over free monoids.

We have applied our techniques on several practical examples, showing excellent approximations and sometimes even exact results. Some problems obviously remain. First, some strong restrictions on the program model limit the practical use of our technique. We should thus work on a graceful degradation of our analyses to encompass a larger set of recursive programs: for example, restrictions on induction variables operations could perhaps be removed by allowing computation of approximate storage mappings. Second, reaching definition analysis is not quite mature now, since it relies on rather ad-hoc techniques whose general applicability is unknown. More theoretical studies are needed to decide whether precise instancewise reaching definition information can be captured by rational and algebraic transducers.

We will show in the next chapters that decidability properties on rational and algebraic transductions allow several applications of our framework, especially in automatic parallelization of recursive programs. These applications include array expansion and parallelism extraction.

Chapter 5

Parallelization via Memory Expansion

The design of program transformations dedicated to dependence removal is a well studied topic, as far as nested loops are concerned. Techniques such as conversion to single-assignment form [Fea91, GC95, Col98], privatization [MAL93, TP93, Cre96, Li92], and many optimizations for efficient memory management [LF98, CFH95, CDRV97, QR99] have been proven useful for practical parallelization of programs (automatically or not). However, these works have mostly targeted affine loop nests and few techniques have been extended to dynamic control flow and general array subscripts. Very interesting issues arise when trying to expand data structures in unrestricted nests of loops, and because of the necessary data-flow restoration, confluent interests with the SSA (static single-assignment) [CFR$^+$91] framework become obvious.

Motivation for memory expansion and introduction of the fundamental concepts is the first goal of Section 5.1; then, we study specific problems related with non-affine nests of loops and we design practical solutions for a general single-assignment form transformation. Novel expansion techniques presented in Sections 5.2, 5.3 and 5.4 are contributions to bridging the gap between the rich applications of memory expansion techniques for affine loop nests and the few results with irregular codes.

When extending the program model to recursive procedures, the problem is of another nature: principles of parallel processing are then very different from the well mastered data parallel model for nested loops. Applicable algorithms have been mostly designed for statementwise dependence tests, when our analysis computes an extensive instance-wise description of the dependence relation! There is of course a large gap between the two approaches and we should now demonstrate that using such a precise information brings practical improvements over existing parallelization techniques. These issues are addressed by Section 5.5, starting with an investigation of memory expansion techniques for recursive programs. Because this last section addresses a new topic, several negative or disappointing answers are mixed with successful results.

5.1 Motivations and Tradeoffs

To point out the most important issues related with memory expansion, and to motivate the following sections of this chapter, we start with a study of the well-known expansion technique called conversion to single-assignment form. Both abstract and practical point of views are discussed. Several results presented here have been already presented by many authors, with their formalism and their program model, but we prefered to rewrite most of this work in our syntax to fix the notations and to show how memory expansion also makes sense out of the loop nest programming model.

5.1.1 Conversion to Single-Assignment Form

One of the most usual and simplest expansion schemes is conversion to *single-assignment* (SA) form. It is the extreme case where each memory location is written at most once during execution. This is slightly different from *static* single-assignment form (SSA) [CFR+91, KS98], where each variable is written at most in *one statement* in the program, and expansion is limited to variable renaming.

The idea of conversion to SA-form is to replace every assignment to a data structure D by an assignment to a new data structure $\mathtt{D}_{\text{EXP}}$ whose elements have the same type as elements of D, and are in one-to-one mapping with the set $\mathbf{W}$ of all possible write accesses during any program execution. Each element of $\mathtt{D}_{\text{EXP}}$ is associated to a single write access. This aggressive transformation ensures that the same memory location is never written twice in the expanded program. The second step is to transform the read references accordingly, and is called *restoration of the flow of data*. Instancewise reaching definition information is of great help to achieve this: for a given program execution $e \in \mathbf{E}$, the value read by some access $\langle \imath, \mathtt{ref} \rangle$ to D in right-hand side of a statement is precisely stored in the element of $\mathtt{D}_{\text{EXP}}$ associated with $\sigma_e(\langle \imath, \mathtt{ref} \rangle)$ (see Section 2.4 for notations and definitions). In general, an exact knowledge of σ_e for each execution e is not available at compile time: the result of instancewise reaching definition analysis is an approximate relation σ. The *compile-time* data-flow restoration scheme above is thus unapplicable when $\sigma(\langle \imath, \mathtt{ref} \rangle)$ is a non-singleton set: the idea is then to generate a *run-time* data-flow restoration code, which tracks what is the *last* instance executed in $\sigma(\langle \imath, \mathtt{ref} \rangle)$. As we have seen for general expansion schemes in Section 1.2, this run-time restoration code is hidden in a ϕ function whose argument is the set $\sigma(\langle \imath, \mathtt{ref} \rangle)$ of possible reaching definitions.

A few notations are required to simplify the syntax of expanded programs.

- CurIns holds the run-time instance value, encoded as a control word or iteration vector, for any statement in the program. It is supposed to be updated on-line in function calls, loop iterations and every block entry. More precisions about this topic in Section 5.1.3 and Section 5.5.3.

- ϕ has the syntax of a function from sets of run-time instances to untyped values, but its semantics is to summarize a piece of data-flow restoration code. It is very similar to ϕ functions in the SSA framework [CFR+91, KS98]. Code generation for ϕ functions is the purpose of Section 5.1.2.

- $\mathtt{D}_{\text{EXP}}$ is the expanded data structure associated with some original data structure D. Its "abstract" syntax is inherited from arrays: $\mathtt{D}_{\text{EXP}}$`[`*`set of element names`*`]` for the declaration and $\mathtt{D}_{\text{EXP}}$`[`*`element name`*`]` for the read or write access. In practice, element names are either integer vectors or words, and $\mathtt{D}_{\text{EXP}}$ is an array, a tree, or a nest of trees and arrays. Its "concrete" syntax is then implemented as an array or as a pointer to a tree structure. See Sections 5.1.3 and 5.5.1 for details.

We now present Abstract-SA: a very general algorithm to compute the single-assignment form. This algorithm is neither really new nor really practical, but it defines a general transformation scheme for SA programs, independently of the control and data structures. It takes as input the sequential program and the result of an instancewise reaching definition analysis—seen as a function. Control structures are left unchanged. This algorithm is very "abstract" since data structures are not defined precisely and some parts of the generated code have been encapsulated in high-level notations: CurIns and ϕ.

Abstract-SA $(program, \mathbf{W}, \sigma)$
  program: an intermediate representation of the program
  W: a conservative approximation of the set of write accesses
  σ: a reaching definition relation, seen as a function
  returns an intermediate representation of the expanded program
for each data structure `D` **in** *program*
do declare a data structure $\mathtt{D}_{\text{EXP}}[\mathbf{W}]$
  for each statement s assigning `D` **in** *program*
  do left-hand side of $s \leftarrow \mathtt{D}_{\text{EXP}}[\text{CurIns}]$
  for each reference ref to `D` **in** *program*
  do $ref \leftarrow$ `if` $(\sigma(\text{CurIns}, ref)$`==`$\{\bot\})$ ref
      `else if` $(\sigma(\text{CurIns}, ref)$`=`$\{\imath\})$ $\mathtt{D}_{\text{EXP}}[\imath]$
      `else` $\phi(\sigma(\text{CurIns}, ref))$
return *program*

We will show in the following that several "abstract" parts of the algorithm can be implemented when dealing with "concrete" data structures. Generating code for the ϕ function is the purpose of the next section.

5.1.2 Run-Time Overhead

When generating code for ϕ functions, the common idea is to compute at run-time the last instance that may possibly be a reaching definition of some use. In general, for each expanded data structure $\mathtt{D}_{\text{EXP}}$ one needs an additional structure in one-to-one mapping with $\mathtt{D}_{\text{EXP}}$. In the *static* single-assignment framework for *arrays* [KS98], these additional structures are called ***@-structures*** and store *statement instances*. Dealing with a more general single-assignment form, we propose another semantics for additional structures, hence another notation: the data structure in one-to-one mapping with $\mathtt{D}_{\text{EXP}}$ is a ***Φ-structures*** denoted by $\Phi\mathtt{D}_{\text{EXP}}$.

To ensure that run-time restoration of the flow of data is possible, elements of $\Phi\mathtt{D}_{\text{EXP}}$ should store two informations: the *memory location assigned in the original program* and the *identity of the last instance* which assigned this memory location. Because we are dealing with single-assignment programs, the identity of the last instance is already captured by the element itself (i.e. the subsrcipt of $\Phi\mathtt{D}_{\text{EXP}}$).[1] Elements of $\Phi\mathtt{D}_{\text{EXP}}$ should thus store *memory locations*.

- $\Phi\mathtt{D}_{\text{EXP}}$ is initialized to `NULL` before the expanded program;
- Every time $\mathtt{D}_{\text{EXP}}$ is modified, the associated element of $\Phi\mathtt{D}_{\text{EXP}}$ is set to the value of the *memory location* that would have been written in the original program.

[1] This run-time restoration technique is thus specific to SA-form. Other expansions require different type and/or semantics of Φ-structures.

- When a read access to D in the original program is expanded into a call of the form $\phi(set)$, the ϕ function is implemented as the maximum—according to the sequential execution order—of all $\imath \in set$ such that $\texttt{$\Phi$D}_{\texttt{EXP}}\texttt{[}\imath\texttt{]}$ is equal to the memory location read in the original program.

Abstract-Implement-Phi (*expanded*)
expanded: an intermediate representation of the expanded program
returns an intermediate representation with run-time restoration code
for each data structure $\texttt{D}_{\text{EXP}}$ **in** *expanded*
do if there are ϕ functions accessing $\texttt{D}_{\text{EXP}}$
  then declare a structure $\texttt{$\Phi$D}_{\text{EXP}}$ with the same shape as $\texttt{D}_{\text{EXP}}$ initialized to NULL
    for each read reference ref_ϕ to $\texttt{D}_{\text{EXP}}$ whose expanded form is $\phi(set)$
    do for each statement s involved in set
      do $ref_s \leftarrow$ write reference in s
        if not already done for s
          then following s insert $\texttt{$\Phi$D}_{\texttt{EXP}}\texttt{[CurIns] = } f_e(\textsc{CurIns}, ref_s)$
      $\phi(set) \leftarrow \texttt{D}_{\texttt{EXP}}\texttt{[max}_{<_{\texttt{seq}}}\ \{\imath \in set : \texttt{ΦD}_{\texttt{EXP}}\texttt{[}\imath\texttt{]} = f_e(\textsc{CurIns}, ref_\phi)\}\texttt{]}$
return *expanded*

Abstract-Implement-Phi is the abstract algorithm to generate the code for ϕ functions. In this algorithm, the syntax $f_e(\textsc{CurIns}, ref)$ means that we are interested in the memory location accessed by reference ref, and *not* that some compile-time knowledge of f_e is required. Of course, practical details and optimizations depend on the control structures, see Section 5.1.4. Notice that the generated code is still in SA form: each element of a new Φ-structure is written at most once.

An important remark at this point is that instancewise reaching definition analysis is the key to run-time overhead optimization. Indeed, as shown by our code generation algorithm, SA-transformed programs are more efficient when ϕ functions are sparse. Thus, a parallelizing compiler has many reasons to perform a precise instancewise reaching definition analysis: it improves parallelism detection, allow to choose between a larger scope of parallel execution orders (depending on the "grain size" and architecture), *and* reduces run-time overhead. An example borrowed from program `sjs` in [Col98] is presented in Figure 5.1. The most precise reaching definition relation for reference `A[i+j-1]` in right-hand side of R is

$$\sigma\left(\langle R, i, j, \texttt{A[i+j-1]}\rangle\right) = \left|\begin{array}{l} \textbf{\textit{if}}\ j \geq 1 \\ \textbf{\textit{then}}\ \langle S, i, j-1\rangle \\ \textbf{\textit{else}} \left|\begin{array}{l} \textbf{\textit{if}}\ i \geq 1 \\ \textbf{\textit{then}}\ \langle S, i-1, j\rangle \\ \textbf{\textit{else}}\ \langle T\rangle \end{array}\right. \end{array}\right. .$$

This exact result shows that definitions associated with the reference in left-hand side of R never reach any use. Expanding the program with a less precise reaching definition relation induces a spurious ϕ function, as in Figure 5.1.b. One may notice that the quast implementation in Figure 5.1.c is not really efficient and may be rather costly; but using classical optimizations such as loop peeling—or general polyhedron scanning techniques [AI91]—can significantly reduce this overhead, see Figure 5.1.d. This remark advocates once more for further studies about integrating optimization techniques.

```
  double A[N];
T A[0] = 0;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++) {
S     A[i+j] = ···;
R     A[i] = A[i+j-1] ···;
    }
```

Figure 5.1.a. Original program

```
  double A[N], A_T, A_S[N, N], A_R[N, N];
T A_T = 0;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++) {
S     A_S[i, j] = ···;
R     A_R[i, j] = φ({⟨T⟩} ∪ {⟨S, i', j'⟩ :
                     (i', j') <_LEX (i, j)})
                  ···
    }
```

Figure 5.1.b. SA without reaching definition analysis

```
  double A[N], A_T, A_S[N, N], A_R[N, N];
T A_T = 0;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++) {
S     A_S[i, j] = ···
R     A_R[i, j] = if (j==0) if (i==0) A_T else A_S[i-1, j]
                  else A_S[i, j-1]
                  ···;
    }
```

Figure 5.1.c. SA with precise reaching definition analysis

```
  double A[N], A_T, A_S[N, N], A_R[N, N];
  A_T = 0;
  A_S[1, 1] = ···;
  A_R[1, 1] = A_T ···;
  for (i=0; i<N; i++) {
    A_S[i, 1] = ···;
    A_R[i, 1] = A_S[i-1, 1] ···;
    for (j=0; j<N; j++) {
      A_S[i, j] = ···;
      A_R[i, j] = A_S[i, j-1] ···;
    }
  }
```

Figure 5.1.d. Precise reaching definition analysis plus loop peeling

Figure 5.1. Interaction of reaching definition analysis and run-time overhead

Eventually, one should notice that ϕ functions are not the only source of run-time overhead: computing reaching definitions using σ at run-time may also be costly, even when it is a function (i.e. it is exact). But there is a big difference between the two sources of overhead: run-time computation of σ can be costly because of the lack of expressiveness of control structures and algebraic operations in the language or because of the mathematical abstraction. For example, transductions generally induce more overhead than quasts. On the opposite, the overhead of ϕ functions is due to the *approximative* knowledge of the flow of data and its *non-deterministic* impact on the generated code; it is thus *intrinsic* to the expanded program, no matter how it is implemented. In many cases, indeed, the run-time overhead to compute σ can be significantly reduced by classical optimization techniques—an example will be presented later on Figure 5.1—but it is not the case for ϕ functions.

5.1.3 Single-Assignment for Loop Nests

In this section, we only consider intra-procedural expansion of programs operating on scalars and arrays. An extension to function calls, recursive programs and recursive data structures is studied at the end of this chapter, in Section 5.5. These restrictions simplify the exposition of a "concrete" SA algorithm in the classical loop nest framework.

When dealing with nest of loops, instancewise reaching definitions are described by an affine relation (see [BCF97, Bar98] and Section 2.4.3). We pointed in Section 3.1.1 that seeing an affine relation as a function, it can be written as a *nested conditional* called a quast [Fea91]. This representation of relation σ is especially interesting for expansion purposes since it can be easily and efficiently implemented in a programming language. Algorithm Make-Quast introduced in Section 3.1.1 builds a quast representation for any affine relation.

We use the following notations:

- Stmt($\langle S, x\rangle$) = S (the statement),
- Iter($\langle S, x\rangle$) = x (the iteration vector),
- and Array(S) is the name of the original data structure assigned by statement S.

Given a quast representation of reaching definitions, Convert-Quast generates an efficient code to retrieve the value read by some reference. This code is more or less a *compile-time implementation* of the conditional generated at the end of Abstract-SA. A ϕ function is generated when a non-singleton set is encountered. Eventually, because statements partition the set of memory locations in the single-assignment program, we use an array $\mathtt{A}_S$[x] instead of the proposed $\mathtt{A}_{\text{EXP}}\langle S, x\rangle$ in the abstract SA algorithm.

Thanks to Convert-Quast, we are ready to specialize Abstract-SA for loop nests. The new algorithm is Loop-Nests-SA. Current instance CurIns is implemented by its iteration vector (built from the surrounding loop variables). To simplify the exposition, scalars are seen are one-dimensional arrays of a single element. All memory accesses are thus performed through array subscripts.

The abstract code generation algorithm for ϕ functions can also be precised when dealing with loop nests and arrays only. For the same reason as before, run-time instances are stored in a distinct structure for each statement: we use $\Phi\mathtt{A}_S$[x] instead of $\Phi\mathtt{A}_{\text{EXP}}$[$\langle S, x\rangle$]. The new algorithm is Loop-Nests-Implement-Phi. Efficient computation of the lexicographic maximum can be done thanks to parallel reduction techniques [RF94].

One part of the code is still unimplemented: the array declaration. The main problem regarding array declaration is to get a compile-time evaluation of its size. In many cases, loop bounds are not easily predictable at compile-time. One may thus have to consider some expanded arrays as *dynamic arrays* whose size is updated at run-time. Another solution proposed by Collard [Col94b, Col95b] is to prefer a storage mapping optimization technique—such as the one presented in Section 5.3—to single-assignment form, and to

Convert-Quast $(quast, ref)$
quast: the quast representation of the reaching definition function
ref: the original reference, used when $\perp$ is encountered
returns the implementation of *quast* as a value retrieval code for reference *ref*

switch
  case $quast = \{\perp\}$:
    return ref
  case $quast = \{\imath\}$:
    $A \leftarrow$ Array$(\imath)$
    $S \leftarrow$ Stmt$(\imath)$
    $x \leftarrow$ Iter$(\imath)$
    return $\mathtt{A}_S\mathtt{[}x\mathtt{]}$
  case $quast = \{\imath_1, \imath_2, \ldots\}$:
    return $\phi(\{\imath_1, \imath_2, \ldots\})$
  case $quast =$ ***if*** *predicate* ***then*** $quast_1$ ***else*** $quast_2$:
    return `if` predicate Convert-Quast $(quast_1, ref)$
      `else` Convert-Quast $(quast_2, ref)$

Loop-Nests-SA $(program, \sigma)$
program: an intermediate representation of the program
σ: a reaching definition relation, seen as a function
returns an intermediate representation of the expanded program

for each array `A` **in** *program*
do for each statement S assigning `A` **in** *program*
  do declare an array $\mathtt{A}_S$
    left-hand side of S is replaced by $\mathtt{A}_S$`[`Iter(CurIns)`]`
  for each read reference *ref* to `A` **in** *program*
  do $\sigma_{/ref} \leftarrow \sigma \cap (\mathbf{I} \times ref)$
    $quast \leftarrow$ Make-Quast $(\sigma_{/ref})$
    $map \leftarrow$ Convert-Quast $(quast, ref)$
    $ref \leftarrow map$ `(CurIns)`
return *program*

fold the unbounded array into a bounded one when the associated memory reuse does not impairs parallelization. Such structures are very usual in high-level languages, but may result in poor performance when the compiler is unable to remove the run-time verification code. Two examples of code generation for ϕ functions are proposed in the next section.

5.1.4 Optimization of the Run-Time Overhead

Most of the run-time overhead comes from dynamic restoration of the data flow, using ϕ functions; and this cost is critical for non-scalar data structures distributed across processors. The technique presented in Section 5.2 (maximal static expansion) eradicates such run-time computations, to the cost of some loss in parallelism extraction. Indeed, ϕ functions may sometimes be a necessary condition for parallelization. This justifies the design of optimization techniques for ϕ function computation, which is the second purpose of this section.

We now present three optimizations to the code-generation algorithm in Section 5.1.2. The first method groups several basic optimizations for loop nests, the second one is based

Loop-Nests-Implement-Phi (*expanded*)
expanded: an intermediate representation of the expanded program
returns an intermediate representation with run-time restoration code

for each array $\mathtt{A}_S$ **in** *expanded*
do $d_A \leftarrow$ dimension of array $\mathtt{A}_S$
    $ref_S \leftarrow$ write reference in S
    if there are ϕ functions accessing $\mathtt{A}_S$
        then declare an array of d_A-dimensional vectors $\Phi\mathtt{A}_S$
            initialize $\Phi\mathtt{A}_S$ to `NULL`
            for each read access to $\mathtt{A}_S$ of the form $\phi(set)$ **in** *expanded*
            do if not already done for S
                then insert
                    $\Phi\mathtt{A}_S\mathtt{[}\textsc{Iter}\mathtt{(}\textsc{CurIns}\mathtt{)]} = f_e(\textsc{CurIns}, ref_S)$
                    immediately after S
for each original array `A` **in** *expanded*
do for each read access $\phi(set)$ `associated with A` **in** *expanded*
    do $\phi(set) \leftarrow$ `parallel for (each` S `in` $\textsc{Stmt}(set)$`)`
        $\mathtt{vector[}S\mathtt{]} = \mathtt{max}_{<_{\text{LEX}}}\{x : \langle S, x\rangle \in set \wedge \Phi\mathtt{A}_S\mathtt{[}x\mathtt{]} = f_e(\textsc{CurIns}, ref_\phi)\}$
        $\mathtt{instance} = \mathtt{max}_{<_{\text{SEQ}}}\{\langle S, \mathtt{vector[}S\mathtt{]}\rangle : S \in \textsc{Stmt}(set)\}$
        $\mathtt{A}_{\textsc{Stmt}(\mathtt{instance})}\mathtt{[}\textsc{Iter}\mathtt{(instance)]}$
return *expanded*

on a new instancewise analysis, and the last one avoid redundant computations during the propagation of "live" definitions. The second and third methods apply to loop nests and recursive programs as well.

First Method: Basic Optimizations for Loop Nests

When dealing with nests of loops, the Φ-structures are Φ-arrays indexed by iteration vectors (see Loop-Nests-Implement-Phi). Because of the hierarchical structure of loop nests, accesses in a set $\sigma(u)$ are very likely to share a few iteration vector components. This allows the removal of the associated dimensions in Φ-arrays and to reduce the complexity of lexicographic maximum computations. Another consequence is the applicability of up-motion techniques for invariant assignments. An example of Φ-array simplification and up-motion is described in Figure 5.2, where function `max` computes the maximum of a set of iteration vectors, and where the maximum of an empty set is the vector $(-\infty, \ldots, -\infty)$.

Another interesting optimization is only applicable to `while` loops and `for` loops whose termination condition is complex: non-affine bounds, `break` statements or exceptions. When a loop assigns the same memory location an unbounded number of times, conversion to single-assignment form often requires a ϕ-function but the last defining write can be computed without using Φ-arrays: its iteration vector is associated with the last value of the loop counter.[2] An example is described in Figure 5.3.

[2] The semantics of the resulting code is correct, but rather dirty: a loop variable is used outside of the loop block.

```
    double x;
    for (i=1; i<=N; i++) {
      for (j=1; j<=N; j++)
        if (···)
          for (k=1; k<=N; k++)
S           x = ···;
R     ··· = x;
    }
```

Figure 5.2.a. Original program

```
    double x, x_S[N+1, N+1, N+1];
    for (i=1; i<=N; i++) {
      for (j=1; j<=N; j++)
        if (···)
          for (k=1; k<=N; k++)
S           x_S[i, j, k] = ···;
R     ··· = φ({⟨S, i, j', N⟩ : 1 ≤ j' ≤ N} ∪ {⊥});
    }
```

Figure 5.2.b. SA program

```
    double x, x_S[N+1, N+1, N+1], Φx_S[N+1, N+1, N+1]={NULL};
    for (i=1; i<=N; i++) {
      for (j=1; j<=N; j++)
        if (···)
          for (k=1; k<=N; k++) {
S           x_S[i, j, k] = ···;
            Φx_S[i, j, k] = &x;
          }
R     ··· = {
        max_S = max {(i, j', k') : 1 ≤ j' ≤ N ∧ k' = N ∧ Φx_S[i, j', k'] = &x};
        if (max_S != (−∞, −∞, −∞)) x_S[max_S] else x;
      }
    }
```

Figure 5.2.c. Standard ϕ implementation

```
    double x, x_S[N+1, N+1, N+1], Φx_S[N+1]={NULL};
    for (i=1; i<=N; i++) {
      for (j=1; j<=N; j++) {
        if (···) {
          for (k=1; k<=N; k++) {
S           x_S[i, j, k] = ···;
          Φx_S[j] = &x;
        }
R     ··· = {
        max_S = max {j' : 1 ≤ j' ≤ N ∧ Φx_S[j'] = &x};
        if (max_S != −∞) x_S[max_S] else x;
      }
    }
```

Figure 5.2.d. Optimized ϕ implementation

Figure 5.2. Basic optimizations of the generated code for ϕ functions

Second Method: Improving the Single-Assignment Form Algorithm

In some cases, ϕ functions can be computed without Φ-arrays to store possible reaching definitions. When the read statement is too complex to be analyzed at compile-time,

```
   double x;
   while (···)
S    x = ···;
R  ··· = x;
```

Figure 5.3.a. Original program

```
   double x, x_S[···];
   w = 1;
   while (···) {
S    x_S[w] = ···;
     w++;
   }
R  ··· = φ({⟨S, w⟩ : 1 ≤ w} ∪ {⊥});
```

Figure 5.3.b. SA program

```
   double x, x_S[···], Φx_S[···]={NULL};
   w = 1;
   while (···) {
S    x_S[w] = ···;
     Φx_S[w] = &x;
     w++;
   }
R  ··· = {
     max_S = max {w : Φx_S[w] = &x};
     if (max_S != −∞) x_S[max_S] else x;
   }
```

Figure 5.3.c. Standard ϕ implementation

```
   double x, x_S[···];
   w = 1;
   while (···) {
S    x_S[w] = ···;
     w++;
   }
R  ··· = if (w>1) x_S[w-1] else x;
```

Figure 5.3.d. Optimized ϕ implementation

Figure 5.3. Repeated assignments to the same memory location

the set of possible reaching definitions can be very large. However, if we could compute the very memory location accessed by the read statement, the set of possible reaching definitions would be much smaller—sometimes reduced to a singleton. This shows the need for an additional instancewise information, called *reaching definition of a memory location*: the exact function which depends on an execution $e \in \mathbf{E}$ of the program is denoted by σ_e^{ML} and its conservative approximation by σ^{ML}. Here are the formal definitions:

$$\forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e, c \in f_e(\mathbf{W}_e): \quad \sigma_e^{\mathrm{ML}}(u, c) = \max_{<_{\mathrm{SEQ}}} \big\{ v \in \mathbf{W}_e : v <_{\mathrm{SEQ}} u \wedge f_e(v) = c \big\},$$

$$\forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e, c \in f_e(\mathbf{W}_e): \quad v = \sigma_e^{\mathrm{ML}}(u, c) \Longrightarrow v \in \sigma^{\mathrm{ML}}(u, c).$$

Computing relation σ^{ML} is not really different from reaching definition analysis. To compute the σ^{ML} for a reference r in right-hand side of a statement, r is replaced by a read access to a new symbolic memory location c, then classical instancewise reaching definition analysis is performed. The result is a reaching definition relation parameterized by c. Seeing c as an argument, it yields the expected approximate relation σ^{ML}. In some rare cases, this computation scheme yields unnecessary complex results:[3] the general solution is then to intersect the result with σ.

Algorithm Abstract-ML-SA is an improved single-assignment form conversion algorithm based on reaching definitions of memory locations. It is based on the *exact*

[3] Consider an array `A`, an assignment to `A[foo]` and a read reference to `A[foo]`, where *foo* is some complex subscript. A precise reaching definition analysis would compute an exact result because the subscript is the same in the two statements. However, the reaching definition of a given memory location is not known precisely, because *foo* in the assignment statement is not known at compile time.

run-time computation of the symbolic memory location with storage mapping f_e. This algorithm can also been specialized for loop nests and arrays, using quasts parameterized by the current instance and the symbolic memory location, see LOOP-NESTS-ML-SA. In both cases, the value of f_e should *not* be interpreted, it must be used as the original reference code—possibly complex—to be substituted to the symbolic memory location c. An example is described in Figure 5.4.

```
    double A[N+1];
    for (i=1; i<=N; i++)
      for (j=1; j<=N; j++)
S       A[j] = A[j] + A[foo];
```

Figure 5.4.a. Original program

```
    double A[N+1], A_S[N+1, N+1];
    for (i=1; i<=N; i++)
      for (j=1; j<=N; j++)
S       A_S[j] = if (i>1) A_S[i-1, j] else A[j]
          + if (i>1 || j>1) φ({⊥} ∪ {⟨S, i', j'⟩ : 1 ≤ i', j' ≤ N ∧ (i', j') <_LEX (i, j)})
            else A[foo];
```

Figure 5.4.b. SA program

```
    double A[N+1], A_S[N+1, N+1];
    for (i=1; i<=N; i++)
      for (j=1; j<=N; j++)
S       A_S[j] = if (i>1) A_S[i-1, j] else A[j]
          + if (foo<j) A_S[i, foo]
            else if (i>1) A_S[i-1, foo] else A[foo];
```

Figure 5.4.c. SA program with reaching definitions of memory locations

Figure 5.4. Improving the SA algorithm

Third Method: Cheating with Single-Assignment

A general problem with implementations of ϕ functions based on Φ-structures is the large redundancy of lexicographic maximum computations. Indeed, each time a ϕ function is encountered, the maximum of the full set of possible reaching definitions must be computed. In the *static* single-assignment framework (SSA) [CFR+91, KS98], a large part of the work is devoted to optimized *placement* of ϕ functions, in order to never recompute the maximum of the same set. These techniques are well suited to the variable renaming involved in SSA, but are unable to support the data structure reconstruction performed by SA algorithms. Nevertheless, for another expansion scheme presented in Section 5.4.7, we are able to avoid redundancies and to optimize the placement of ϕ functions, but the algorithm is rather complex.

The method we propose here has been studied with the help of Laurent Vibert. It removes redundant computations, but computation is not made with Φ-structures in SA

ABSTRACT-ML-SA $(program, \mathbf{W}, \sigma^{\text{ML}})$
    program: an intermediate representation of the program
    $\mathbf{W}$: a conservative approximation of the set of write accesses
    σ^{ML}: reaching definitions of memory locations
    returns an intermediate representation of the expanded program
    for each data structure `D` **in** *program*
    do declare a data structure $\texttt{D}_{\text{EXP}}[\mathbf{W}]$
        for each statement s assigning `D` **in** *program*
        do left-hand side of $s \leftarrow \texttt{D}_{\text{EXP}}[\text{CURINS}]$
        for each reference ref to `D` **in** *program*
        do $ref \leftarrow$ `if` $(\sigma^{\text{ML}}((\texttt{CURINS}, ref), f_e(\texttt{CURINS}, ref))$`=`$\{\bot\})$ ref
                `else if` $(\sigma^{\text{ML}}((\texttt{CURINS}, ref), f_e(\texttt{CURINS}, ref))$`==`$\{\imath\})$ $\texttt{D}_{\texttt{EXP}}[\imath]$
                `else` $\phi(\sigma^{\text{ML}}((\texttt{CURINS}, ref), f_e(\texttt{CURINS}, ref)))$
    return *program*

LOOP-NESTS-ML-SA $(program, \sigma^{\text{ML}})$
    program: an intermediate representation of the program
    σ^{ML}: reaching definitions of memory locations
    returns an intermediate representation of the expanded program
    for each array `A` **in** *program*
    do for each statement S assigning `A` **in** *program*
        do declare an array $\texttt{A}_S$
            left-hand side of $S \leftarrow \texttt{A}_S[\texttt{ITER}(\texttt{CURINS})]$
        for each reference ref to `A` **in** *program*
        do $\sigma^{\text{ML}}_{/ref} \leftarrow \sigma^{\text{ML}} \cap (\mathbf{I} \times ref)$
            $u \leftarrow$ symbolic access associated with reference ref
            $quast \leftarrow$ MAKE-QUAST $(\sigma^{\text{ML}}_{/ref}(u, f_e(u)))$
            $map \leftarrow$ CONVERT-QUAST $(quast, ref)$
            $ref \leftarrow map$ `(CURINS)`
    return *program*

form: it is based on *@-structures* whose semantics is similar to @-arrays in the *static* single-assignment (SSA) framework [KS98]. This is a simple compromise between dependence removal and efficient computation of ϕ functions, based on the commutativity and associativity of the lexicographic maximum. The idea is to use @-structures in one-to-one mapping with the *original* data structures instead of the expanded ones. Notice @-structures are *not* in *single-assignment* form, and maximum computation must be done in a *critical section*. Both the write instance and the memory location should be stored, but the memory location is now encoded in the subscript: @-structures are thus *storing instances instead of memory locations*, see ABSTRACT-IMPLEMENT-PHI-NOT-SA.

The original memory-based dependences are displaced from the original data structures to their @-structures: they have not disappeared! However, thanks to the properties of the lexicographic maximum, output dependences can be ignored without violating the original program semantics. Spurious anti-dependences remain, and must be taken into account for parallelization purposes. The first example in Figure 5.5 can be parallelized with this technique, but not the second.

In the case of loop nests and arrays, a simple extension to the technique can be helpful. It is sufficient, for example, to parallelize the second example in Figure 5.5. Consider a call of the form $\phi(set)$. If the component value of some dimensions is constant for all

Abstract-Implement-Phi-Not-SA (*expanded*)
expanded: an intermediate representation of the expanded program
returns an intermediate representation with run-time restoration code
for each original data structure `D`[*shape*] **in** *expanded*
do if there are ϕ functions accessing $\mathtt{D}_{\text{EXP}}$
  then declare a data structure `@D`[*shape*] initialized to $\perp$
    for each read reference ref_ϕ to `D` whose expanded form is $\phi(set)$
    do $sub_\phi \leftarrow$ subscript of reference ref_ϕ
      for each statement s involved in set
      do $sub_s \leftarrow$ subscript of the write reference to `D` in s
        if not already done for s
          then following s insert `@D`[sub_s] `= max (@D`[sub_s]`,` CurIns`)`
      $\phi(set) \leftarrow$ `if (@D`[sub_ϕ]`!=`$\perp$`)` $\mathtt{D}_{\text{EXP}}$`[@D] else D`[sub_ϕ]
return *expanded*

iteration vectors of instances in *set*, then it is legal to expand the `@`-array along these dimensions. Applied to the second example in Figure 5.5, `@x` is replaced by `@x[i]`, which makes the outer loop parallel.

```
  double x;
  for (i=1; i<=N; i++)
S   if (···) x = ···;
R ··· = x;
```

Figure 5.5.a. First example

```
  double x, x_S[N+1], @x=−∞;
  parallel for (i=1; i<=N; i++)
S   if (···) {
      x = ···;
      @x = max (@x, i);
    }
R ··· = if (@x != −∞) x_S[@x]
        else x;
```

Figure 5.5.b. First example: parallel expansion

```
  double x;
  for (i=1; i<=N; i++) {
T   x = ···;
    for (j=1; j<=N; j++)
S     if (···) x = x ···;
R   ··· = x;
  }
```

Figure 5.5.c. Second example

```
  double x, x_T[N+1], x_S[N+1, N+1];
  double @x=(−∞,−∞);
  for (i=1; i<=N; i++) {
T   x_T[i] = ···;
    for (j=1; j<=N; j++)
S     if (···) {
        x_S[i, j] = if (j>1) x_S[i, j-1]
                    else x_T[i] ···;
        @x = max (@x, (i, j));
      }
R   ··· = if (@x != (−∞,−∞)) x_S[@x]
          else x_T[i];
  }
```

Figure 5.5.d. Second example: not parallelizable expansion

Figure 5.5. Parallelism extraction versus run-time overhead

In practice, this technique is both very easy to implement and very efficient for run-time restoration of the data flow, but it can often hamper parallelism extraction. It is a first and simple attempt to find a tradeoff between parallelism and overhead.

5.1.5 Tradeoff between Parallelism and Overhead

All the single-assignment form algorithms described and most techniques for run-time restoration of the data flow share the same major drawback: run-time overhead. By essence, SA form requires a huge memory usage, and is not practical for real programs. Moreover, some ϕ functions cannot be implemented efficiently with the optimizations proposed. To avoid or reduce these sources of run-time overhead, it is thus necessary to design more pragmatic expansion schemes: both memory usage and run-time data-flow restoration code should be handled with care. This is the purpose of the three following sections.

5.2 Maximal Static Expansion

The present section studies a novel memory expansion paradigm: its motivation is to stick with the compile-time restoration of the flow of data while keeping in mind the approximative nature of the compile-time information. More precisely, we would like to remove as many memory-based dependences as possible, without the need of any ϕ function (associated with run-time restoration of the data-flow). We will show that this goal requires a change in the way expanded data structures are accessed, to take into account the approximative knowledge of storage mappings.

An expansion of data structures that does not need a ϕ function is called a *static expansion* [BCC98, BCC00].[4] The goal is to find automatically a *static* way to expand 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 algorithm to derive the maximal static expansion; its input is the (perhaps conservative) output of a reaching definition analysis, so our method is "optimal" with respect to the precision of this analysis. Our framework is valid for any imperative program, without restriction—the only restrictions being those of your favorite reaching definition analysis. We then present an *intra-procedural* algorithm to construct the maximal static expansion for programs with arrays and scalars only, but where subscripts and control structures are unrestricted.

5.2.1 Motivation

The three following examples introduce the main issues and advocate for a maximal static expansion technique.

First Example: Dynamic Control Flow

We first study the pseudo-code shown in Figure 5.6; this kernel appears in several convolution codes[5]. Parts denoted by $\cdots$ are supposed to have no side-effect.

[4] Notice that according to our definition, an expansion in the *static single-assignment* framework [CFR+91, KS98] may *not* be static.

[5] For instance, Horn and Schunck's algorithm to perform 3D Gaussian smoothing by separable convolution.

```
    double x;
    for (i=1; i<=N; i++) {
T     x = ···;
      while (···)
S       x = x ···;
R     ··· = x ···;
    }
```

Figure 5.6. First example

Each instance $\langle T, i\rangle$ assigns a new value to variable `x`. In turn, statement S assigns `x` an undefined number of times (possibly zero). The value read in `x` by statement R is thus *defined either* by T, *or* by some instance of S, in *the same iteration* of the `for` loop (the same i). Therefore, if the expansion assigns distinct memory locations to $\langle T, i\rangle$ and to instances of $\langle S, i, w\rangle$,[6] how could instance $\langle R, i\rangle$ "know" which memory location to read from?

We have already seen that this problem is solved with an *instancewise reaching definition analysis* which describe where values are defined and where they are used. We may thus call σ the mapping from a read instance to its *set of possible reaching definitions*. Applied to the example in Figure 5.6, it tells us that the set $\sigma(\langle S, i, w\rangle)$ of definitions reaching instance $\langle S, i, w\rangle$ is:

$$\sigma(\langle S, i, w\rangle) = \textbf{\textit{if}}\ \ w > 1\ \ \textbf{\textit{then}}\ \ \{\langle S, i, w-1\rangle\}\ \ \textbf{\textit{else}}\ \ \{\langle T, i\rangle\} \tag{5.1}$$

And the set $\sigma(\langle R, i\rangle)$ of definitions reaching instance $\langle R, i\rangle$ is:

$$\sigma(\langle R, i\rangle) = \big\{\langle T, i\rangle\big\} \cup \big\{\langle S, i, w\rangle : w \geq 1\big\}, \tag{5.2}$$

where w is an artificial counter of the `while`-loop.

Let us try to expand scalar `x`. One way is to convert the program into SA, making T write into $\mathtt{x}_T$`[i]` and S into $\mathtt{x}_S[i, w]$: then, each memory location 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 (5.2) yields the program in Figure 5.7. While the right-hand side of S only depends on w, the right-hand side of R depends on the control flow, thus needing a ϕ function.

The aim of maximal static expansion is to expand `x` as much as possible in this program *but* without having to insert ϕ functions.

A possible static expansion is to uniformly expand `x` into `x[i]` and to avoid output dependencies between distinct iterations of the `for` loop. Figure 5.8 shows the resulting *maximal static expansion* of this example. It has the same degree of parallelism and is simpler than the program in single-assignment.

Notice that it should be easy to adapt the array privatization techniques by Maydan *et al.* [MAL93] to handle the program in Figure 5.6; this would tell us that `x` can be privatized along i. However, we want to do more than privatization along loops, as illustrated in the following examples.

[6] We need a virtual loop variable w to track iterations of the `while` loop.

```
   for (i=1; i<=N; i++) {
T    x_T[i] = ···
     w = 1;
     while (···) {
S      x_S[i,w] = if (w==1) x_T[i] else x_S[i,w-1] ···
       w++;
     }
R    ··· = φ({⟨T,i⟩} ∪ {⟨S,i,w⟩ : w ≥ 1}) ···
   }
```

Figure 5.7. First example, continued

```
   for (i=1; i<=N; i++) {
T    x[i] = ···
     while (···)
S      x[i] = x[i] ···
R    ··· = x[i] ···
   }
```

Figure 5.8. Expanded version of the first example

Second Example: Array Expansion

Let us give a more complex example; we would like to expand array `A` in the program in Figure 5.9.

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 5.9 describes the data-flow 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).

```
   double A[4*N];
   for (i=1; i<=2*N; i++)
     for (j=1; j<=2*N; j++) {
       if (···)
S        A[i-j+2*N] = ··· A[i-j+2*N] ···;
T      if (j==N) A[i+N] = ···;
     }
```

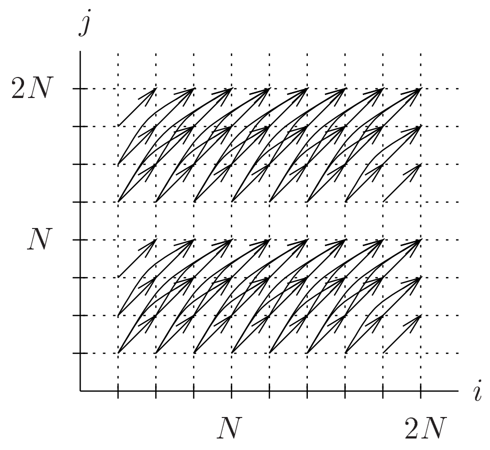


Figure 5.9. Second example

Formally, the definition reaching an instance of statement S is:[7]

$$\sigma(\langle S,i,j\rangle) = \left|\begin{array}{l} \textbf{\textit{if }} j \le N \\ \textbf{\textit{then }} \left\{\langle S,i',j'\rangle : 1 \le i' \le 2N \wedge 1 \le j' < j \wedge i'-j' = i-j\right\} \\ \textbf{\textit{else}} \left|\begin{array}{l} \left\{\langle S,i',j'\rangle : 1 \le i' \le 2N \wedge N < j' < j \wedge i'-j' = i-j\right\} \\ \cup \left\{\langle T,i',N\rangle : 1 \le i' < i \wedge i' = i-j+N\right\} \end{array}\right. \end{array}\right. \tag{5.3}$$

Because reaching definitions are non-singleton sets, converting this program to SA form would require run-time computation of the memory location read by S.

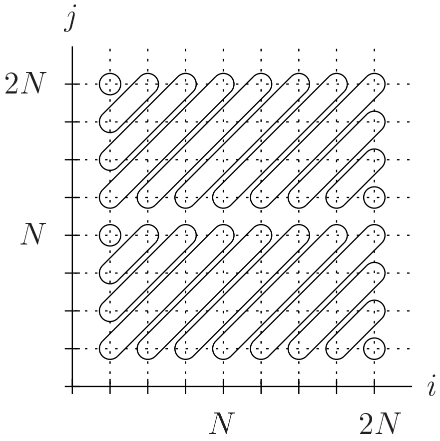


Figure 5.10.a. Instances involved in the same data flow

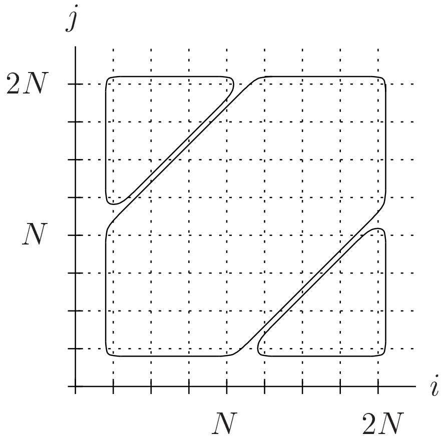


Figure 5.10.b. Counting groups per memory location

Figure 5.10. Partition of the iteration domain ($N = 4$)

However, we notice that the iteration domain of S may be split into *disjoint subsets* by grouping together instances involved in the same data flow. These subsets build a partition of the iteration domain. Each subset may have its own memory space that will not be written nor read by instances outside the subset. The partition is given in Figure 5.10.a.

Using this property, we can duplicate only those elements of `A` that appear in two distinct subsets. These are all the array elements `A[`c`]`, $1 + N \le c \le 3N - 1$. They are accessed by instances in the large central set in Figure 5.10.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 S_2 and S_3 in Figure 5.11, respectively. Elements `A[`c`]`, $1 \le c \le N$ are accessed by instances in the upper left triangle in Figure 5.10.b and have only one subset each (one subset in the corresponding diagonal in Figure 5.10.a), which we label with 1. The same labeling holds for sets corresponding to instances in the lower right triangle.

The maximal static expansion is shown in Figure 5.11. Notice that this program has the same degree of parallelism as the corresponding single-assignment program, without the run-time overhead.

```
   double A[4*N, 2];
   for (i=1; i<=2*N; i++)
     for (j=1; j<=2*N; j++) {
       // expansion of statement S
       if (-2*N+1<=i-j && i-j<=-N) {
         if (···)
S1         A[i-j+2*N, 0] = ··· A[i-j+2*N, 1] ···;
       } else if (-N+1<=i-j && i-j<=N-1) {
         if (j<=N) {
           if (···)
S2           A[i-j+2*N, 0] = ··· A[i-j+2*N, 0] ···;
         } else
           if (···)
S3           A[i-j+2*N, 1] = ··· A[i-j+2*N, 1] ···;
       } else
         if (···)
S4         A[i-j+2*N, 0] = ··· A[i-j+2*N, 0] ···;
       // expansion of statement T
T      if (j==N) A[i+N, 2] = ···;
     }
```

Figure 5.11. Maximal static expansion for the second example

```
   double A[N+1];
   for (i=1; i<=N; i++) {
     for (j=1; j<=N; j++)
T      A[j] = ···;
S    A[foo(i)] = ···;
R    ··· = ··· A[bar(i)];
   }
```

Figure 5.12.a. Source program

```
   double A[N+1, N+1];
   for (i=1; i<=N; i++) {
     for (j=1; j<=N; j++)
T      A[j, i] = ···;
S    A[foo(i), i] = ···;
R    ··· = ··· A[bar(i), i];
   }
```

Figure 5.12.b. Expanded version

Figure 5.12. Third example

Third Example: Non-Affine Array Subscripts

Consider the program in Figure 5.12.a, where *foo* and *bar* are arbitrary subscripting functions[8]. Since all array elements are assigned by T, the value read by R at the i^{th} iteration must have been produced by S or T at the same iteration. The data-flow graph

[7]Some instances of S read uninitialized values (e.g. when $j = 1$) and they have no reaching definition. As a consequence, the expanded program in Figure 5.11 shoud begin with a *copy-in* code from the original array to the expanded one.

[8]`A[foo(i)]` stands for an array subscript between 1 and N, "too complex" to be analyzed at compile-time.

is similar to the first example:

$$\sigma(\langle R, i\rangle) = \big\{\langle S, i\rangle\big\} \cup \big\{\langle T, i, j\rangle : 1 \le j \le N\big\}. \tag{5.4}$$

The maximal static expansion adds a new dimension to `A` subscripted by i. It is sufficient to make the first loop parallel.

What Next?

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 convenient property of our algorithm.

It is natural to compare *array privatization* [MAL93, TP93, Cre96, Li92] and maximal static expansion: both methods expose parallelism in programs at a lower cost than single-assignment form transformation. However, privatization generally resorts to dynamic restoration of the data flow, and it only detects parallelism along the enclosing loops; it is thus less powerful than general array expansion techniques. Indeed, the example in Section 5.2.1 shows that our method not only may expand along diagonals in the iteration space but may also do some "blocking" along these diagonals.

5.2.2 Problem Statement

We assume an instancewise reaching definition analysis is performed previously, yielding a conservative approximation σ of the relation between *uses* and *reaching definitions*.

The definition of *static expansion* has first been introduced in [BCC98]: the idea is to *avoid dynamic restoration of the data flow*. Let us consider two writes v and w belonging to the same set of reaching definitions of some read u. Suppose they both write in the same memory location. If we assign two distinct memory locations to v and w in the expanded program, then a ϕ function is needed to restore the data flow, since we do not know which of the two locations has the value needed by u. Using the notations introduced in Sections 2.4 and 2.5, "v and w write in the same memory location" is denoted by $f_e(v) = f_e(w)$, and "u and w are assigned distinct memory locations in the expanded program" is denoted by $f_e^{\text{EXP}}(v) \ne f_e^{\text{EXP}}(w)$.

We introduce relation $\mathfrak{R}$ between definitions that possibly reach the same read (recall that we do not require the reaching definition analysis to give exact results):

$$\forall v, w \in \mathbf{W}: \quad v\,\mathfrak{R}\,w \iff \exists u \in \mathbf{R} : v\,\sigma\,u \wedge w\,\sigma\,u.$$

Whenever two definitions possibly reaching the same read assign the same memory location in the original program, they must still assign the same memory location in the expanded program. Since "writing in the same memory location" is an equivalence relation, we actually use $\mathfrak{R}^*$, the transitive closure of $\mathfrak{R}$ (see Section 5.2.4 for computation details). Relation $\mathfrak{R}^*$, therefore, generalizes webs [Muc97] to instances of references, and the rest of this work shows how to compute $\mathfrak{R}^*$ in the presence of arrays.[9]

[9] Strictly speaking, webs include definitions *and* uses, whereas $\mathfrak{R}^*$ applies to definitions only.

Relation $\mathfrak{R}$ holds between definitions that reach the same use. Therefore, mapping these writes to different memory locations is precisely the case where ϕ functions would be necessary, a case a static expansion is designed to avoid:

Definition 5.1 (static expansion) For an execution $e \in \mathbf{E}$ of the program, an expansion from storage mapping f_e to storage mapping f_e^{EXP} is static if

$$\forall v, w \in \mathbf{W}_e : \quad v \,\mathfrak{R}^*\, w \wedge f_e(v) = f_e(w) \implies f_e^{\text{EXP}}(v) = f_e^{\text{EXP}}(w). \tag{5.5}$$

When clear from the context, we say "static expansion f_e^{EXP}" instead of "static expansion from f_e to f_e^{EXP}". Now, we are interested in removing as many dependences as possible, without introducing ϕ functions. We are looking for the *maximal static expansion* (MSE), assigning the largest number of memory locations while verifying (5.5):

Definition 5.2 (maximal static expansion) For an execution e, a static expansion f_e^{EXP} is maximal on the set $\mathbf{W}_e$ of writes, if for any *static* expansion f'_e,

$$\forall v, w \in \mathbf{W}_e : \quad f_e^{\text{EXP}}(v) = f_e^{\text{EXP}}(w) \implies f'_e(v) = f'_e(w). \tag{5.6}$$

Intuitively, if f_e^{EXP} is maximal, then f'_e cannot do better: it maps two writes to the same memory location when f_e^{EXP} does.

We need to characterize the sets of statement instances on which a maximal static expansion f_e^{EXP} is constant, i.e. equivalence classes of relation $\{u, v \in \mathbf{W}_e : f_e^{\text{EXP}}(u) = f_e^{\text{EXP}}(v)\}$. However, this hardly gives us an expansion scheme, because this result does not tell us how much each individual memory location should be expanded. The purpose of Section 5.2.3 is to design a practical expansion algorithm for each memory location used in the original program.

5.2.3 Formal Solution

Following the lines of [BCC00], we are interested in the *static expansion* which removes the largest number of dependences.

Proposition 5.1 (maximal static expansion) Given a program execution e, a storage mapping f_e^{EXP} is both a maximal static expansion of f_e and finer than f_e if and only if

$$\forall v, w \in \mathbf{W}_e : \quad v \,\mathfrak{R}^*\, w \wedge f_e(v) = f_e(w) \iff f_e^{\text{EXP}}(v) = f_e^{\text{EXP}}(w) \tag{5.7}$$

Proof: Sufficient condition—the "if" part

Let f_e^{EXP} be a mapping s.t. $\forall u, v \in \mathbf{W} : f_e^{\text{EXP}}(u) = f_e^{\text{EXP}}(v) \Leftrightarrow u \,\mathfrak{R}^*\, v \wedge f_e(u) = f_e(v)$. By definition, f_e^{EXP} is a static expansion and f_e^{EXP} is finer than f_e.

Let us show that f_e^{EXP} is maximal. Suppose that for $u, v \in \mathbf{W}$: $f_e^{\text{EXP}}(u) = f_e^{\text{EXP}}(v)$. (5.7) implies $u \,\mathfrak{R}^*\, v$ and $f_e(u) = f_e(v)$. Thus, from (5.5), any other static expansion f'_e satisfies $f'_e(u) = f'_e(v)$ too. Hence, $f_e^{\text{EXP}}(u) = f_e^{\text{EXP}}(v) \Rightarrow f'_e(u) = f'_e(v)$, so f_e^{EXP} is maximal.

Necessary condition—the "only if" part

Let f_e^{EXP} be a maximal static expansion finer than f_e. Because f_e^{EXP} is a static expansion, we only have to prove that

$$\forall u, v \in \mathbf{W} : \quad f_e^{\text{EXP}}(u) = f_e^{\text{EXP}}(v) \implies u \,\mathfrak{R}^*\, v \wedge f_e(u) = f_e(v).$$

On the one hand, $f_e^{\text{EXP}}(u) = f_e^{\text{EXP}}(v) \Rightarrow f_e(u) = f_e(v)$ because f_e is finer than f_e. On the other hand, for some u and v in $\mathbf{W}$, assume $f_e^{\text{EXP}}(u) = f_e^{\text{EXP}}(v)$ and $\neg(u \,\mathfrak{R}^*\, v)$. We show that it contradicts the maximality of f_e^{EXP}: for any w in $\mathbf{W}$, let $f'_e(w) = f_e^{\text{EXP}}(w)$ when $\neg(u \,\mathfrak{R}^*\, w)$, and $f'_e(w) = c$ when $u \,\mathfrak{R}^*\, w$, for some $c \neq f_e^{\text{EXP}}(u)$. f'_e is a static expansion: By construction, $f'_e(u') = f'_e(v')$ for any u' and v' such that $u' \,\mathfrak{R}^*\, v'$. The contradiction comes from the fact that $f'_e(u) \neq f'_e(v)$. ■

Results above make use of a general memory expansion f_e^{EXP}. However, constructing it from scratch is another issue. To see why, consider a memory location c and two accesses v and w writing into c. Assume that $v \,\mathfrak{R}^*\, w$: these accesses *must* assign the same memory location in the expanded program. Now assume the contrary: if $\neg(v \,\mathfrak{R}^*\, w)$, then the expansion should make them assign two distinct memory locations.

We are thus strongly encouraged to choose an expansion f_e^{EXP} of the form (f_e, ν) where function ν is constructed by the analysis and must be *constant* on equivalence classes of $\mathfrak{R}^*$. Notation (f_e, ν) is merely abstract. A concrete method for code generation involves adding dimensions to arrays, and extending array subscripts with ν, see Section 5.2.4.

Now, a storage mapping $f_e^{\text{EXP}} = (f_e, \nu)$ is finer than f_e by construction, and it is a maximal static expansion if function ν satisfies the following equation:

$$\forall e \in \mathbf{E}, \forall v, w \in \mathbf{W}_e, f_e(v) = f_e(w): \quad v \,\mathfrak{R}^*\, w \iff \nu(v) = \nu(w).$$

In practice, $f_e(v) = f_e(w)$ can only be decided when f_e is affine. In general, we have to approximate f_e with relation κ and derive two constraints from the previous equation:

$$\text{Expansion must be static: } \forall v, w \in \mathbf{W}: \quad v \,\kappa\, w \wedge v \,\mathfrak{R}^*\, w \Longrightarrow \nu(v) = \nu(w); \tag{5.8}$$

$$\text{Expansion must be maximal: } \forall v, w \in \mathbf{W}: \quad v \,\kappa\, w \wedge \neg(v \,\mathfrak{R}^*\, w) \Longrightarrow \nu(v) \neq \nu(w). \tag{5.9}$$

First, notice that changing κ into its transitive closure κ^* has no impact on (5.8), and that the transformed equation yields an equivalence class enumeration problem. Second, (5.9) is a graph coloring problem: it says that two writes cannot "share the same color" if related. Direct methods exists to address these two problems simultaneously (see [Coh99b] or Section 5.4), but they seem much two complicated for our purpose.

Now, the only purpose of relation κ is to avoid unnecessary memory allocation, and using a conservative approximation harms neither the maximality not the static property of the expansion. Actually, we found that relation κ differs from κ^*—meaning κ is not transitive—only in contrived examples, e.g. with tricky combinations of affine and non-affine array subscripts. Therefore, consider the following *maximal static expansion* criterion:

$$\forall v, w \in \mathbf{W}, v \,\kappa^*\, w: \quad v \,\mathfrak{R}^*\, w \iff \nu(v) = \nu(w) \tag{5.10}$$

Now, given an equivalence class of κ^*, classes of $\mathfrak{R}^*$ are exactly the sets where storage mapping f_e^{EXP} is constant:

Theorem 5.1 A storage mapping $f_e^{\text{EXP}} = (f_e, \nu)$ is a maximal static expansion for all execution $e \in \mathbf{E}$ iff for each equivalence class $\mathbf{C} \in \mathbf{W}/_{\kappa^*}$, ν is constant on each class in $\mathbf{C}/_{\mathfrak{R}^*}$ and takes distinct values between different classes: $\forall v, w \in \mathbf{C}: \ v \,\mathfrak{R}^*\, w \Leftrightarrow \nu(v) = \nu(w)$.

Proof: $\mathbf{C} \in \mathbf{W}/_{\kappa^*}$ denotes a set of writes which may assign the same memory cell, and $\mathbf{C}/_{\mathfrak{R}^*}$ is the set of equivalence classes for relation $\mathfrak{R}^*$ on writes in $\mathbf{C}$. A straightforward application of (5.10) concludes the proof. ■

Notice that ν is only supposed to take different values between classes in the same $\mathbf{C}$: if $\mathbf{C}_1, \mathbf{C}_2 \in \mathbf{W}/_{\mathcal{K}^*}$ with $\mathbf{C}_1 \neq \mathbf{C}_2$, $u_1 \in \mathbf{C}_1$ and $u_2 \in \mathbf{C}_2$, nothing prevents that $\nu(u_1) = \nu(u_2)$.

As a consequence, two maximal static expansions f_e^{EXP} and f'_e are identical on a class of $\mathbf{W}/_{\mathcal{K}^*}$, up to a one-to-one mapping between constant values. An interesting result follows:

Lemma 5.1 The *expansion factor* for each memory location assigned by writes in $\mathbf{C}$ is $\text{Card}(\mathbf{C}/_{\mathfrak{R}^*})$.

Let $\mathbf{C}$ be an equivalence class in $\mathbf{W}/_{\mathcal{K}^*}$ (statement instances that may hit the same memory location). Suppose we have a function ρ mapping each write u in $\mathbf{C}$ to a representative of its equivalence class in $\mathbf{C}$ (see Section 5.2.4 for details). One may label each class in $\mathbf{C}/_{\mathfrak{R}^*}$, or equivalently, label each element of $\rho(\mathbf{C})$. Such a labeling scheme is obviously arbitrary, but all programs transformed using our method are equivalent up to a permutation of these labels. Labeling boils down to scanning exactly once all the integer points in the set of representatives $\rho(\mathbf{C})$, see Section 5.2.5 for details. Now, remember that function f_e^{EXP} is of the form (f_e, ν). From Theorem 5.1, we can take for $\nu(u)$ the label we choose for $\rho(u)$, then storage mapping f_e^{EXP} is a maximal static expansion for our program.

Eventually, one has to generate code for the expanded program, using storage mapping f_e^{EXP}. It is done in Section 5.2.4.

5.2.4 Algorithm

The maximal static expansion scheme given above works for any imperative program. More precisely, you may expand any imperative program using maximal static expansion, provided that a reaching definition analysis technique can handle it (at the instance level) and that transitive closure computation, relation composition, intersection and union are feasible in your framework.

In the sequel, since we use FADA (see [BCF97, Bar98] and Section 2.4.3) as reaching definition analysis, we inherit its syntactical restrictions: data structures are scalars and arrays; pointers are not allowed. Loops, conditionals and array subscripts are unrestricted. Therefore, Maximal-Static-Expansion and MSE-Convert-Quast are based on the classical single-assignment algorithms for loop nests, see Section 5.1. They rely on Omega [KPRS96] and PIP [Fea88b] for symbolic computations. Additional algorithms and technical points are studied in Section 5.2.5. In Maximal-Static-Expansion, the function ρ mapping instances to their representatived is encoded as an affine relation between iteration vectors (augmented with the statement label), and labeling function ν is encoded as an affine relation between the same iteration vectors and a "compressed" vector space found by Enumerate-Representatives, see Section 5.2.5.

An interesting but technical remark is that, by construction of function ν—seen as a parameterized vector, a few components may take a finite—and hopefully small—number of values. Indeed, such components may represent the "statement part" of an instance. In such case, splitting array `A` into several (renamed) data structures[10] should improve performance and decrease memory usage (avoiding convex hulls of disjoint polyhedra). Consider for instance MSE of the second example: expanding `A` into `A1` and `A2` would require $6N-2$ array elements instead of $8N-2$ in Figure 5.11. Other techniques reducing

[10]Recall that in single-assignment form, statements assign disjoint (renamed) data structures.

```
MAXIMAL-STATIC-EXPANSION (program, κ, σ)
    program: an intermediate representation of the program
    κ: the conflict relation
    σ: the reaching definition relation, seen as a function
    returns an intermediate representation of the expanded program
κ* ← TRANSITIVE-CLOSURE (κ)
ℜ* ← TRANSITIVE-CLOSURE (σ ∘ σ^{-1})
ρ ← COMPUTE-REPRESENTATIVES (κ* ∩ ℜ*)
ν ← ENUMERATE-REPRESENTATIVES (κ*, ρ)
for each array A in program
do ν_A ←component-wise maximum of ν(u) for all write accesses u to A
    declaration A[shape] is replaced by A_EXP[shape, ν_A]
    for each statement S assigning A in program
    do left-hand side A[subscript] of S is replaced by A_EXP[subscript, ν(CURINS)]

    for each read reference ref to A in program
    do σ_/ref ← restriction of σ to accesses of the form (ı, ref)
        quast ← MAKE-QUAST (ν ∘ σ_/ref)
        map ← MSE-CONVERT-QUAST (quast, ref)
        ref ← map (CURINS)
return program
```

```
MSE-CONVERT-QUAST (quast, ref)
    quast: the quast representation of the reaching definition function
    ref: the original reference
    returns the implementation of quast as a value retrieval code for reference ref
switch
  case quast = {⊥} :
       return ref
  case quast = {ı} :
       A ← ARRAY(ı)
       S ← STMT(ı)
       x ← ITER(ı)
       subscript ←original array subscript in ref
       return A_EXP[subscript, x]
  case quast = {ı_1, ı_2, ...} :
       error "this case should never happen with static expansion!"
  case quast = if predicate then quast_1 else quast_2 :
       return if predicate MSE-CONVERT-QUAST (quast_1, ref)
                 else MSE-CONVERT-QUAST (quast_2, ref)
```

the number of useless memory locations allocated by our algorithm are not described in this paper.

5.2.5 Detailed Review of the Algorithm

A few technical points and computational issues are raised in the previous algorithm. This section is devoted to their analysis and resolution.

Finding Representatives for Equivalence Classes

Finding a "good" canonical representative in a set is not a simple matter. We choose the lexicographic minimum because it can be computed using classical techniques, and our first experiments gave good results.

Notice also that representatives must be described by a function ρ on write instances. Therefore, the good "parametric" properties of lexicographical minimum computations [Fea91, Pug92] are well suited to our purpose.

A general technique to compute the lexicographical minimum follows. Let $\equiv$ be an equivalence relation, and $\mathbf{C}$ an equivalence class for $\equiv$. The lexicographical minimum of $\mathbf{C}$ is:

$$\min_{<_{\text{LEX}}}(\mathbf{C}) \quad = v \in \mathbf{C} \ s.t. \ \nexists u \in \mathbf{C}, u <_{\text{LEX}} v.$$

Since $<_{\text{LEX}}$ is a relation, we can rewrite the definition using algebraic operations:

$$\min_{<_{\text{LEX}}}(\mathbf{C}) \quad = \quad \big(\equiv \backslash (<_{\text{LEX}} \circ \equiv) \big)(\mathbf{C}). \tag{5.11}$$

This is applied in our framework to classes of $\mathfrak{R}^*$ and κ^* with order $<_{\text{SEQ}}$.

Compute-Representatives $(equivalence)$
  equivalence: an affine equivalence relation over instances
  returns an affine function mapping instances to a canonical representative
  $repres \leftarrow equivalence \setminus (<_{\text{SEQ}} \circ equivalence)$
  return *repres*

Applying Algorithm Compute-Representatives to relation $\mathfrak{R}^*$ yields an affine function ρ, but this does not readily provide the labeling function ν. The last step consists in enumerating the image of ρ inside classes of equivalence relation κ^*.

Computing a Dense Labeling

To label each memory location, we associate each location to an integer point in the affine polyhedron of representatives, i.e. the image of function ρ whose range is restricted to a class of equivalence relation κ^*. Labeling boils down to scanning exactly once all the integer points in the set of representatives. This can be done using classical polyhedron-scanning techniques [AI91, CFR95] or simply by considering a "part" of the representative function in one-to-one mapping with this set. It is thus easy to compute a labeling function ν.

But computing a "good" labeling function is much more difficult: a "good" labeling should be as *dense* as possible, meaning that the number of memory locations accessed by the program must be as near as possible as the number of memory locations allocated from the shape of function ν.

A possible idea would be to count the number of integer points in the image of function ρ, thanks to Erhart polynomials [Cla96], and to build a labeling (non-affine in general) from this computation. But this would be extremely costly in practice and would sometimes generate very intricate subscripts; moreover, most compile-time properties on ν would be lost, due to the possible non-affine form. As a result, the "dense labeling problem" is mostly open at the moment. We have found an interesting partial result by Wilde and Rajopahye [WR93], but studying applicability of their technique to our more general case is left for future work.

Many simple transformations can be applied to ρ to compress its image. Thanks to the regularity of iteration spaces of practical loop nests, techniques such as global translation, division by an integer constant—when a constant stride is discovered—and projection gave excellent results on every example we studied. Algorithm Enumerate-Representatives implements these simple transformations to enumerate the image of a function whose range is restricted to a class of some equivalence relation.

Enumerate-Representatives (rel, fun)

rel: equivalence relation whose classes define enumeration domains
fun: the affine function whose image should be enumerated
returns a dense labeling of the image of *fun* restricted to a class of *rel*

repres $\leftarrow$ Compute-Representatives (rel)
enum $\leftarrow$ Symbolic-Vector-Subtract $(fun, repres \circ fun)$
apply appropriate translations, divisions and projections to iteration vectors in *enum*
return *enum*

What about Complexity and Practical Use?

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 consists only 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 $\mathfrak{R}$ or κ is impossible in general: Presburger arithmetic is not closed under transitive closure. However, very precise conservative approximations (if not exact results) can be computed. Kelly *et al.* [KPRS96] do not give a formal bound on the complexity of their algorithm, but their implementation in the Omega toolkit proved to be efficient if not concise. A short review of their algorithm is presented in Section 3.1.2. Notice again that the *exact* transitive closure is *not* necessary for our expansion scheme to be correct.

  Moreover, $\mathfrak{R}$ and κ happens to be transitive in most practical cases. In our implementation, the Transitive-Closure algorithm first checks whether the difference $(\mathfrak{R} \circ \mathfrak{R}) \setminus \mathfrak{R}$ is empty, before triggering the computation. In all three examples, both relations $\mathfrak{R}$ and κ are already transitive.
- In the algorithm above, ρ is a lexicographical minimum. The expansion scheme just needs a way to pick one element per equivalence class. Computing the lexicographical minimum is expensive a priori, but was easy to implement.
- Finally, numbering classes becomes costly only when we have to scan a polyhedral set of representatives in dimension greater than 1. In practice, we only had intervals on our benchmark examples.

Is our Result Maximal?

Our expansion scheme depends on the transitive closure calculator, and of course on the accuracy of input information: instancewise reaching definitions σ and approximation κ^* of the original program storage mapping. We would like to stress the fact that the expansion produced is static and maximal with respect to the results yielded by these parts, whatever their accuracy:

- The exact transitive closure may not be available (for computability or complexity reasons) and may therefore be over-approximated. The expansion factor of a memory location c is then lower than $\mathrm{Card}(\{u \in \mathbf{W} : f_e(u) = c\}/\mathfrak{R}^*)$. However, the expansion remains *static* and is *maximal* with respect to the transitive closure given to the algorithm.

- Relation κ^* approximating the storage mapping of the original program may be more or less precise, but we required it to be *pessimistic* (a.k.a. conservative). This point does not interfere with the staticity or maximality of the expansion; but the more accurate the relation κ^*, the less unused memory is allocated by the expanded program.

5.2.6 Application to Real Codes

Despite good performance results on small kernels (see following sections), it is obvious that reaching definition analysis and MSE will become unacceptably expensive on larger codes. When addressing real programs, it is therefore necessary to apply the MSE algorithm independently to several loop nests. A parallelizing compiler (or a profiler) can isolate loop nests that are critical program parts and where spending time in powerful optimization techniques is valuable. Such techniques have been investigated by Berthou in [Ber93], and also in the Polaris [BEF$^+$96] and SUIF [H$^+$96] projects.

However, some values may be initialized outside of the analyzed code. When the set of possible reaching definitions for some read accesses is *not a singleton* and includes $\perp$, it is necessary to perform some *copy-in* at the beginning of the code. Each array holding values that may be read by such accesses must be copied into the appropriate expanded arrays. In practice this is expensive when expanded arrays hold many copies of original values. However, the process is fully parallel and can hopefully not cost more than the loop nest itself.

There is a simple way to avoid copy-in, to the cost of some loss in the expansion degree. It consists in adding "virtual write accesses" for every memory location and replacing $\perp$s in the reaching definition relation by the appropriate virtual access (access*es* indeed, when the memory location accessed is unknown). Since all $\perp$s have been removed, computing the maximal static expansion from this modified reaching definition relation requires no copy-in; but additional constraints due to the "virtual accesses" may forbid some array expansions. This technique is especially useful when many temporary arrays are involved in a loop nest. But its application to the second motivating example (Figure 5.9) would forbid all expansion since almost all reads may access values defined outside the nest.

Moreover, the data structures created by MSE on each loop nest may be different, and the accesses to the same original array may now be inconsistent. Consider for instance the original pseudo code in Figure 5.13.a. We assume the first nest was processed separately by MSE, and the second nest by any technique. The code appears in Figure 5.13.b. Clearly, references to `A` may be inconsistent: a read reference in the second nest does not know which ν_1 to read from.

A simple solution is then to insert, between the two loop nests, a *copy-out* code in which the original structure is restored (see Figure 5.13). Doing this only requires to add, at the end of the first nest, "virtual accesses" that reads every memory locations written

```
for i ···
  ··· A[f1(i)] ···
end for
···
for i ···
  ··· = A[f2(i)] ···
end for
```

Figure 5.13.a. Original code

```
for i ···
  ··· A1[f1(i), ν1(i)] ···
end for
···
for i ···
  ··· = A1[f2(i), /* unknown */] ···
end for
```

Figure 5.13.b. MSE version

```
for i ···
  ··· A1[f1(i), ν1(i)] ···
end for
···
for c ···    // copy-out code
  A[c] = A1[c, ν1(σ(···))]
end for
···
for i ···
  ··· = A[f2(i)] ···
end for
```

Figure 5.13.c. MSE with copy-out

Figure 5.13. Inserting copy-out code

in the nest. The reaching definitions within the nest give the identity of the memory location to read from. Notice that no ϕ functions are necessary in the copy code—the opposite would lead to a non-static expansion. More precisely, if we call $V(c)$ the "virtual access" to memory location c after the loop nest, we can compute the maximal static expansion for the nest and the additional virtual accesses, and the value to copy back into c is located in $(c, \nu(\sigma(V(c))))$.

Fortunately, with some knowledge on the program-wide flow of data, several optimizations can remove the copy-out code[11]. The simplest optimization is to remove the copy-out code for some data structure when no read access executing after the nest uses a value produced inside this nest. The copy-out code can also be removed when no ϕ functions are needed in read accesses executing after the nest. Eventually, it is always possible to remove the copy-out code in performing a forward substitution of $(c, \nu(\sigma(V(c))))$ into read accesses to a memory location c in following nests.

5.2.7 Back to the Examples

This section applies our algorithm to the motivating examples, using the Omega Calculator [Pug92] as a tool to manipulate affine relations.

[11] Let us notice that, if MSE is used in codesign, the intermediate copy-code and associated data structures would correspond to additional logic and buffers, respectively. Both should be minimized in complexity and/or size.

First Example

Consider again the program in Figure 5.6 page 169. Using the Omega Calculator text-based interface, we describe a step-by-step execution of the expansion algorithm. We have to code instances as integer-valued vectors. An instance $\langle S_s, i\rangle$ is denoted by vector `[i,..,s]`, where `[..]` possibly pads the vector with zeroes. We number T, S, R with 1, 2, 3 in this order, so $\langle T, i\rangle$, $\langle S, i, j\rangle$ and $\langle R, i\rangle$ are written `[i,0,1]`, `[i,j,2]` and `[i,0,3]`, respectively.

From (5.1) and (5.2), we construct the relation `S` of reaching definitions:

```
S := {[i,1,2]->[i,0,1] : 1<=i<=N}
     union {[i,w,2]->[i,w-1,2] : 1<=i<=N && 2<=w}
     union {[i,0,3]->[i,0,1] : 1<=i<=N}
     union {[i,0,3]->[i,w,2] : 1<=i<=N && 1<=w};
```

Since we have only one memory location, relation κ tells us that all instances are related together, and can be omitted.

Computing $\mathfrak{R}$ is straightforward:

```
S' := inverse S;
R := S(S');
R;

{[i,0,1]->[i,0,1] : 1<=i<=N} union
{[i,w,2]->[i,0,1] : 1<=i<=N && 1<=w} union
{[i,0,1]->[i,w',2] : 1<=i<=N && 1<=w'} union
{[i,w,2]->[i,w',2] : 1<=i<=N && 1<=w' && 1<=w}
```

In mathematical terms, we get:

$$\begin{aligned}
\langle T, i\rangle \,\mathfrak{R}\, \langle T, i\rangle &\iff 1 \le i \le N \\
\langle S, i, w\rangle \,\mathfrak{R}\, \langle S, i, w'\rangle &\iff 1 \le i \le N, w \ge 1, w' \ge 1 \\
\langle S, i, w\rangle \,\mathfrak{R}\, \langle T, i\rangle &\iff 1 \le i \le N \wedge w \ge 1 \\
\langle T, i\rangle \,\mathfrak{R}\, \langle S, i, w'\rangle &\iff 1 \le i \le N \wedge w' \ge 1
\end{aligned} \tag{5.12}$$

Relation $\mathfrak{R}$ is already transitive, no closure computation is necessary:

$$\mathfrak{R} = \mathfrak{R}^*$$

There is only one equivalence class for κ^*.

Let us choose $\rho(u)$ as the first executed instance in the equivalence class of u for $\mathfrak{R}^*$ (the least instance according to the sequential order): $\rho(u) = \min_{<_{\text{SEQ}}}(\{u' : u' \,\mathfrak{R}^*\, u\})$. We may compute this expression using (5.11):

$$\forall i, w,\ 1 \le i \le N, w \ge 1 : \rho(\langle T, i\rangle) = \langle T, i\rangle, \rho(\langle S, i, w\rangle) = \langle T, i\rangle.$$

Computing $\rho(\mathbf{W})$ yields N instances of the form $\langle T, i\rangle$. Maximal static expansion of accesses to variable `x` requires N memory locations. Here, i is an obvious label:

$$\forall i, w,\ 1 \le i \le N, w \ge 1 \ :\ \nu(\langle S, i, w\rangle) = \nu(\langle T, i\rangle) = i. \tag{5.13}$$

All left-hand side references to `x` are transformed into `x[i]`; all references to `x` in the right hand side are transformed into `x[i]` too since their reaching definitions are instances of S or T for the same i. The expanded code is thus exactly the one found intuitively in Figure 5.8.

The size declaration of the new array is `x[1..N]`.

Second Example

We now consider the program in Figure 5.9. Instances $\langle S, i, j\rangle$ and $\langle T, i, N\rangle$ are denoted by `[i,j,1]` and `[i,N,2]`, respectively.

From (5.3), the relation `S` of reaching definitions 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',j',1] : 1<=i,i'<=2N && N<j'<j<=2N && i'-j'=i-j}
     union {[i,j,1]->[i',N,2] : 1<=i,i'<=2N && N<j<=2N && i'=i-j+N};
```

It is easy to compute relation κ since all array subscripts are affine: two instances of S or T, whose iteration vectors are (i, j) and (i', j') write in the same memory location iff $i - j = i' - j'$. This relation is transitive, hence $\kappa = \kappa^*$. We call it `May` in Omega's syntax:

```
May := {[i,j,s]->[i',j',s'] : 1<=i,j,i',j'<=2N && i-j=i'-j' &&
        (s=1 || (s=2 && j=N) || s'=1 || (s'=2 && j'=N))};
```

As in the first example, we compute relation $\mathfrak{R}$ 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}
```

That is:

$$
\begin{aligned}
\langle T,i,N\rangle\,\mathfrak{R}\,\langle T,i,N\rangle \quad &\Leftrightarrow \quad 1\le i\le 2N-1\\
\langle S,i,j\rangle\,\mathfrak{R}\,\langle S,i',j'\rangle \quad &\Leftrightarrow \quad (1\le i,i'\le 2N-1)\wedge(i-j=i'-j')\\
&\qquad\ \wedge\big(1\le j,j'<N\vee N<j,j'<2N-1\big)\\
\langle S,i,j\rangle\,\mathfrak{R}\,\langle T,N+i-j,N\rangle \quad &\Leftrightarrow \quad (1\le i\le 2N-1)\wedge(N<j\le 2N-1)\wedge(j<N+i)\\
\langle T,i,N\rangle\,\mathfrak{R}\,\langle S,i',N-i+i'\rangle \quad &\Leftrightarrow \quad 1\le i<i'\le 2N-1\wedge i'<N+i
\end{aligned}
$$

Relation $\mathfrak{R}$ is already transitive: $\mathfrak{R} = \mathfrak{R}^*$. Figure 5.10.a shows the equivalence classes of $\mathfrak{R}^*$.

Let $\mathbf{C}$ be an equivalence class for relation κ^*. There is an integer k s.t. $\mathbf{C} = \{\langle S,i,j\rangle : i-j=k\}\cup\{\langle T,k+N,N\rangle\}$. Now, for $u\in\mathbf{C}$, $\rho(u) = \min_{<_{\text{SEQ}}}(\{u'\in\mathbf{W} : u'\,\kappa^*\,u\wedge u'\,\mathfrak{R}^*\,u\})$.

Then, we compute $\rho(u)$ using Omega:

$$
\begin{aligned}
1-2N\le i-j\le -N \quad &:\quad \rho(\langle S,i,j\rangle)=\langle S,1,1-i+j\rangle\\
1-N\le i-j\le N-1\wedge j<N \quad &:\quad \rho(\langle S,i,j\rangle)=\langle S,i-j+1,1\rangle\\
1-N\le i-j\le N-1\wedge j>=N \quad &:\quad \rho(\langle S,i,j\rangle)=\langle T,i,N\rangle\\
N\le i-j\le 2N-1 \quad &:\quad \rho(\langle S,i,j\rangle)=\langle S,i-j+1,1\rangle\\
1\le i\le 2N-1 \quad &:\quad \rho(\langle T,i,N\rangle)=\langle T,i,N\rangle
\end{aligned}
$$

The result shows three intervals of constant cardinality of $\mathbf{C}/_{\mathfrak{R}^*}$; they are described in Figure 5.10.b. A labeling can be found mechanically. If $i-j \leq -N$ or $i-j \geq N$, there is only one representative, thus $\nu(\langle S,i,j\rangle) = 1$. If $1-N \leq i-j \leq N-1$, there are two representatives; then we define $\nu(\langle S,i,j\rangle) = 1$ if $j \leq N$, $\nu(\langle S,i,j\rangle) = 2$ if $j > N$, and $\nu(\langle T,i,N\rangle) = 2$.

The static expansion code appears in Figure 5.11. As hinted in Section 5.2.4, conditionals in ν have been taken out of array subscripts.

Array `A` is allocated as `A[4*N, 2]`. Note that some memory could have been spared in defining two different arrays: `A1` standing for `A[`$\cdots$`,0]` holding $4N-1$ elements, and `A2` standing for `A[`$\cdots$`,1]` holding only $2N-1$ elements. This idea was pointed out in Section 5.2.4.

Third Example: Non-Affine Array Subscripts

We come back to the program in Figure 5.12.a. Instances $\langle T,i,j\rangle$, $\langle S,i\rangle$ and $\langle R,i\rangle$ are written `[i,j,1]`, `[i,0,2]` and `[i,0,3]`.

From (5.4), we build the relation of reaching definitions:

```
S := {[i,0,3]->[i,j,1] : 1<=i,j<=N}
     union {[i,0,3]->[i,0,2] : 1<=i<=N};
```

Since some subscripts are non affine, we cannot compute at compile-time the exact relation between instances writing in some location `A[x]`. We can only make the following pessimistic approximation of κ: all instances are related together (because they *may* assign the same memory location).

```
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}
```

$\mathfrak{R}$ is already transitive: $\mathfrak{R} = \mathfrak{R}^*$.

There is only one equivalence class for κ^*.

We compute $\rho(u)$ using Omega:

$$\forall i,\ 1 \leq i \leq N\ :\ \rho(\langle S,i\rangle) = \langle T,i,1\rangle$$
$$\forall i,j,\ 1 \leq i \leq N, 1 \leq j \leq N\ :\ \rho(\langle T,i,j\rangle) = \langle T,i,1\rangle$$

Note that every $\langle T,i,j\rangle$ instance is in relation with $\langle T,i,1\rangle$.

Computing $\rho(\mathbf{W})$ yields N instances of the form $\langle T,i\rangle$. Maximal static expansion of accesses to variable `x` requires N memory locations. We can use i to label these representatives; thus the resulting ν function is:

$$\nu(\langle S,i\rangle) = \nu(\langle T,i,j\rangle) = i.$$

Using this labeling, all left hand side references to `A[···]` become `A[···, i]` in the expanded code. Since the source of $\langle R, i\rangle$ 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 in Figure 5.12.b.

The size declaration of `A` is now `A[N+1, N+1]`.

5.2.8 Experiments

We ran a few experiments on an SGI Origin 2000, using the `mp` library. Implementation issues are discussed in Section 5.2.9.

Performance Results for the First Example

For the first example, the parallel SA and MSE programs are given in Figure 5.14. Remember that w is an artificial counter of the `while`-loop, and M is the maximum number of iterations of this loop. We have seen that a ϕ function is necessary for SA form, but it can be computed at low cost: it represents the *last* iteration of the inner loop.

```
     double xT[N], xS[N, M];
     parallel for (i=1; i<=N; i++) {
T      xT[i] = ···;
       w = 1;
       while (···) {
S        xS[i][w] = if (w==1) xT[i] ···;
         w++;
       }
           else xS[i, w-1] ···;
R      ··· = if (w==1) xT[i] ···;
         else xS[i, w-1] ···;
       // the last two lines implement
       // φ({⟨T, i⟩} ∪ {⟨S, i, w⟩ : 1 ≤ w ≤ M})
     }
```

Figure 5.14.a. Single-assignment

```
     double x[N+1];
     parallel for (i=1; i<=N; i++)
T      x[i] = ···;
       while (···)
S        x[i] = x[i] ···;
R      ··· = x[i] ···;
     }
```

Figure 5.14.b. Maximal static expansion

Figure 5.14. Parallelization of the first example.

Table in Figure 5.15 first describes speed-ups for the maximal static expansion relative to the *original sequential* program, then speed-ups for the MSE version relative to the *single-assignment* form. As expected, MSE shows a better scaling, and the relative speed-up quickly goes over 2. Moreover, for larger memory sizes, the SA program may swap or fail for lack of memory.

5.2.9 Implementation

The maximal static expansion is implemented in C++ on top of the Omega library. Figure 5.16 summarizes the computation times for our examples (on a 32MB Sun SPARCstation 5). These results do not include the computation times for reaching definition analysis and code generation.

	$M \times N$				
Configuration	200×250	200×500	200×1000	200×2000	200×4000
Speed-ups for MSE versus original program					
16 processors	6.72	9.79	12.8	13.4	14.7
32 processors	5.75	9.87	15.3	21.1	24.8
Speed-ups for MSE versus SA					
16 processors	1.43	1.63	1.79	1.96	2.07
32 processors	1.16	1.33	1.52	1.80	1.99

Figure 5.15. Experimental results for the first example

	1st example	2nd example	3rd example
transitive closure (check)	100	100	110
picking the representatives (function ρ)	110	160	110
other	130	150	70
total	340	410	290

Figure 5.16. Computation times, in milliseconds.

Moreover, computing the class representatives is relatively fast; it validates our choice to compute function ρ (mapping instances to their representatives) using a lexicographical minimum. The intuition behind these results is that the computation time mainly depends on the number of affine constraints in the data-flow analysis relation.

Our only concern, so far, would be to find a way to approximate the expressions of transitive closures when they become large.

5.3 Storage Mapping Optimization

Memory expansion techniques have two main drawbacks: high memory usage and run-time overhead. Parallelization via memory expansion thus requires both *moderation* in the expansion degree and *efficiency* in the run-time computation of data-flow restoration code.

Moderation in the expansion degree can be addressed in two ways: either with "hard constraints" such as the one presented in Section 5.2 or with optimization techniques that do not interfere with parallelism extraction. This section addresses such optimization techniques, and presents the main results of a collaboration with Vincent Lefebvre. It can be seen as an extension of a work by Feautrier and Lefebvre [LF98] and also by Strout et al. [SCFS98].

Our contributions are the following: we formalize the correctness of a storage mapping, *according to a given parallel execution order*, for any nest of loops with *unrestricted conditional expressions and array subscripts*; we show that schedule-independent storage mappings defined in [SCFS98] correspond to correct storage mappings *according to the data-flow execution order*; and we present an algorithm for storage mapping optimization, applicable to any nest of loops and to *all parallelization techniques* based on polyhedral dependence graphs (i.e. captured by Presburger arithmetics).

5.3.1 Motivation

First Example: Dynamic Control Flow

We first study the kernel in Figure 5.17.a, which was already the first motivating example in Section 5.2. Parts denoted by $\cdots$ have no side-effect. Each loop iteration spawns *instances* of statements included in the loop body.

```
    double x;
    for (i=1; i<=N; i++) {
T     x = ···;
      while (···) {
S       x = x ···;
      }
R     ··· = x ···;
    }
```

Figure 5.17.a. Original program

```
    double x_T[N+1], x_S[N+1, M+1]
    parallel for (i=1; i<=N; i++) {
T     x_T[i] = ···;
      w = 1;
      while (···) {
S       x_S[i][w] = if(w=1) x_T[i] ···;
          else x_S[i, w-1] ···;
        w++;
      }
R     ··· = if (w==1) x_T[i] ···;
        else x_S[i, w-1] ···;
      // the last two lines implement
      // φ({⟨T,i⟩} ∪ {⟨S,i,w⟩ : 1 ≤ w ≤ M})
    }
```

Figure 5.17.b. Single-assignment

```
    double x_TS[N+1]
    parallel for (i=1; i<=N; i++) {
T     x_TS[i] = ···;
      while (···) {
S       x_TS[i] = x_TS[i] ···;
      }
R     ··· = x_TS[i] ···;
    }
```

Figure 5.17.c. Partial expansion

Figure 5.17. Convolution example

Any instancewise reaching definition analysis is suitable to our purpose, but FADA

[BCF97] is prefered since it handles any loop nest and achieves today's best precision. Value-based dependence analysis [Won95] is also a good choice. In the following, The results for references `x` in right-hand side of R and S are *nested conditionals*:

$$\begin{aligned}\sigma\left(\langle S, i, w, \mathtt{x}\rangle\right) &= \textbf{\textit{if}}\ w = 1\ \textbf{\textit{then}}\ \{T\}\ \textbf{\textit{else}}\ \{\langle S, i, w-1\rangle\} \\ \sigma\left(\langle R, i, \mathtt{x}\rangle\right) &= \{\langle S, i, w\rangle : 1 \leq w\}.\end{aligned}$$

Here, memory-based dependences hampers direct parallelization via *scheduling* or *tiling*. We need to expand scalar `x` and remove as many output, flow and anti-dependences as possible. Reaching definition analysis is at the core of single-assignment (SA) algorithms, since it records the location of values in expanded data structures. However, when the flow of data is unknown at compile-time, ϕ functions are introduced for run-time restoration of values [CFR$^+$91, Col98]. Figure 5.17.b shows our program converted to SA form, with the outer loop marked parallel (M is the maximum number of iterations of the inner loop). A ϕ function is necessary but can be computed at low cost since it represents the *last* iteration of the inner loop.

SA programs suffer from high memory requirements: S now assigns a huge $N \times M$ array. Optimizing memory usage is thus a critical point when applying memory expansion techniques to parallelization.

Figure 5.17.c shows the parallel program after partial expansion. Since T executes before the inner loop in the parallel version, S and T may assign the same array. Moreover a one-dimensional array is sufficient since the inner loop is not parallel. As a side-effect, no ϕ function is needed any more. Storage requirement is N, to be compared with $NM+N$ in the SA version, and with 1 in the original program (allowing no legal parallel reordering).

This partial expansion has been designed for a particular parallel execution order. However, it is easy to show that it is also compatible with all other execution orders, since the inner loop cannot be parallelized. We have thus built a ***schedule-independent*** (a.k.a. universal) storage mapping, in the sense of [SCFS98]. On many programs, a more memory-economical technique consists in computing a legal storage mapping *according to a given parallel execution order*, instead of finding a schedule-independent storage compatible with any legal execution order. This is done in [LF98] for *affine* loop nests only.

Second Example: a More Complex Parallelization

We now consider the program in Figure 5.18 which solves the well known knapsack problem (KP). This kernel naturally models several optimization problems [MT90]. Intuitively: M is the number of objects, C is the "knapsack" capacity, `W[k]` (resp. `P[k]`) is the weight (resp. profit) of object number k; the problem is to maximize the profit without exceeding the capacity. Instances of S are denoted by $\langle S, k, W[k]\rangle$, ... ,$\langle S, k, C\rangle$, for $1 \leq k \leq M$.

```
  int A[C+1], W[M+1], P[M+1];
  for (k=1; k<=M; k++)
    for (j=W[k]; j<=C; j++)
S     A[j] = max (A[j], P[k] + A[j-W[k]]);
```

Figure 5.18. Knapsack program

We suppose (from additional static analyses) that `W[k]` is always *positive* and less than or equal to an integer K. The result for references `A[j]` and `A[j-W[k]]` in right-hand side of S are *conditionals*:

$$\begin{array}{rcl} \sigma\left(\langle S, k, j, \texttt{A[j]}\rangle\right) & = & \left| \begin{array}{l} \textbf{\textit{if}}\ k = 1 \\ \textbf{\textit{then}}\ \{\bot\} \\ \textbf{\textit{else}}\ \{\langle S, k-1, j\rangle\} \end{array} \right. \\ \sigma\left(\langle S, k, j, \texttt{A[j-W[k]]}\rangle\right) & = & \{\langle S, k', j'\rangle : 1 \le k' \le k \wedge \max(0, j-K) < j' < j-1\} \end{array}$$

First notice that program KP does not have any parallel loops, and that memory-based dependences hampers direct parallelization. Therefore, parallelizing KP requires the application of preliminary program transformations.

Thanks to the reaching definition information, Figure 5.19 shows program KP converted to SA form. The unique ϕ function implements a run-time choice between values produced by $\{\langle S, k', j'\rangle : 1 \le k' \le k \wedge \max(0, j-K) < j' < j-1\}$, for some read access $\langle S, k, j, \texttt{A[j-W[k]]}\rangle$.

```
     int A[C+1], W[M+1], P[M+1]
     int A_S[M+1, C+1]
     for (k=1; k<=M; k++)
       for (j=W[k]; j<=C; j++)
S        A_S[k, j] = if (k==1)
             max (A[j], P[1] + A[j-W[1]]);
           else
             max (A_S[k-1, j],
                  P[k] + φ({⟨S, k', j'⟩ : 1 ≤ k' ≤ k ∧ max(0, j − K) < j' < j − 1});
```

Figure 5.19. KP in single-assignment form

Eventually, in this particular case, the ϕ function is really easy to compute: the value of `A[j-W[k]]` has been "moved" by SA form transformation "to" $\texttt{A}_S$`[k, j-W[k]]`. Then $\phi(\{\langle S, k', j'\rangle : 1 \le k' \le \texttt{k} \wedge \max(0, \texttt{j} - K) < j' < \texttt{j} - 1\})$ is equal to $\texttt{A}_S$`[k,j-W[k]]`. This optimization avoids the use of temporary arrays. It can be performed automatically, along with other interesting optimizations, see Section 5.1.4.

The good thing with SA-transformed programs is that the only remaining dependences are *true dependences* between a *reaching definition* instance and its *use* instances. Thus a legal parallel *schedule* for program KP is: *"execute instance $\langle S, k, j\rangle$ at step $k + j$"*, see Figure 5.20 (see Section 2.5.2 for schedule computation).

Since KP is a perfectly nested loop, it is also possible to apply *tiling* techniques to single-assignment KP, based on instancewise reaching definition information. Tiling techniques improve data locality and reduce communications in grouping together computations affecting the same part of a data structure (see Section 2.5.2). Rectangular $m \times c$ tiles seem appropriate in our case; the height m and width c can be tuned thanks to theoretical models [IT88, CFH95, BDRR94] or profiling techniques. The knapsack problem has been much studied and very efficient parallelizations have been crafted by Andonov and Rajopadhye [AR94], see also [BBA98] for additional information on tiling

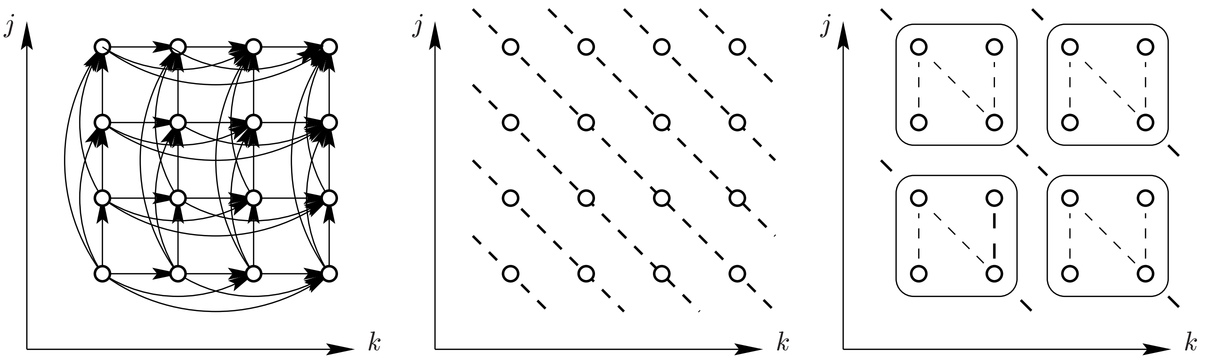


Figure 5.20. Instancewise reaching definitions, schedule, and tiling for KP

the knapsack algorithm. The third graph in Figure 5.20 represents 2×2 tiles, but larger sizes are used in practice, see Section 5.3.10.

Consider the dependences in Figure 5.20. The value produced by instance $\langle S, k, j\rangle$ may be used by $\langle S, k, j+1\rangle, \ldots, \langle S, k, \min(C, j+K)\rangle$ or by $\langle S, k+1, j\rangle$. Using the schedule or the tiling proposed in Figure 5.20, we can prove that some value produced during the execution stops being useful after a given delay: if $1 \leq k, k' \leq M$ and $1 \leq j, j' \leq C$ are such that $k + j + K < k' + j'$, the value produced by $\langle S, k, j\rangle$ is not used by $\langle S, k', j'\rangle$. This allows a cyclic folding of the storage mapping: every access of the form $\texttt{A}_S\texttt{[k, j]}$ can be safely replaced by $\texttt{A}_S\texttt{[k \% (K+1), j]}$. The result is shown in Figure 5.21.

```
    int A[C+1], W[M+1], P[M+1]
    int A_S[K+2, C+1]
    for (k=1; k<=M; k++)
      for (j=W[k]; j<=C; j++)
S       A_S[k % (K+1), j] = if (k==1)
            max (A[j], P[1] + A[j-W[1]]);
          else
            max (A_S[(k-1) % (K+1), j],
                  P[k] + φ({⟨S,k',j'⟩ : 1 ≤ k' ≤ k ∧ max(0,j − K) < j' < j − 1});
```

Figure 5.21. Partial expansion for KP

Storage requirement for array $\texttt{A}_S$ is $(K+1)C$, to be compared with MC in the SA version, and with C in the original program (where no legal parallel reordering was possible). This suggests two observations:

- first, the gain is only significant when K is much smaller than M, which may not be the case in practice;
- second, the expanded subscript in left-hand side is not affine any more, since K is a symbolic constant.

In general, when the cyclic folding is based on a symbolic constant (like K), it becomes both difficult to measure the effectiveness of the optimization and to reuse the generated code in subsequent analyses. In [Lef98], Lefebvre proposed to forbid such symbolic foldings, but we believe they can still be useful when some compile-time information on the symbolic bounds (like K) is available.

Eventually, this partial expansion is not schedule-independent, because it highly depends on the "parallel front" direction associated with the proposed schedule and tiling.

5.3.2 Problem Statement and Formal Solution

Given an original program $(<_{\text{SEQ}}, f_e)$, we suppose that an instancewise reaching definition analysis has already been performed—yielding relation σ—and that a parallel execution order $<_{\text{PAR}}$ has been computed using some suitable technique (see Chapter 2.5.2). Our problem is here to compute a new storage mapping f_e^{EXP} such that $(<_{\text{PAR}}, f_e^{\text{EXP}})$ preserves the original semantics of $(<_{\text{SEQ}}, f_e)$.

Given a parallel execution order $<_{\text{PAR}}$, we have to characterize *correct expansions* allowing parallel execution to preserve the program semantics. In addition to the conflict relation κ_e, we use the ***no-conflict relation*** $\not\kappa_e$, which is the complement of κ_e. As in Section 2.4.1, we build a conservative approximation $\not\kappa$ of this relation:

$$\forall e \in \mathbf{E}, \forall v, w \in \mathbf{A}_e : \quad \big(f_e(v) \neq f_e(w) \implies v \not\kappa\, w\big).$$

Since both approximations κ and $\not\kappa$ are *conservative*, we have to be very careful that they are *not complementary* in general. Indeed, κ_e and $\not\kappa_e$ are complementary for the *same* execution $e \in \mathbf{E}$, but κ is defined as a "may conflict" approximation for all executions, and $\not\kappa$ is the negation of the "must conflict" approximation.

Our first task is to formalize the memory reuse constraints enforced by the partial order $<_{\text{PAR}}$. We introduce σ'_e: the exact reaching definition function for a given execution e of parallelized program $(<_{\text{PAR}}, f_e^{\text{EXP}})$.[12] The expansion is correct iff, for every program execution, the source of every access is the same in the sequential and in the parallel program:

$$\forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e, \forall v \in \mathbf{W}_e : \quad v = \sigma_e(u) \implies v = \sigma'_e(u). \tag{5.14}$$

We are looking for a *correctness criterion* telling whether two writes may use the same memory location or not. To do this, we return to the definition of σ'_e:

$$\begin{aligned}&\forall e \in \mathbf{E} : \quad v = \sigma'_e(u) \iff \\ &v <_{\text{PAR}} u \wedge f_e^{\text{EXP}}(u) = f_e^{\text{EXP}}(v) \wedge \big(\forall w \in \mathbf{W}_e : u <_{\text{PAR}} w \vee w <_{\text{PAR}} v \vee f_e^{\text{EXP}}(v) \neq f_e^{\text{EXP}}(w)\big).\end{aligned} \tag{5.15}$$

Plugging (5.15) in (5.14), we get

$$\begin{aligned}\forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e, \forall v, w \in \mathbf{W}_e : \quad & v = \sigma_e(u) \wedge u \not<_{\text{PAR}} w \wedge w \not<_{\text{PAR}} v \implies \\ & v <_{\text{PAR}} u \wedge f_e^{\text{EXP}}(u) = f_e^{\text{EXP}}(v) \wedge f_e^{\text{EXP}}(v) \neq f_e^{\text{EXP}}(w).\end{aligned}$$

We may simplify this result since $v <_{\text{PAR}} u$ and $f_e^{\text{EXP}}(u) = f_e^{\text{EXP}}(v)$ constraints are already implied by $v = \sigma_e(u)$—through (5.14)—and do not bring any information between $f_e^{\text{EXP}}(v)$ and $f_e^{\text{EXP}}(w)$:

$$\begin{aligned}&\forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e, \forall v, w \in \mathbf{W}_e : \\ &\qquad v = \sigma_e(u) \wedge u \not<_{\text{PAR}} w \wedge w \not<_{\text{PAR}} v \implies f_e^{\text{EXP}}(v) \neq f_e^{\text{EXP}}(w).\end{aligned} \tag{5.16}$$

[12] The fact that $<_{\text{PAR}}$ is not a total order makes no difference for reaching definitions.

It means that we cannot reuse memory (i.e. we must expand) when both $v = \sigma_e(u)$ and $v \not\prec_{\text{PAR}} w \wedge u \not\prec_{\text{PAR}} w$ are true. Starting from this *dynamic* correctness condition, we would like to deduce a correctness criterion based on *static* knowledge only. This criterion must be valid for all executions; in other terms, it should be *stronger* than condition (5.16).

We can now expose the *expansion correctness criterion*. It requires the reaching definition v of a read u and an other write w to assign different memory locations when: w executes between v and u in the *parallel* program, and either w does *not* execute between v and u or w assigns a different memory location from v ($v \not\bowtie w$) in the *original* program; see Figure 5.22. Here is the precise formulation of the correctness criterion:

Theorem 5.2 (correctness of storage mappings) If the following condition holds, then the expansion is correct—i.e. allows parallel execution to preserve the program semantics.

$$\begin{aligned}&\forall e \in \mathbf{E}, \forall v, w \in \mathbf{W} :\\ &\qquad \exists u \in \mathbf{R} : v \,\sigma\, u \wedge w \not\prec_{\text{PAR}} v \wedge u \not\prec_{\text{PAR}} w \wedge (u <_{\text{SEQ}} w \vee w <_{\text{SEQ}} v \vee v \not\bowtie w)\\ &\qquad\qquad \Longrightarrow f_e^{\text{EXP}}(v) \neq f_e^{\text{EXP}}(w).\end{aligned} \tag{5.17}$$

Proof: We first rewrite the definition of v being the reaching definition of u:

$$\begin{aligned}&\forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e, \forall v \in \mathbf{W}_e :\\ &\qquad v = \sigma_e(u) \Longrightarrow v <_{\text{SEQ}} u \wedge f_e(u) = f_e(v) \wedge\\ &\qquad\qquad \big(\forall w \in \mathbf{W}_e : u <_{\text{SEQ}} w \vee w <_{\text{SEQ}} v \vee f_e(v) \neq f_e(w)\big).\end{aligned}$$

As a consequence,

$$\begin{aligned}&\forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e, \forall v \in \mathbf{W}_e :\\ &\qquad v = \sigma_e(u) \Longrightarrow \big(\forall w \in \mathbf{W}_e : u <_{\text{SEQ}} w \vee w <_{\text{SEQ}} v \vee f_e(v) \neq f_e(w)\big).\end{aligned} \tag{5.18}$$

The right-hand side of (5.18) can be inserted into (5.16) as an additional constraint: (5.16) is equivalent to

$$\begin{aligned}&\forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e, \forall v, w \in \mathbf{W}_e :\\ &\qquad v = \sigma_e(u) \wedge w \not\prec_{\text{PAR}} v \wedge u \not\prec_{\text{PAR}} w \wedge \big(u <_{\text{SEQ}} w \vee w <_{\text{SEQ}} v \vee f_e(v) \neq f_e(w)\big)\\ &\qquad\qquad \Longrightarrow f_e^{\text{EXP}}(v) \neq f_e^{\text{EXP}}(w).\end{aligned} \tag{5.19}$$

Let us now replace σ_e with its approximation σ in (5.19)—using $v = \sigma_e(u) \Rightarrow v \,\sigma\, u$:

$$\begin{gathered}\forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e, \forall v, w \in \mathbf{W}_e :\\ v \,\sigma\, u \wedge \big(u <_{\text{SEQ}} w \vee w <_{\text{SEQ}} v \vee f_e(v) \neq f_e(w)\big) \wedge w \not\prec_{\text{PAR}} v \wedge u \not\prec_{\text{PAR}} w\\ \Longrightarrow f_e^{\text{EXP}}(v) \neq f_e^{\text{EXP}}(w)\\ \Downarrow \text{ approximation: } v = \sigma_e(u) \Rightarrow v \,\sigma\, u\\ \forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e, \forall v, w \in \mathbf{W}_e :\\ v = \sigma_e(u) \wedge \big(u <_{\text{SEQ}} w \vee w <_{\text{SEQ}} v \vee f_e(v) \neq f_e(w)\big) \wedge w \not\prec_{\text{PAR}} v \wedge u \not\prec_{\text{PAR}} w\\ \Longrightarrow f_e^{\text{EXP}}(v) \neq f_e^{\text{EXP}}(w)\end{gathered}$$

Eventually, we approximate f_e over all executions thanks to relation $\not\kappa$—using $f_e(v) \neq f_e(u) \Rightarrow v \not\kappa\, u$:

$$\begin{gathered}
\forall v, w \in \mathbf{W} : \\
\exists u \in \mathbf{R} : v\, \sigma\, u \wedge w \not\prec_{\text{PAR}} v \wedge u \not\prec_{\text{PAR}} w \wedge (u <_{\text{SEQ}} w \vee w <_{\text{SEQ}} v \vee v \not\kappa\, w) \\
\Longrightarrow f_e^{\text{EXP}}(v) \neq f_e^{\text{EXP}}(w) \\
\Downarrow \quad \text{approximation: } f_e(v) \neq f_e(u) \Rightarrow v \not\kappa\, u \\
\forall e \in \mathbf{E}, \forall u \in \mathbf{R}_e, \forall v, w \in \mathbf{W}_e : \\
v\, \sigma\, u \wedge \big(u <_{\text{SEQ}} w \vee w <_{\text{SEQ}} v \vee f_e(v) \neq f_e(w)\big) \wedge w \not\prec_{\text{PAR}} v \wedge u \not\prec_{\text{PAR}} w \\
\Longrightarrow f_e^{\text{EXP}}(v) \neq f_e^{\text{EXP}}(w)
\end{gathered}$$

This proves that (5.17) is stronger than (5.19), itself equivalent to (5.16). ■

Notice we returned to the definition of σ_e at the beginning of the proof. Indeed, *some information* on the storage mapping may be available, and we do not want to loose it[13]: the right-hand side of (5.18) gathers information on w which would have been lost in approximating σ_e by σ in (5.16). Without this information on w, we would have computed the following correctness criterion:

$$\begin{aligned}
&\forall e \in \mathbf{E}, \forall v, w \in \mathbf{W} : \\
&\qquad \big(\exists u \in \mathbf{R} : v = \sigma(u) \wedge u \not\prec_{\text{PAR}} w \wedge w \not\prec_{\text{PAR}} v\big) \Longrightarrow f_e^{\text{EXP}}(v) \neq f_e^{\text{EXP}}(w).
\end{aligned} \tag{5.20}$$

Sadly, this choice is not satisfying here.[14] Indeed, consider the motivating example: two instances $\langle S, i, w\rangle$ and $\langle S, i, w'\rangle$ would satisfy the left-hand side of (5.20) as long as $w \neq w'$. Therefore, they should assign different memory locations in any correct expanded program. This leads to the single-assignment version of the program... but we showed in Section 5.3.1 that a more memory-economical solution was available: see Figure 5.17.c.

A precise look to (5.16) explains why replacing σ_e with σ in 5.16) is too conservative: it "forgets" that w is not executed after *the* reaching definition $\sigma_e(u)$. Indeed, $w \not\prec_{\text{PAR}} v$ in left-hand side of (5.20) is much stronger: it states that w is not executed after any possible reaching definitions of u, which includes many instances execution before *the* reaching definition $\sigma_e(u)$.

In the following, we introduce a new notation for the expansion correctness criterion: the *interference relation* $\bowtie$ is defined as the symmetric closure of the left-hand side of (5.17):

$$\begin{aligned}
&\forall v, w \in \mathbf{W} : \quad v \bowtie w \overset{\text{def}}{\Longleftrightarrow} \\
&\qquad \big(\exists u \in \mathbf{R} : v\, \sigma\, u \wedge w \not\prec_{\text{PAR}} v \wedge u \not\prec_{\text{PAR}} w \wedge (u <_{\text{SEQ}} w \vee w <_{\text{SEQ}} v \vee v \not\kappa\, w)\big) \\
&\vee \quad \big(\exists u \in \mathbf{R} : w\, \sigma\, u \wedge v \not\prec_{\text{PAR}} w \wedge u \not\prec_{\text{PAR}} v \wedge (u <_{\text{SEQ}} v \vee v <_{\text{SEQ}} w \vee w \not\kappa\, v)\big).
\end{aligned} \tag{5.21}$$

We take the symmetric closure because v and w play symmetric roles in (5.17). Using a tool like Omega [Pug92], it is much easier to handle set and relation operations than

[13] Such information may be more precise than deriving it from the approximate reaching definition relation σ.

[14] This criterion was enough for Lefebvre and Feautrier in [LF98] since they only considered affine loop nests and exact reaching definition relations.

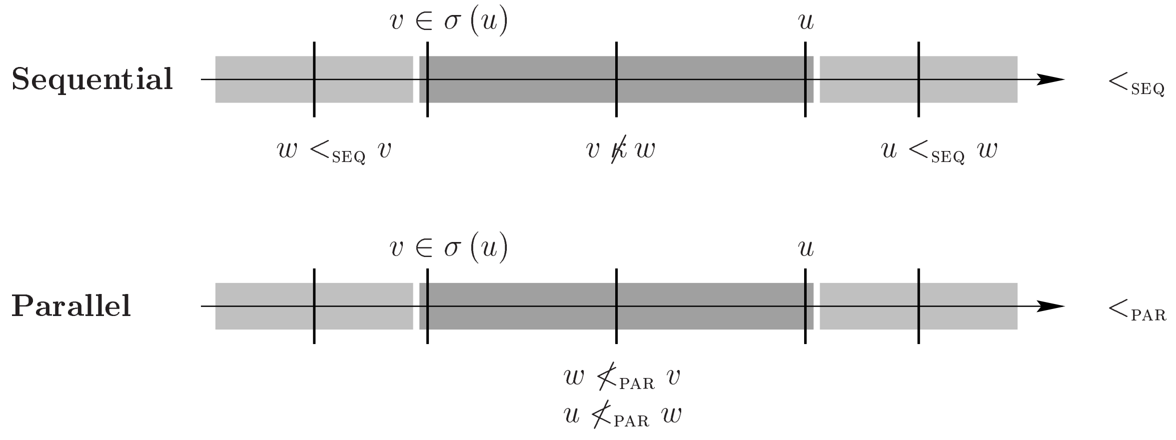


Figure 5.22. Cases of $f_e^{\text{EXP}}(v) \neq f_e^{\text{EXP}}(w)$ in (5.17)

logic formulas with quantifiers. We thus rewrite the previous definition using algebraic operations:[15]

$$\begin{aligned}\bowtie \quad = \quad & \big((\sigma(\mathbf{R}) \times \mathbf{W}) \cap \not>_{\text{PAR}} \cap (>_{\text{SEQ}} \cup \not\kappa)\big) \cup \big(\not>_{\text{PAR}} \cap (\sigma \circ (\not<_{\text{PAR}} \cap <_{\text{SEQ}}))\big) \\ \cup \quad & \big((\sigma(\mathbf{R}) \times \mathbf{W}) \cap \not<_{\text{PAR}} \cap (<_{\text{SEQ}} \cup \not\kappa)\big) \cup \big(\not<_{\text{PAR}} \cap (\sigma \circ (\not<_{\text{PAR}} \cap <_{\text{SEQ}}))\big). \qquad (5.22)\end{aligned}$$

Rewriting (5.17) with this new syntax, v and w must assign distinct memory locations when $v \bowtie w$—one may say that "*v interferes with w*":

$$\forall e \in \mathbf{E}, \forall v, w \in \mathbf{W}: \quad v \bowtie w \implies f_e^{\text{EXP}}(v) \neq f_e^{\text{EXP}}(w). \qquad (5.23)$$

An algorithm to compute f_e^{EXP} from Theorem 5.2 is presented in Section 5.3.4. Notice that we compute an *exact* storage mapping f_e^{EXP} which *depends on the execution*.

5.3.3 Optimality of the Expansion Correctness Criterion

We start with three examples showing the usefulness of each constraint in the definition of $\bowtie$, see Figure 5.23.

We now present the following optimality result:[16]

Proposition 5.2 Let $<_{\text{PAR}}$ be a parallel execution order. Consider two writes v and w such that $v \bowtie w$ (defined in (5.22) page 194), and a storage mapping f_e^{EXP} such that $f_e^{\text{EXP}}(v) = f_e^{\text{EXP}}(w)$—that is, f_e^{EXP} does not satisfy the expansion correctness criterion defined by Theorem 5.2. Then, executing program $(<_{\text{PAR}}, f_e^{\text{EXP}})$ violates the original program semantics, according to approximations σ and $\not\kappa$.

Proof: Suppose $v\,\sigma\,u \wedge w \not<_{\text{PAR}} v \wedge u \not<_{\text{PAR}} w \wedge (u <_{\text{SEQ}} w \vee w <_{\text{SEQ}} v \vee v \not\kappa\, w)$ is satisfied for a read u, and two writes v and w. One may distinguish three cases regarding execution of w relatively to u and v, see Figure 5.22.

[15]Each line of (5.21) is rewritten independently, then predicates depending on u are separated from the others. The existential quantification on u is captured by composition with σ. Because v is the possible reaching definition of some read access, intersection with $(\sigma(\mathbf{R}) \times \mathbf{W})$ is necessary in the first disjunct of each line.

[16]See Section 2.4.4 for a general remark about optimality.

T `x = ···;`
S `x = ···;`
R `··· = x ···;`

$S \parallel T <_{\text{SEQ}} R$ is legal but requires renaming: this is enforced by $T <_{\text{SEQ}} S$, i.e. $w <_{\text{SEQ}} v$ (and $T \not<_{\text{PAR}} S$, i.e. $w \not<_{\text{PAR}} v$, and $R \not<_{\text{PAR}} T$, i.e. $u \not<_{\text{PAR}} w$).

Figure 5.23.a. Constraints $w <_{\text{SEQ}} v$ and $w \not<_{\text{PAR}} v$, $u \not<_{\text{PAR}} w$

S `x = ···;`
R `··· = x ···;`
T `x = ···;`

$S <_{\text{SEQ}} T <_{\text{SEQ}} R$ is legal but requires renaming: this is enforced by $R <_{\text{SEQ}} T$, i.e. $u <_{\text{SEQ}} w$.

Figure 5.23.b. Constraints $w \not<_{\text{PAR}} v$, $u \not<_{\text{PAR}} w$ and $u <_{\text{SEQ}} w$

S `A[1] = ···;`
T `A[`foo`] = ···;`
R `··· = A[1] ···;`

$S \parallel T <_{\text{SEQ}} R$ is legal but requires renaming: this is enforced by $S \not\bowtie T$, i.e. $v \not\bowtie w$, since S *may* assign a *different* memory location as T.

Figure 5.23.c. Constraints $w \not<_{\text{PAR}} v$, $u \not<_{\text{PAR}} w$ and $v \not\bowtie w$

Figure 5.23. Motivating examples for each constraint in the definition of the interference relation

The first two cases are (1) u executes before w in the sequential program, i.e. $u <_{\text{SEQ}} w$, or (2) w executes before v in the sequential program, i.e. $w <_{\text{SEQ}} v$: then w must assign a different memory location than v, otherwise the value produced by v would never reach u as in the sequential program.

When w executes neither before v nor after u in the sequential program, one may keep v and w assigning the same memory location if it was the case in the sequential program. However, if it might not be the case, i.e. if $v \not\bowtie w$, then w must assign a different memory location than v, otherwise the value produced by v would never reach u as in the sequential program. ∎

5.3.4 Algorithm

The formalism presented in the previous section is general enough to handle any imperative program. However, as a compromise between expressivity and computability, and because our prefered reaching definition analysis is FADA [BCF97], we choose affine relations as an abstraction. Tools like Omega [Pug92] and PIP [Fea91] can thus be used for symbolic computations, but our program model is now restricted to loop nests operating on arrays, with unrestricted conditionals, loop bounds and array subscripts.

Finding the minimal amount of memory to store the values produced by the program is a graph coloring problem where vertices are instances of writes and edges represent interferences between instances: there is an edge between v and w iff they can't share the same memory location, i.e. when $v \bowtie w$. Since classic coloring algorithms only apply to finite graphs, Feautrier and Lefebvre designed a new algorithm [LF98], which we extend to general loop-nests.

The more general application of our technique starts with instancewise reaching definition analysis, then apply a parallelization algorithm using σ as dependence graph —thus avoiding constraints due to spurious memory-based dependences, describe the result as a partial order $<_{\text{PAR}}$, and eventually apply the following *partial expansion* algorithm.

Partial Expansion Algorithm

Storage-Mapping-Optimization and SMO-Convert-Quast are simple extensions of the classical single-assignement algorithms for loop nests, see Section 5.1. Input is the sequential program, the results σ and $\not\kappa$ of an instancewise analysis, and parallel execution order $<_{\text{PAR}}$ (not used for simple SA form conversion). The big difference with SA-form is the computation of an *expansion vector* $\mathtt{E}_S$ of integers or symbolic constants: its purpose is to reduce memory usage of each expanded array $\mathtt{A}_S$ with a "cyclic folding" of memory references, see Build-Expansion-Vector in Section 5.3.5. To reduce the number of expanded arrays, *partial renaming* is called at the end of the process to coalesce data structures using a classical graph coloring heuristic, see Partial-Renaming in Section 5.3.5.

Storage-Mapping-Optimization $(program, \sigma, \not\kappa, <_{\text{PAR}})$

program: an intermediate representation of the program

σ: the reaching definition relation, seen as a function

$\not\kappa$: the no-conflict relation

$<_{\text{PAR}}$: the parallel execution order

returns an intermediate representation of the expanded program

$\bowtie \leftarrow \big((\sigma(\mathbf{R}) \times \mathbf{W}) \cap \not\succ_{\text{PAR}} \cap (>_{\text{SEQ}} \cup \not\kappa)\big) \cup \big(\not\succ_{\text{PAR}} \cap (\sigma \circ (\not\prec_{\text{PAR}} \cap <_{\text{SEQ}}))\big)$

$\quad \cup \quad \big((\sigma(\mathbf{R}) \times \mathbf{W}) \cap \not\prec_{\text{PAR}} \cap (<_{\text{SEQ}} \cup \not\kappa)\big) \cup \big(\not\prec_{\text{PAR}} \cap (\sigma \circ (\not\prec_{\text{PAR}} \cap <_{\text{SEQ}}))\big)$

for each array `A` **in** *program*

do for each statement S assigning `A` **in** *program*

  do $\mathtt{E}_S \leftarrow$ Build-Expansion-Vector $(S, \bowtie)$

    declare an array $\mathtt{A}_S$

    left-hand side of $S \leftarrow \mathtt{A}_S$`[Iter(CurIns) % `$\mathtt{E}_S$`]`

  for each reference ref to `A` **in** *program*

  do $\sigma_{/ref} \leftarrow \sigma \cap (\mathbf{I} \times ref)$

    $quast \leftarrow$ Make-Quast $(\sigma_{/ref})$

    $map \leftarrow$ SMO-Convert-Quast $(quast, ref)$

    $ref \leftarrow map$ `(CurIns)`

$program \leftarrow$ Partial-Renaming $(program, \bowtie)$

return *program*

This algorithm outputs an expanded program whose data layout is well suited for parallel execution order $<_{\text{PAR}}$: we are assured that the original program semantic will be preserved in the parallel version.

Two technical issues have been pointed out. How is the expansion vector $\mathtt{E}_S$ built for each statement S? How is partial renaming performed? This is the purpose of Section 5.3.5.

5.3.5 Array Reshaping and Renaming

Building an Expansion Vector

For each statement S, the expansion vector must ensure that two instances v and w assign different memory locations when $v \bowtie w$. Moreover, it should introduce memory

SMO-Convert-Quast $(quast, ref)$
quast: the quast representation of the reaching definition function
ref: the original reference, used when $\perp$ is encoutered
returns the implementation of *quast* as a value retrieval code for reference *ref*

```
switch
  case quast = {⊥} :
       return ref
  case quast = {ı} :
       A ← Array(ı)
       S ← Stmt(ı)
       x ← Iter(ı)
       return A_S[x % E_S]
  case quast = {ı1, ı2, ...} :
       return φ({ı1, ı2, ...})
  case quast = if predicate then quast1 else quast2 :
       return if predicate SMO-Convert-Quast (quast1, ref)
                 else SMO-Convert-Quast (quast2, ref)
```

reuse between instances of S as often as possible.

Building an expanded program with memory reuse on S introduces output dependences between some instances of this statement (there is an output dependence between two instances v and w in the expanded code if $v \in \mathbf{W}$, $w \in \mathbf{W}$ and $f_e^{\text{EXP}}(v) = f_e^{\text{EXP}}(w)$). An output dependence between v and w is valid in the expanded program iff the left-hand side of the expansion correctness criterion is false for v and w, i.e. iff v and w are not related by $\bowtie$. Such an output dependence is called a *neutral output dependence* [LF98]. The aim is to elaborate an expansion vector which gives to $\mathtt{A}_S$ an optimized but sufficient shape to only authorize neutral output dependences on S.

The dimension of $\mathtt{E}_S$ is equal to the number of loops surrounding S, written N_S. Each element $\mathtt{E}_S\mathtt{[}p+1\mathtt{]}$ is the *expansion degree* of S at depth p (the depth of the loop considered), with $p \in \{0, \ldots, N_S - 1\}$ and gives the size of dimension $(p+1)$ of $\mathtt{A}_S$. Each dimension of $\mathtt{A}_S$ must have a sufficient size to forbid any non-neutral output dependence. For a given access v, the set of instances which *may not write* in the same location as v can be deduced from the expansion correctness criterion (5.17), call it $W_p^S(v)$. It holds all instances w such that:

- w is an instance of S: $\textsc{Stmt}(w) = S$;
- $\textsc{Iter}(v)\mathtt{[}1..p\mathtt{]} = \textsc{Iter}(w)\mathtt{[}1..p\mathtt{]}$ and $\textsc{Iter}(v)\mathtt{[}p+1\mathtt{]} < \textsc{Iter}(w)\mathtt{[}p+1\mathtt{]}$;
- And $v \bowtie w$.

Let $w_p^S(v)$ be the lexicographic maximum of $W_p^S(v)$. For all w in $W_p^S(v)$, we have the following relations:

$$\begin{cases} \textsc{Iter}(v)\mathtt{[}1..p\mathtt{]} = \textsc{Iter}(w)\mathtt{[}1..p\mathtt{]} = \textsc{Iter}(w_p^S(v))\mathtt{[}1..p\mathtt{]} \\ \textsc{Iter}(v)\mathtt{[}p+1\mathtt{]} < \textsc{Iter}(w)\mathtt{[}p+1\mathtt{]} \le \textsc{Iter}(w_p^S(v))\mathtt{[}p+1\mathtt{]} \end{cases}$$

If $\mathtt{E}_S\mathtt{[}p+1\mathtt{]}$ is equal to $(\textsc{Iter}(w_p^S(v))\mathtt{[}p+1\mathtt{]} - \textsc{Iter}(v)\mathtt{[}p+1\mathtt{]} + 1)$ and knowing that the index function will be $\mathtt{A}_S\mathtt{[}\textsc{Iter}(v)\ \mathtt{\%}\ \mathtt{E}_S\mathtt{]}$, we ensure that no non-neutral output dependence appear between v and any instance of $W_p^S(v)$. But this property must be verified for each instance of S, and $\mathtt{E}_S$ should be set to the maximum of $(\textsc{Iter}(w_p^S(v))\,\mathtt{[}p+1\mathtt{]} - \textsc{Iter}(v)\,\mathtt{[}p+1\mathtt{]}+1)$ for all instances v of S. This proves that the following definition of $\mathtt{E}_S$ forbids any output dependence between instances of S in relation with $\bowtie$:

$$\mathtt{E}_S\,\mathtt{[}p+1\mathtt{]} = \max\left\{\textsc{Iter}(w_p^S(v))\,\mathtt{[}p+1\mathtt{]} - \textsc{Iter}(v)\,\mathtt{[}p+1\mathtt{]} + 1 : v \in \mathbf{W} \wedge \textsc{Stmt}(v) = S\right\} \tag{5.24}$$

Computing this for each dimension of $\mathtt{E}_S$ ensures that $\mathtt{A}_S$ has a sufficient size for the expansion to preserve the sequential program semantics. This is the purpose of Build-Expansion-Vector: *working* is relation $(v, W_p^S(v))$ and *maxv* is relation $(v, w_p^S(v))$. For a detailed proof, an intuitive introduction and related works, see [LF98, Lef98]. For the Build-Expansion-Vector algorithm, the simplest optimality concept is defined by the number of integer-valued components in $\mathtt{E}_S$, i.e. the number of "projected" dimensions, as proposed by Quilleré and Rajopadhye in [QR99]. But even with this simple definition, optimality is still an open problem. Since the algorithm proposed by [QR99] has been proven optimal, we should try to combine both techniques to yield better results, but his is left for future work.

Build-Expansion-Vector $(S, \bowtie)$
 S: the current statement
 $\bowtie$: the interference relation
 returns expansion vector $\mathtt{E}_S$ (a vector of integers or symbolic constants)
 $N_S \leftarrow$number of loops surrounding S
 for $p = 1$ **to** N_S
 do $\mathit{working} \leftarrow \{(v, w) : \langle S, v\rangle \in \mathbf{W} \wedge \langle S, w\rangle \in \mathbf{W}$
 $\wedge\; v[1..p] = w[1..p] \wedge v[1..p+1] < w[1..p+1]$
 $\wedge\; \langle S, v\rangle \bowtie \langle S, w\rangle\}$
 $\mathit{maxv} \leftarrow \{(v, \max_{<_{\text{LEX}}}\{w : (v, w) \in \mathit{working}\})\}$
 $\mathit{vector}[p+1] \leftarrow \max_{<_{\text{LEX}}}\{w - v[p+1] + 1 : (v, w) \in \mathit{maxv}\}$
 return *vector*

Now, a component of $\mathtt{E}_S$ computed by Build-Expansion-Vector can be a symbolic constant. When this constant can be proven "much smaller" than the associated dimension of iteration space of S, it is useful for reducing memory usage; but if such a result cannot be shown with the available compile-time information, the component is set to $+\infty$, meaning that no modulo computation should appear in the generated code (for this particular dimension). The interpretation of "much smaller" depends on the application: Lefebvre considered in [Lef98] that only integer constants where allowed in $\mathtt{E}_S$, but we believe that this requirement is too strong, as shown in the knapsack example (a modulo $K+1$ is needed).

Partial Renaming

Now every array $\mathtt{A}_S$ has been built, one can perform an additional storage reduction to the generated code. Indeed, for two statements S and T, partial expansion builds two structures $\mathtt{A}_S$ and $\mathtt{A}_T$ which can have different shapes. If at the end of the renaming process S and T are authorized to share the same array, this one would have to be the rectangular hull of $\mathtt{A}_S$ and $\mathtt{A}_T$: $\mathtt{A}_{ST}$. It is clear that these two statements can share the same data iff this sharing is not contradictory with the expansion correctness criterion for instances of S and T. One must verify for every instance u of S and v of T, that the value produced by u (resp. v) cannot be killed by v (resp. u) before it stops being useful.

Finding the minimal renaming is NP-complete. Our method consists in building a graph similar to an interference graph as used in the classic register allocation process. In this graph, each vertex represents a statement of the program. There is an edge between two vertices S and T iff it has been shown that they cannot share the same data structure in their left-hand sides. Then one applies on this graph a greedy coloring algorithm. Finally it is clear that vertices that have the same color can have the same data structure. This partial renaming algorithm is sketched in Partial-Renaming (the Greedy-Coloring algorithm returns a function mapping each statement to a color).

```
Partial-Renaming (program, ⋈)
    program: the program where partial renaming is required
    ⋈: the interference relation
    returns the program with coalesced data structures
    for each array A in program
    do interfere ← ∅
        for each pair of statements S and T assigning A in program
        do if ∃⟨S,v⟩,⟨T,w⟩ ∈ W : ⟨S,v⟩ ⋈ ⟨T,w⟩
                then interfere ← interfere ∪ {(S,T)}
        coloring ← Greedy-Coloring (interfere)
        for each statements S assigning A in program
        do left-hand side A[subscript] of S ← A_coloring(S)[subscript]
    return program
```

5.3.6 Dealing with Tiled Parallel Programs

The partial expansion algorithm often yields poor results, especially on tiled programs. The reason is that subscripts of expanded arrays are of the form `A`$_S$`[`*`subscript`* `%` `E`$_S$`]`, and the *block regularity* of tiled programs does not really fit in this *cyclic pattern*. Figure 5.24 shows an example of what we would like to achieve on some block-regular expansions. No cyclic folding would be possible on such an example, since the two outer loops are parallel.

The design of an improved graph coloring algorithm able to consider *both block and cyclic patterns* is still an open problem, because it requires non-affine constraints to be optimized. We only propose a work-around, which works when some a priori knowledge on the tile shape is available. The technique consists in dividing each dimension with the associated tile size. Sometimes, the resulting storage mapping will be compatible with the required parallel execution, and sometimes not: decision is made with Theorem 5.2. Expanded array subscripts are thus of the form `A`$_S$`[`$i_1/shape_1$`, ··· ,` $i_N/shape_N$`]`, where $(i_1, \ldots, i_N)$ is the iteration vector associated with CurIns (defined in Section 5.1), and where $shape_i$ is either 1 or the size of the i^{th} dimension of the tile.

It is possible to improve this technique in combining divisions and modulo operations, but the expansion scheme is somewhat different: see Section 5.4.6.

```
   int x;
   for (i=0; i<N; i++)
     for (j=0; j<N; j++) {
S      x = ···;
R      ··· = x ···;
     }
```

Figure 5.24.a. Original program

```
   int xS[N, N];
   for (i=0; i<N; i++)
     for (j=0; j<N; j++) {
S      xS[i, j] = ···;
R      ··· = xS[i, j] ···;
     }
```

Figure 5.24.b. Single-assignment program

```
   int xS[N/16, N/16];
   parallel for (i=0; i<N; i+=16)
     parallel for (j=0; j<N; j+=16)
       for (ii=0; ii<16; ii++)
         for (jj=0; jj<16; jj++) {
S          xS[i/16, j/16] = ···;
R          ··· = xS[i/16, j/16] ···;
         }
```

Figure 5.24.c. Partially expanded tiled program

Figure 5.24. An example of block-regular storage mapping

5.3.7 Schedule-Independent Storage Mappings

The technique presented in Section 5.3.4 yields the best results, but involves an external parallelization technique, such as scheduling or tiling. It is well suited to parallelizing compilers.

A *schedule-independent* (a.k.a. universal) storage mapping [SCFS98] is useful whenever no parallel execution scheme is enforced. The aim is to preserve the "portability" of SA form, at a much lower cost in memory usage.

From the definition of $\bowtie$—the interference relation—in (5.21), and considering two parallel execution orders $<^1_{\text{PAR}}$ and $<^2_{\text{PAR}}$ whose associated interference relations are $\bowtie^1$ and $\bowtie^2$, we have:

$$<^1_{\text{PAR}} \subseteq <^2_{\text{PAR}} \Longrightarrow \bowtie^2 \subseteq \bowtie^1 .$$

Now, a schedule-independent storage mapping f_e^{EXP} must be compatible with any possible parallel execution $<_{\text{PAR}}$ of the program. Partial order $<_{\text{PAR}}$ used in the Storage-Mapping-Optimization algorithm should thus be *included in any correct execution order*. By definition of correct execution orders—Theorem 2.2 page 81—this condition is satisfied by the ***data-flow execution order***, which is the transitive closure of the reaching definition relation: σ^+.

Section 3.1.2 describes a way to compute the transitive closure of σ (useful remarks and experimental study are also presented in Section 5.2.5). In general, no exact result can be hoped for the data-flow execution order σ^+, because Presburger arithmetic is not closed under transitive closure. Hence, we need to compute an *approximate relation*. Because the approximation must be included in *all possible correct execution order*, we want it to be a *sub-order* of the exact data-flow order (i.e. the opposite of a conservative approximation). Such an approximation can be computed with Omega [Pug92].

5.3.8 Dynamic Restoration of the Data-Flow

Implementing ϕ functions for a partially expanded program is not very different from what we have seen in Section 5.1.3. Indeed, algorithm LOOP-NESTS-IMPLEMENT-PHI applies without modification. But doing this, no storage mapping optimization is performed on Φ-arrays. Now, remember Φ-arrays are supposed to be in one-to-one mapping with expanded data structures. Single-assignment Φ-arrays are not necessary to preserve the semantics of the original program, since the same dependences will be shared by expanded arrays and Φ-arrays.

The resulting code generation algorithm is very similar to LOOP-NESTS-IMPLEMENT-PHI. The first step consists in replacing every reference to $\Phi\mathtt{A}_S\mathtt{[}x\mathtt{]}$ with its "folded" counterpart $\Phi\mathtt{A}_S\mathtt{[}x\ \mathtt{\%}\ \mathtt{E}_S\mathtt{]}$. In a second step, one merge Φ-arrays together using the result of algorithm PARTIAL-RENAMING.

Eventually, for a given ϕ function, the set of possible reaching definitions should be reconsidered: values produced by a few instances may now be overwritten, according to the new storage mapping. As in the motivating example, the ϕ function can even disappear, see Figure 5.17. A good technique to automatically achieve this is not to perform a new reaching definition analysis. One should update the available sets of reaching definitions: a $\phi(set)$ reference should be replaced by

$$\phi\big(\{v \in set : \nexists w \in set : v <_{\text{SEQ}} w \wedge f_e^{\text{EXP}}(v) = f_e^{\text{EXP}}(w)\}\big).$$

Moreover, if *coloring* is the result of the greedy graph coloring algorithm in PARTIAL-RENAMING, $f_e^{\text{EXP}}(\langle s, x\rangle) = f_e^{\text{EXP}}(\langle s', x'\rangle)$ is equivalent to

$$coloring(s) = coloring(s') \wedge (x \mod \mathtt{E}_s = x' \mod \mathtt{E}_{s'}).$$

5.3.9 Back to the Examples

First Example

Using the Omega Calculator text-based interface, we describe a step-by-step execution of the expansion algorithm. We have to code instances as integer-valued vectors. An instance $\langle s, i\rangle$ is denoted by vector `[i, ..., s]`, where `[...]` possibly pads the vector with zeroes. We number T, S, R with 1, 2, 3 in this order, so $\langle T, i\rangle$, $\langle S, i, j\rangle$ and $\langle R, i\rangle$ are written `[i,0,1]`, `[i,j,2]` and `[i,0,3]`, respectively.

Schedule-dependent storage mapping. We first apply the partial expansion algorithm according to the parallel execution order proposed in Figure 5.17.

The result of instancewise reaching definition analysis is written in Omega's syntax:

```
S := {[i,0,2]->[i,0,1] : 1<=i<=N}
     union {[i,w,2]->[i,w-1,2] : 1<=i<=N && 1<=w}
     union {[i,0,3]->[i,0,1] : 1<=i<=N}
     union {[i,0,3]->[i,w,2] : 1<=i<=N && 0<=w};
```

The no-conflict relation is trivial here, since the only data structure is a scalar variable:

```
NCon := {[i,w,s]->[i',w',s'] : 1=2}; # 1=2 means FALSE!
```

We consider that the outer loop is parallel. It gives the following execution order:

```
Par := {[i,w,2] -> [i,w',2] : 1 <= i <= N && 0 <= w < w'} union
       {[i,0,1] -> [i,w',2] : 1 <= i <= N && 0 <= w'} union
       {[i,0,1] -> [i,0,3] : 1 <= i <= N} union
       {[i,w,2] -> [i,0,3] : 1 <= i <= N && 0 <= w};
```

We have to compute relation $\bowtie$ in left-hand side of the expansion correctness criterion, call it `Int`.

```
# The "full" relation
Full := {[i,w,s]->[i',w',s'] : 1<=s<=3 && (s=2 || w=w'=0)
         && 1<=i,i'<=N && 0<=w,w'};

# The sequential execution order
Lex := {[i,w,2]->[i',w',2] : 1<=i<=i'<=N && 0<=w,w' && (i<i' || w<w')}
       union {[i,0,1]->[i',0,1] : 1<=i<i'<=N}
       union {[i,0,3]->[i',0,3] : 1<=i<i'<=N}
       union {[i,0,1]->[i',w',2] : 1<=i<=i'<=N && 0<=w'}
       union {[i,w,2]->[i',0,1] : 1<=i,i'<=N && 0<=w && i<i'}
       union {[i,0,1]->[i',0,3] : 1<=i<=i'<=N}
       union {[i,0,3]->[i',0,1] : 1<=i<i'<=N}
       union {[i,w,2]->[i',0,3] : 1<=i<=i'<=N && 0<=w}
       union {[i,0,3]->[i',w',2] : 1<=i<i'<=N && 0<=w'};
ILex := inverse Lex;

NPar := Full - Par;
INpar := inverse NPar;

Int := (INPar intersection (ILex union NCon))
       union (INPar intersection S(NPar intersection Lex));
Int := Int union (inverse Int);
```

The result is:

```
Int;

{[i,w,2] -> [i',w',2] : 1 <= i' < i <= N && 1 <= w <= w'} union
{[i,0,2] -> [i',w',2] : 1 <= i' < i <= N && 0 <= w'} union
{[i,w,2] -> [i',w-1,2] : 1 <= i' < i <= N && 1 <= w} union
{[i,w,2] -> [i',w',2] : 1 <= i' < i <= N && 0 <= w' <= w-2} union
{[i,0,1] -> [i',0,1] : 1 <= i' < i <= N} union
{[i,0,2] -> [i',0,1] : 1 <= i' < i <= N} union
{[i,0,1] -> [i',w',2] : 1 <= i' < i <= N && 0 <= w'} union
{[i,0,3] -> [i',0,1] : 1 <= i' < i <= N} union
{[i,0,3] -> [i',w',2] : 1 <= i' < i <= N && 0 <= w'} union
{[i,w,2] -> [i',0,3] : 1 <= i < i' <= N && 0 <= w} union
{[i,0,1] -> [i',0,3] : 1 <= i < i' <= N} union
{[i,w,2] -> [i',0,1] : 1 <= i < i' <= N && 0 <= w} union
{[i,0,1] -> [i',0,2] : 1 <= i < i' <= N} union
```

```
{[i,0,1] -> [i',0,1] : 1 <= i < i' <= N} union
{[i,w,2] -> [i',w',2] : 1 <= i < i' <= N && 0 <= w <= w'-2} union
{[i,w,2] -> [i',w+1,2] : 1 <= i < i' <= N && 0 <= w} union
{[i,w,2] -> [i',0,2] : 1 <= i < i' <= N && 0 <= w} union
{[i,w,2] -> [i',w',2] : 1 <= i < i' <= N && 1 <= w' <= w}
```

A quick verification shows that

```
Int intersection {[i,w,s]->[i,w',s']}
```

is empty, meaning that neither expansion nor renaming must be done inside an iteration of the outer loop. In particular: $\mathtt{E}_S\mathtt{[2]}$ should be set to 0. However, computing the set $W_0^S(v)$ (i.e. for the outer loop) yields all accesses w executing after v (for the same i). Then $\mathtt{E}_S\mathtt{[1]}$ should be set to N. We have automatically found the partially expanded program.

Schedule-independent storage mapping. We now apply the expansion algorithm according to the "data-flow" execution order. The parallel execution order is defined as follows:

```
Par := S+;
```

Once again

```
Int intersection {[i,w,s]->[i,w',s']}
```

is empty. The schedule-independent storage mapping is thus the same as the previous, parallelization-dependent, one.

The resulting program for both techniques is the same as the hand-crafted one in Figure 5.17.

Second Example

We now consider the knapsack program in Figure 5.18. It is easy to show that a schedule-independent storage mapping would give no better result that single-assignment form. More precisely, it is impossible to find any schedule such that a "cyclic folding"—a storage mapping with subscripts of the form $\mathtt{A}_S\mathtt{[CurIns \% E}_S\mathtt{]}$—would be more economical than single-assignment form.

We are thus looking for a schedule-dependent storage mapping. An efficient parallelization of program KP requires tiling of the iteration space. This can be done using classical techniques since the loop is perfectly nested. Section 5.3.10 has shown good performance for 16×32 tiles, but we consider 2×1 tiles for the sake of simplicity. The parallel execution order considered is the same as the one presented in Section 5.3.1: tiles are scheduled in fronts of constant $k + j$, and the inner-tile order is the original sequential execution one.

The result of instancewise reaching definition analysis is written in Omega's syntax:

```
S := {[k,j]->[k-1,j] : 2<=k<=M && 1<=j<=C} union
     {[k,j]->[k,j'] : 1<=k<=M && 1<=j'<j<=C && j'-K<=j};
```

Instances which may not assign the same memory location are defined by the following relation:

```
NCon := {[k,j]->[k',j'] : 1<=k,k'<=M && 1<=j,j'<=C && j!=j'};
```

Considering the 2×1 tiling, it is easy to compute $<_{\text{PAR}}$:

```
InnerTile := {[k,j]->[k',j] : (exists kq,kr,kr' : k=2kq+kr
  && k'=2kq+kr' && 0<=kr<kr'<2)};
InterTile := {[k,j]->[k',j'] : (exists kq,kr,kq',kr' : k=2kq+kr
  && k'=2kq'+kr' && 0<=kr,kr'<2 && kq+j<kq'+j')};
Par := Lex intersection (InnerTile union InterTile);
```

We have to compute relation $\bowtie$ in left-hand side of the expansion correctness criterion, call it `Int`.

```
# The "full" relation
Full := {[k,j]->[k',j'] : 1<=k,k'<=M && 1<=j,j'<=C};

# The sequential execution order
Lex := Full intersection {[k,j]->[k',j'] : k<k' || (k=k' && j<j')};
ILex := inverse Lex;

NPar := Full - Par;
INpar := inverse NPar;

Int := (INPar intersection (ILex union NCon))
       union (INPar intersection S(NPar intersection Lex));
Int := Int union (inverse Int);
```

The result is:

```
Int;

{[k,j] -> [k',j'] : 1 <= k <= k' <= M && 1 <= j < j' <= C} union
{[k,j] -> [k',j'] : 1 <= k < k' <= M && 1 <= j' < j <= C} union
{[k,j] -> [k',j'] : Exists ( alpha : 1, 2alpha+2 <= k < k' < M
 && j <= C && 1 <= j' && k'+2j' <= 2+2j+2alpha)} union
{[k,j] -> [k',j'] : Exists ( alpha : 1, 2alpha+2 <= k' < k < M
 && j' <= C && 1 <= j && k+2j <= 2+2j'+2alpha)} union
{[k,j] -> [k',j'] : 1 <= j < j' <= C && 1 <= k' < k <= M} union
{[k,j] -> [k',j'] : 1 <= k' <= k <= M && 1 <= j' < j <= C}
```

A quick verification shows that

```
Int intersection {[k,j]->[k+K+1,j']}
```

is empty, meaning that `E`$_S$`[1]` should be set to $K + 1$.

5.3.10 Experiments

Partial expansion has been implemented for Cray-Fortran affine loop nests [LF98]. Semi-automatic storage mapping optimization has also been performed on general loop-nests, using FADA, Omega, and PIP.

Figure 5.25 summarizes expansion and parallelization results for several programs. The three affine loop nests examples have already been studied by Lefebvre in [LF98,

	Sequential		Parallel	Parallel Size		Run-time Overhead	
Program	Complexity	Size	Complexity	SA	Optimized	SA	Optimized
MVProduct	$\mathcal{O}(N^2)$	N^2+2N+1	$\mathcal{O}(N)$	$2N^2+3N$	N^2+2N	no ϕ	no ϕ
Cholesky	$\mathcal{O}(N^3)$	N^2+N+1	$\mathcal{O}(N)$	N^3+N^2	$2N^2+N$	no ϕ	no ϕ
Gaussian	$\mathcal{O}(N^3)$	N^2+N+1	$\mathcal{O}(N)$	N^3+N^2+N	$2N^2+2N$	no ϕ	no ϕ
Knapsack	$\mathcal{O}(MC)$	$C+2M$	$\mathcal{O}(M+C)$	$MC+C+2M$	$KC+2C+2M$	free ϕ	free ϕ
Convolution	$\mathcal{O}(NM)$	1	$\mathcal{O}(M)$	$NM+N$	N	cheap ϕ	no ϕ

Figure 5.25. Time and space optimization

Lef98]: matrix-vector product, Cholesky factorization and Gaussian elimination. A few experiments have been made on an SGI Origin 2000, using the `mp` library (but not `PCA`, the built-in automatic parallelizer). As one would expect, results for the convolution program are excellent even for small values of `N`. Execution times for program KP appear in Figure 5.26. The first graph compares execution time of the parallel program and of the *original (not expanded) one*; the second one shows the speed-up. We got very good results for medium array sizes,[17] both in terms of speed-up and relatively to the original knapsack program.

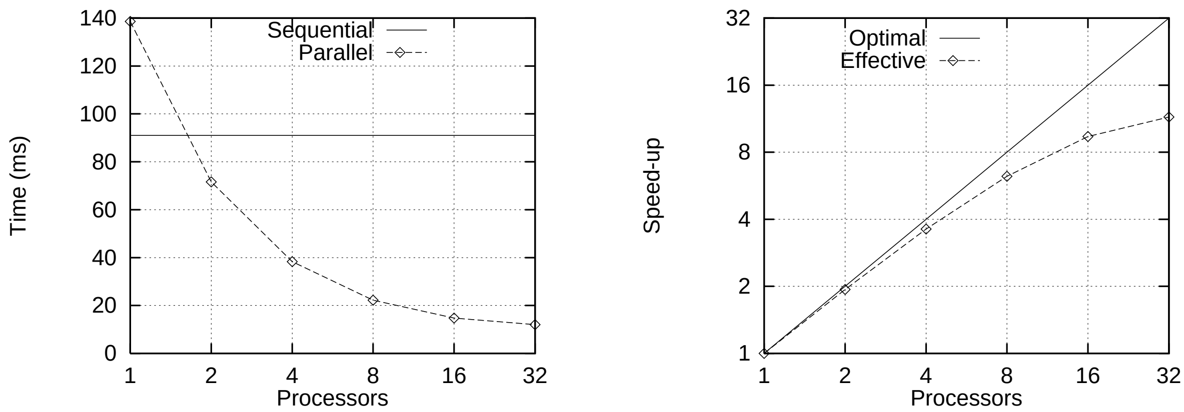


Figure 5.26. Performance results

5.4 Constrained Storage Mapping Optimization

Sections 5.2 and 5.3 addressed two techniques to optimize parallelization via memory expansion. We show here that combining the two techniques in a more general expansion framework is possible and brings significant improvements. Optimization is achieved from two complementary directions:

- Adding *constraints* to limit memory expansion, like *static expansion* avoiding ϕ-functions [BCC98], *privatization* [TP93, MAL93], or *array static single assignment* [KS98]. All these techniques allow partial removal of memory-based dependences, but may extract less parallelism than conversion to single assignment form.

[17] Here `C=2048`, `M=1024` and `K=16`, with 16×32 tiles (scheduled similarly to Figure 5.18).

- Applying *storage mapping optimization* techniques [CL99]. Some of these are either schedule-independent [SCFS98] or schedule-dependent [LF98] (yielding better optimizations) whether they require former computation of a parallel execution order (scheduling, tiling, etc.) or not.

We try here to get the best of both directions and show the benefit of combining them into a unified framework for memory expansion. The motivation for such a framework is the following: because of the increased complexity of dealing with irregular codes, and given the wide range of parameters which can be tuned when parallelizing such programs, a broad range of expansion techniques have been or will be designed for optimizing one or a few of these parameters. The two preceding sections are some of the best examples of this trend. We believe that our constrained expansion framework greatly reduces the complexity of the optimization problem, in reducing the number of parameters and helping the automation process.

With the help of a motivating example we introduce the general concepts, before we formally define correct *constrained storage mappings*. Then, we present an intra-procedural algorithm which handles any imperative program and most loop nest parallelization techniques.

5.4.1 Motivation

We study the pseudo-code in Figure 5.27.a. Such nested loops with conditionals appear in many kernels, but most parallelization techniques fail to generate efficient code for these programs. Instances of T are denoted by $\langle T, i, j\rangle$, instances of S by $\langle S, i, j, k\rangle$, and instances of R by $\langle R, i\rangle$, for $1 \le i, j \le M$ and $1 \le k \le N$. ("$P(i, j)$" is a boolean function of i and j.)

```
    double x;
    for (i=1; i<=M; i++) {
      for (j=1; j<=M; j++)
        if (P(i,j)) {
T         x = 0;
          for (k=1; k<=N; k++)
S           x = x ···;
        }
R     ··· = x ···;
    }
```

Figure 5.27.a. Original program

```
    double x_T[M+1, M+1], x_S[M+1, M+1, N+1];
    for (i=1; i<=M; i++) {
      for (j=1; j<=M; j++)
        if (P(i,j)) {
T         x_T[i, j] = 0;
          for (k=1; k<=N; k++)
S           x_S[i, j, k] = if (k==1) x_T[i, j];
              else x_S[i, j, k-1] ···;
        }
R     ··· = φ({⟨S,i,1,N⟩, ..., ⟨S,i,M,N⟩}) ···;
    }
```

Figure 5.27.b. Single assignment form

Figure 5.27. Motivating example

On this example, assume N is positive and predicate "$P(i, j)$" evaluates to true at least one time for each iteration of the outer loop. A precise instancewise reaching definition analysis tells us that the reaching definition of the read access $\langle S, i, j, k\rangle$ to `x` is $\langle T, i, j\rangle$ when $k = 1$ and $\langle S, i, j, k-1\rangle$ when $k > 1$. We only get an approximate result for definitions that *may reach* $\langle R, i\rangle$: those are $\{\langle S, i, 1, N\rangle, \ldots, \langle S, i, M, N\rangle\}$. In fact, the value of x may only come from S (since $N > 0$) for the same i (since T executes at least one time for each iteration of the outer loop), and for $k = N$.

Obviously, memory-based dependences on x hampers parallelization. Our intent is to expand scalar x so as to get rid of as many dependences as possible. Figure 5.27.b shows our program converted to SA form. The unique ϕ function implements a run-time choice between values produced by $\langle S, i, 1, N\rangle, \ldots, \langle S, i, M, N\rangle$.

SA removed enough dependences to make the two outer loops parallel, see Figure 5.28.a. Function ϕ is computed at run-time using array @x. It holds the last value of j at statement S when x was assigned. This information allows value recovery in R, see the third method in Section 5.1.4 for details.

But this parallel program is not usable on any architecture. The main reason is memory usage: variable x has been replaced by a huge three-dimensional array, plus two smaller arrays. This code is approximately five times slower than the original program on a single processor (when arrays can be accomodated in memory).

```
  double xT[M+1, M+1], xS[M+1, M+1, N+1];
  int @x[M+1];
  parallel for (i=1; i<=M; i++) {
    @x[i] = ⊥;
    parallel for (j=1; j<=M; j++)
      if (P(i,j)) {
T       xT[i, j] = 0;
        for (k=1; k<=N; k++)
S         xS[i, j, k] = if (k==1)
              xT[i, j];
            else xS[i, j, k-1] ···;
        @x[i] = max (@x[i], j);
      }
R   ··· = xS[i, @x[i], N] ···;
  }
```

Figure 5.28.a. Parallel SA

```
  double x[M+1, M+1];
  int @x[M+1];
  parallel for (i=1; i<=M; i++) {
    @x[i] = ⊥;
    parallel for (j=1; j<=M; j++)
      if (P(i,j)) {
T       x[i, j] = 0;
        for (k=1; k<=N; k++)
S         x[i, j] = x[i, j] ···;
        @x[i] = max (@x[i], j);
      }
R   ··· = x[i, @x[i]] ···;
  }
```

Figure 5.28.b. Parallel SMO

Figure 5.28. Parallelization of the motivating example

This shows the need for a memory usage optimization technique. Storage mapping optimization (SMO) [CL99, LF98, SCFS98] consists in reducing memory usage as much as possible as soon as a parallel execution order has been crafted, see Section 5.3. A single two-dimensional array can be used, while keeping the two outer loops parallel, see Figure 5.28.b. Run-time computation of function ϕ with array @x seems very cheap at first glance, but execution of @x[i] = max (@x[i], j) hides *synchronizations* behind the computation of the maximum! As usual, it results in a very bad scaling: good accelerations are obtained for a very small number of processors, then speed-up drops dramatically because of synchronizations. Figure 5.29 gives execution time and speed-up for the parallel program, compared to the original—not expanded—one. We used the mp library on an SGI Origin 2000, with $M = 64$ and $N = 2048$, and simple expressions for "···" parts.

This bad result shows the need for a finer parallelization scheme. The question is to

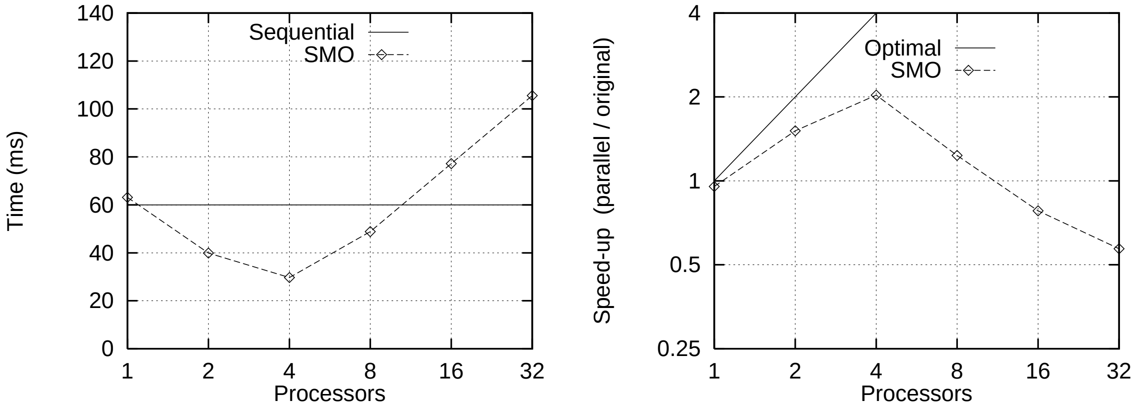


Figure 5.29. Performance results for storage mapping optimization

find a good tradeoff between expansion overhead and parallelism extraction. If we target widely-used parallel computers, the processor number is likely to be less than 100, but SA form extracted *two* parallel loops involving M^2 processors! The intuition is that we wasted memory and run-time overhead.

One would prefer a pragmatic expansion scheme, such as *maximal static expansion* (MSE) [BCC98], or *privatization* [TP93, MAL93]. Choosing static expansion has the benefit that no ϕ function is necessary any more: `x` can be safely expanded along outermost and innermost loops, but expansion along `j` is forbidden—it requires a ϕ function thus violates the *static* constraint, see Section 5.2. Now, only the outer loop is parallel, and we get much better scaling, see Figure 5.30. However, on a single processor the program still runs two times slower than the original one: scalar `x`—probably promoted to a register in the original program—has been replaced by a two-dimensional array.

```
    double x[M+1, N+1];
    parallel for (i=1; i<=M; i++) {
      for (j=1; j<=M; j++)
        if (P(i,j)) {
T         x[i, 0] = 0;
          for (k=1; k<=N; k++)
S           x[i, k] = x[i, k-1] ···;
        }
R     ··· = x[i, N] ···;
    }
```

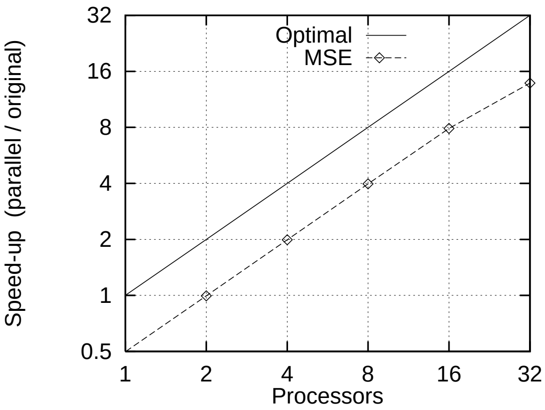


Figure 5.30. Maximal static expansion

Maximal static expansion expanded `x` along the innermost loop, but it was of no interest regarding parallelism extraction. Combining it with storage mapping optimization solves the problem, see Figure 5.31. Scaling is excellent and parallelization overhead is very low: the parallel program runs 31.5 times faster than the *original* one on 32 processors (for $M = 64$ and $N = 2048$).

This example shows the use of combining constrained expansions—such as privatization and static expansion—with storage mapping optimization techniques, to improve parallelization of general loop nests (with unrestricted conditionals and array subscripts). In the following, we present an algorithm useful for automatic parallelization of imperative programs. Although this algorithm cannot itself choose the "best" parallelization, it aims to *simultaneous optimization of expansion and parallelization constraints*.

```
     double x[M+1];
     parallel for (i=1; i<=M; i++) {
       for (j=1; j<=M; j++)
         if (P(i,j)) {
T          x[i] = 0;
           for (k=1; k<=N; k++)
S            x[i] = x[i] ···;
         }
R      ··· = x[i] ···;
     }
```

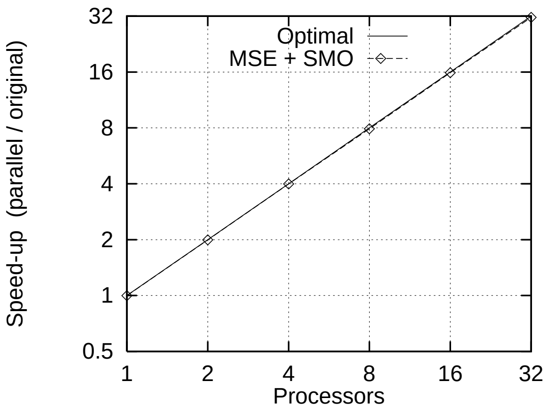


Figure 5.31. Maximal static expansion combined with storage mapping optimization

5.4.2 Problem Statement

Because our framework is based on maximal static expansion and storage mapping optimization, we inherit their program model and mathematical abstraction: we only consider nests of loops operating on arrays and abstract these programs with affine relations.

Introducing Constrained Expansion

The motivating example shows the benefits of putting an a priori limit to expansion. Static expansion [BCC98] is a good example of *constrained expansion*. What about other expansion schemes? The goal of constrained expansion is to design pragmatic techniques that does not expand variables when the incurred overhead is "too high". To generalize static expansion, we suppose that some equivalence relation $\equiv$ on writes is available from previous compilation stages—possibly with user interaction. It is called the *constraint relation*. A storage mapping *constrained by* $\equiv$ is any mapping f_e^{EXP} such that

$$\forall e \in \mathbf{E}, \forall v, w \in \mathbf{W}: \quad v \equiv w \wedge f_e(v) = f_e(w) \implies f_e^{\text{EXP}}(v) = f_e^{\text{EXP}}(w). \tag{5.25}$$

It is difficult to decide whether to forbid expansion of some variable or not. A short survey of this problem is presented in Section 5.4.5, along with a discussion about building constraint relation $\equiv$ from a "syntactical" or "semantical" constraint. Moreover, we leave for Section 5.4.8 all discussions about picking the right parallel execution order.

Now, the two problems are part of the same two-criteria optimization problem: tuning expansion and parallelism for performance. We *do not* present here a solution to this complex problem. The algorithm described in the next sections should be seen as an integrated *tool for parallelization*, as soon as the "strategy" has been chosen—what expansion constraints, what kind of schedule, tiling, etc. Most of these strategies have already been shown useful and practical for some programs; *our main contribution is their integration in an automatic optimization process.* The summary of our optimization framework is presented in Figure 5.32.

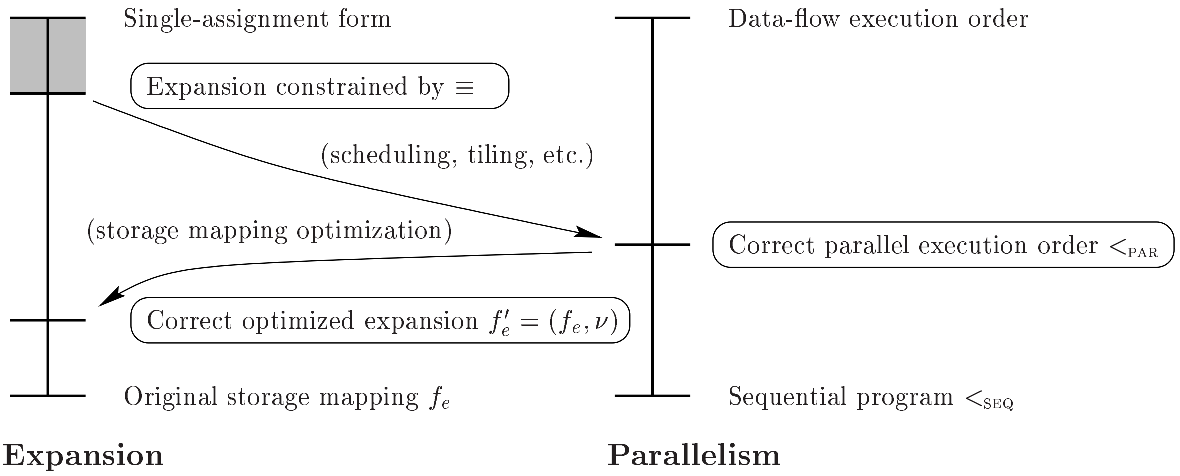


Figure 5.32. What we want to achieve

5.4.3 Formal Solution

We first define correct parallelizations then state our optimization problem.

What is a Correct Parallel Execution Order?

Memory expansion partially removes dependences due to memory reuse. Recall from Section 2.5 that relation δ^{EXP} approximates the dependence relation of $(<_{\text{SEQ}}, f_e^{\text{EXP}})$, the *expanded* program with *sequential* execution order. (δ^{EXP} equals σ when the program is converted to SA form.) Thanks to Theorem 2.2 page 81, we want any parallel execution order $<_{\text{PAR}}$ to satisfy the following condition:

$$\forall (\imath_1, r_1), (\imath_2, r_2) \in \mathbf{A} : \quad (\imath_1, r_1)\, \delta^{\text{EXP}}\, (\imath_2, r_2) \implies \imath_1 <_{\text{PAR}} \imath_2. \tag{5.26}$$

Computation of approximate dependence relation δ^{EXP} from storage mapping f_e^{EXP} is presented in Section 5.4.8.

What is a Correct Expansion?

Given parallel order $<_{\text{PAR}}$, we are looking for *correct expansions* allowing parallel execution to preserve original semantics. Our task is to formalize memory reuse constraints enforced by $<_{\text{PAR}}$. Using *interference relation* $\bowtie$ defined in Section 5.3.2, we have proven in Theorem 5.2 that the expansion is correct if the following condition holds.

$$\forall e \in \mathbf{E}, \forall v, w \in \mathbf{W} : \quad v \bowtie w \implies f_e^{\text{EXP}}(v) \neq f_e^{\text{EXP}}(w). \tag{5.27}$$

Computing Parallel Execution Orders and Expansions

We formalized the parallelization correctness with an expansion constraint (5.25) and two correctness criteria (5.26) and (5.27). Let us show how solving these equations simultaneously yields a suitable parallel program $(<_{\text{PAR}}, f_e^{\text{EXP}})$.

Following the lines of Section 5.2.3, we are interested in removing as many dependences as possible, without violating the expansion constraint. We can prove—like Proposition 5.1 in Section 5.2.3—that a constrained expansion is *maximal*—i.e. assigns the largest number of memory locations while verifying (5.25)—iff

$$\forall e \in \mathbf{E}, \forall v, w \in \mathbf{W}_e : \quad v \equiv w \wedge f_e(v) = f_e(w) \iff f_e^{\text{EXP}}(v) = f_e^{\text{EXP}}(w).$$

Still following Section 5.2.3, we assume that $f_e^{\text{EXP}} = (f_e, \nu)$, where ν is constant on equivalence classes of $\equiv$. Indeed, if $f_e(v) = f_e(w)$, condition $f_e^{\text{EXP}}(v) = f_e^{\text{EXP}}(w)$ becomes equivalent to $\nu(v) = \nu(w)$. Because we need to approximate over all possible executions, we use conflict relation κ^*, and our maximal constrained expansion criterion becomes

$$\forall v, w \in \mathbf{W}, v \,\kappa^*\, w : \quad v \equiv w \iff \nu(v) = \nu(w) \tag{5.28}$$

Computing ν is done by enumerating equivalence classes of $\equiv$. For any access v in a class of κ^* (instances that "may" hit the same memory location), $\nu(v)$ can be defined via a *representative* of the equivalence class of v for relation $\equiv$. Computing the lexicographical minimum is a simple way to find representatives, see Section 5.2.5.

It is time to compute dependences δ^{EXP} of program $(<_{\text{SEQ}}, f_e^{\text{EXP}})$: an access w depends on v if they hit the same memory location, v executes before w, and at least one is a write. The full computation is done in Section 5.4.8 and uses (5.28); the result is

$$\begin{aligned}
\forall v \in \mathbf{W}, w \in \mathbf{R} : &\quad v \,\delta^{\text{EXP}}\, w \;\Leftrightarrow\; \big(\exists u \in \mathbf{W} : u \,\sigma\, w \wedge v \,\kappa\, u \wedge v \equiv u\big) \wedge v <_{\text{SEQ}} w \\
\forall v \in \mathbf{R}, w \in \mathbf{W} : &\quad v \,\delta^{\text{EXP}}\, w \;\Leftrightarrow\; \big(\exists u \in \mathbf{W} : u \,\sigma\, v \wedge u \,\kappa\, w \wedge u \equiv w\big) \wedge v <_{\text{SEQ}} w \\
\forall v, w \in \mathbf{W} : &\quad v \,\delta^{\text{EXP}}\, w \;\Leftrightarrow\; v \,\kappa\, w \wedge v \equiv w \wedge v <_{\text{SEQ}} w
\end{aligned} \tag{5.29}$$

We rely on classical algorithms to compute $<_{\text{PAR}}$ from δ^{EXP} [Fea92, DV97, IT88, CFH95].

Knowing $(<_{\text{PAR}}, f_e^{\text{EXP}})$, we could stop and say we have successfully parallelized our program; but nothing ensures that f_e^{EXP} is an "economical" storage mapping (remember the motivating example). We must build a new expansion from $<_{\text{PAR}}$ that *minimizes* memory usage while satisfying (5.27).

For constrained expansion purposes, f_e^{EXP} has been chosen of the form (f_e, ν). This has some consequences on the expansion correctness criterion: when $f_e(v) \neq f_e(w)$, it is not necessary to set $\nu(v) \neq \nu(w)$ to enforce $f_e^{\text{EXP}}(v) \neq f_e^{\text{EXP}}(w)$. As a consequence, the $v \not\kappa w$ clause in (5.22) is not necessary any more (see page 194), and we may rewrite the expansion correctness criterion thanks to a simplified definition of interference relation $\bowtie$. Let ∞ be the interference relation for constrained expansion:

$$\begin{aligned}
v \infty w \;\stackrel{\text{def}}{\iff}\; & \big(\exists u \in \mathbf{R} : v \,\sigma\, u \wedge w \not<_{\text{PAR}} v \wedge u \not<_{\text{PAR}} w \wedge (u <_{\text{SEQ}} w \vee w <_{\text{SEQ}} v)\big) \\
\vee \quad & \big(\exists u \in \mathbf{R} : w \,\sigma\, u \wedge v \not<_{\text{PAR}} w \wedge u \not<_{\text{PAR}} v \wedge (u <_{\text{SEQ}} v \vee v <_{\text{SEQ}} w)\big).
\end{aligned} \tag{5.30}$$

We can rewrite this definition using algebraic operations:

$$\begin{aligned}
\infty \;=\; & \big((\sigma(\mathbf{R}) \times \mathbf{W}) \cap \not>_{\text{PAR}} \cap >_{\text{SEQ}}\big) \cup \big(\not>_{\text{PAR}} \cap (\sigma \circ (\not<_{\text{PAR}} \cap <_{\text{SEQ}}))\big) \\
\cup \quad & \big((\sigma(\mathbf{R}) \times \mathbf{W}) \cap \not<_{\text{PAR}} \cap <_{\text{SEQ}}\big) \cup \big(\not<_{\text{PAR}} \cap (\sigma \circ (\not<_{\text{PAR}} \cap <_{\text{SEQ}}))\big).
\end{aligned} \tag{5.31}$$

Theorem 5.3 (correctness of constrained storage mappings) If a storage mapping f_e^{EXP} is of the form (f_e, ν) and the following condition holds, then f_e^{EXP} is a correct expansion of f_e—i.e. f_e^{EXP} allows parallel execution to preserve the program semantics.

$$\forall v, w \in \mathbf{W}, v \,\kappa\, w : \quad v \infty w \implies \nu(v) \neq \nu(w). \tag{5.32}$$

Proving Theorem 5.3 is a straightforward rewriting of the proof of Theorem 5.2 and the optimality result of Proposition 5.2 also holds: the only difference is that the $v \not\bowtie w$ clause has been replaced by $v \,\kappa\, w$ in left-hand side of (5.32).

Building a function ν satisfying (5.32) is almost what the *partial expansion* algorithm presented in Section 5.3.5 has been crafted for. Instead of generating code, one can redesign this algorithm to compute an *equivalence relation* χ over writes: the *coloring relation*. Its only requirement is to assign different colors to interfering writes,

$$\forall v, w \in \mathbf{W} : v \infty w \implies \neg(v \,\chi\, w), \tag{5.33}$$

but we are also interested in minimizing the number of colors. When $v \,\chi\, w$, it says that it is *correct* to have $f_e^{\text{EXP}}(v) = f_e^{\text{EXP}}(w)$. The new graph coloring algorithm is presented in Section 5.4.6.

By construction of relation χ, a function ν defined by

$$\forall v, w \in \mathbf{W}, v \,\kappa\, w : \quad v \,\chi\, w \iff \nu(v) = \nu(w)$$

satisfies expansion correctness (5.32), but annoyingly, nothing ensures that expansion constraint (5.25) is still satisfied: for all $v, w \in \mathbf{W}$ such as $v \,\kappa\, w$, we have $v \infty w \Rightarrow \nu(v) \neq \nu(w)$ but not necessarily $v \equiv w \Rightarrow \nu(v) \neq \nu(w)$. Indeed, χ defines a minimal expansion allowing the *parallel execution order* to preserve the original semantics, but it does not enforce that this expansion satisfies the *constraint*.

The first problem is to check the compatibility of $\equiv$ and ∞. This is ensured by the following result.[18]

Proposition 5.3 For all writes v and w, it is not possible that $v \equiv w$ and $v \infty w$ at the same time.[19]

Proof: Suppose $v \,\kappa\, w$, $v \equiv w$, $v \infty w$ and $v <_{\text{SEQ}} w$. The third line of (5.29) shows that $v \,\delta^{\text{EXP}}\, w$, hence $v <_{\text{PAR}} w$ from (5.26). This proves that the $v \not<_{\text{PAR}} w$ conjunct in second line of (5.30) does not hold. Now, since $v \infty w$, one may consider a read instance $u \in \mathbf{R}$ such that the first line of (5.30) is satisfied: $v \,\sigma\, u \wedge w \not<_{\text{PAR}} v \wedge u \not<_{\text{PAR}} w \wedge u <_{\text{SEQ}} w$. Exchanging the role of u and v in the second line of (5.29) shows that $u \,\delta^{\text{EXP}}\, w$, hence $u <_{\text{PAR}} w$ from (5.26); this is contradictory with $u \not<_{\text{PAR}} w$.

Likewise, the case $w <_{\text{SEQ}} v$ yields a contradiction with $u \not<_{\text{PAR}} v$ in the second line of (5.30). This terminates the proof. ■

We now have to define ν from a new equivalence relation, considering both $\equiv$ and χ. Figure 5.33 shows that $\equiv \cup \chi$ is not sufficient: consider three writes u, v and w such that $f_e(u) = f_e(v) = f_e(w)$, $u \equiv v$ and $v \,\chi\, w$. (5.28) enforces $f_e^{\text{EXP}}(u) = f_e^{\text{EXP}}(v)$ since $u \equiv v$. Moreover, to spare memory, we should use coloring relation χ and set $f_e^{\text{EXP}}(v) = f_e^{\text{EXP}}(w)$. Then, no expansion is done and parallel order $<_{\text{PAR}}$ may be violated.

[18] The proof of this strong result is rather technical but helps understanding the role of each conjunct in equations (5.29), (5.26) and (5.30).

[19] A non-optimal definition of relation ∞ would not yield such a compatibility result.

w	`if (···) x = ···`
r_w	`··· = ··· x ···`
u	`x = ···`
v	`if (···) x = ···`
r_{uv}	`··· = ··· x ···`

Original program, $\sigma(r_w) = \{w\}$ and $\sigma(r_{uv}) = \{u, v\}$.

u	`x = ···`
w	`if (···) x = ···`
r_w	`··· = ··· x ···`
v	`if (···) x = ···`
r_{uv}	`··· = ··· x ···`

Wrong expansion when moving u to the top: r_w may read the value produced by u.

u	`y = ···`
w	`if (···) x = ···`
r_w	`··· = ··· x ···`
v	`if (···) y = ···`
r_{uv}	`··· = ··· y ···`

Correct when assigning `y` in u and v and moving u to the top.

Figure 5.33. Strange interplay of constraint and coloring relations

To avoid this pitfall, coloring relation must be used with care: one may safely set $f_e^{\text{EXP}}(u) = f_e^{\text{EXP}}(v)$ when for all $u' \equiv u$, $v' \equiv v$: $u' \,\chi\, v'$ (i.e. u' and v' share the same color). We thus build a new relation over writes, built from χ and $\equiv$. It is called the *constraint coloring relation*, and is defined by

$$\forall v, w \in \mathbf{W} : \quad v \,\chi_{\equiv}\, w \overset{\text{def}}{\iff} v \equiv w \vee \big(\forall v', w' : v' \equiv v \wedge w' \equiv w \implies v' \,\chi\, w'\big). \tag{5.34}$$

We can rewrite this definition using algebraic operations:

$$\chi_{\equiv} \quad = \quad \equiv \cup \big(\chi \setminus \equiv \circ ((\mathbf{W} \times \mathbf{W}) \setminus \chi) \circ \equiv\big). \tag{5.35}$$

The good thing is that relation $\chi_{\equiv}$ is an equivalence: the proof is simple since both $\equiv$ and χ are equivalence relations. Moreover, choosing $\nu(v) = \nu(w)$ when $v \,\chi_{\equiv}\, w$ and $\nu(v) \neq \nu(w)$ when its not the case ensures that $f_e^{\text{EXP}} = (f_e, \nu)$ satisfies both the expansion constraint *and* the expansion correctness criterion.

The following result solves the constraint storage mapping optimization problem:[20]

Theorem 5.4 Storage mapping f_e^{EXP} of the form (f_e, ν) such that

$$\forall v, w \in \mathbf{W}, v \,\kappa^*\, w : \quad v \,\chi_{\equiv}\, w \iff \nu(v) = \nu(w) \tag{5.36}$$

is the *minimal* storage mapping—i.e. accesses the fewer memory locations—which is constrained by $\equiv$ and allows the parallel execution order $<_{\text{PAR}}$ to preserve the program semantics, $\equiv$ and χ being the only information about permitting two instances to assign the same memory location.

Proof: From Proposition 5.3, we already know that $\equiv$ and ∞ have an empty intersection. Together with the inclusion of $\chi \setminus \equiv \circ((\mathbf{W} \times \mathbf{W}) \setminus \chi) \circ \equiv$ into χ, this proves the correctness of $f_e^{\text{EXP}} = (f_e, \nu)$. The constraint is also enforced by f_e^{EXP} since $\equiv \subset \chi_{\equiv}$. To prove the optimality result, one first observe that ν defines an equivalence relation of write instances, and second that $\chi_{\equiv}$ is the largest equivalence relation included in $\equiv \cup \chi$. ■

Theorem 5.4 gives us an automatic method to minimize memory usage, according to a parallel execution order and a predefined expansion constraint. Figure 5.34 gives an

[20] See Section 2.4.4 for a general remark about optimality.

intuitive presentation of this complex result: starting from the "*maximal constrained expansion*", we compute a parallel execution order, from which we compute a "*minimal correct expansion*", before combining the result with the constraint to get a "*minimal correct constrained expansion*".

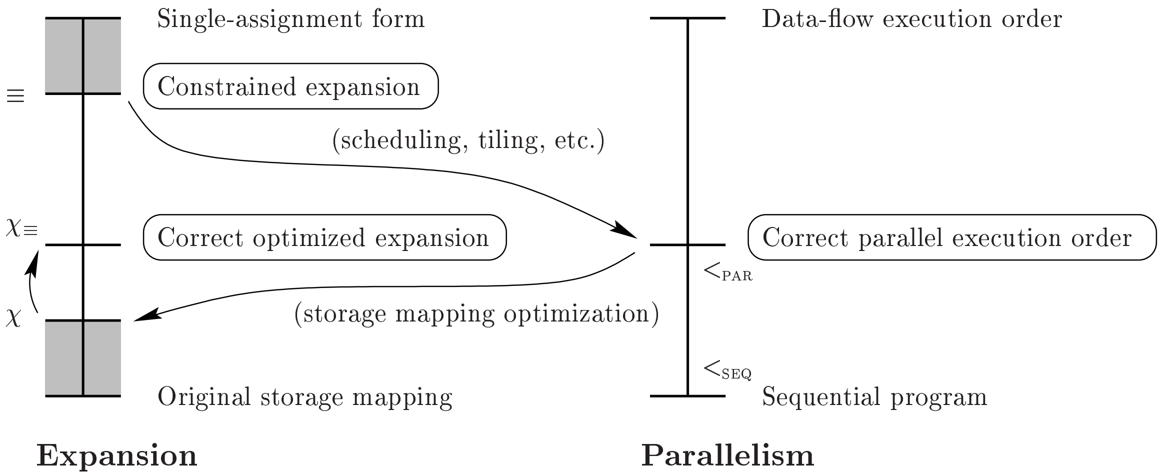


Figure 5.34. How we achieve constrained storage mapping optimization

5.4.4 Algorithm

As a summary of the optimization problem, one may group the formal constraints exposed in Section 5.4.3 into the system:

$$\begin{cases} \text{Constraints on } f_e^{\text{EXP}} = (f_e, \nu): \\ \forall v, w \in \mathbf{W}: \quad v \,\kappa\, w \wedge v \equiv w \implies \nu(v) = \nu(w) \\ \forall v, w \in \mathbf{W}: \quad v \,\kappa\, w \wedge v \not\bowtie w \implies \nu(v) \neq \nu(w) \\ \\ \text{Constraints on } <_{\text{PAR}}: \\ \forall (\imath_1, r_1), (\imath_2, r_2) \in \mathbf{A}: \quad (\imath_1, r_1)\, \delta^{\text{EXP}}\, (\imath_2, r_2) \implies \imath_1 <_{\text{PAR}} \imath_2 \end{cases}$$

Figure 5.35 shows the acyclic graph allowing computation of relations and mappings involved in this system.

The algorithm to solve this system is based on Theorem 5.4. It computes relation $\chi_\equiv$ with an extension of the partial expansion algorithm presented in Section 5.3.4, rewritten to handle constrained expansion. Before applying CONSTRAINED-STORAGE-MAPPING-OPTIMIZATION, we suppose that parallel execution order $<_{\text{PAR}}$ has been computed from $<_{\text{SEQ}}$, κ, σ, and $\equiv$, by first computing dependence relation δ^{EXP} then applying some appropriate parallel order computation algorithm (scheduling, tiling, etc.). Then, this parallel execution order is used to compute the expansion correctness criterion $\not\bowtie$. Algorithm CONSTRAINED-STORAGE-MAPPING-OPTIMIZATION reuses COMPUTE-REPRESENTATIVES and ENUMERATE-REPRESENTATIVES from Section 5.2.5.

As in the last paragraph of Section 5.2.4, one may consider splitting expanded arrays into renamed data structures to improve performance and reduce memory usage.

Eventually, when the compiler or the user knows that the parallel execution order $<_{\text{PAR}}$ has been produced by a *tiling* technique, we have already pointed in Section 5.3.6 that

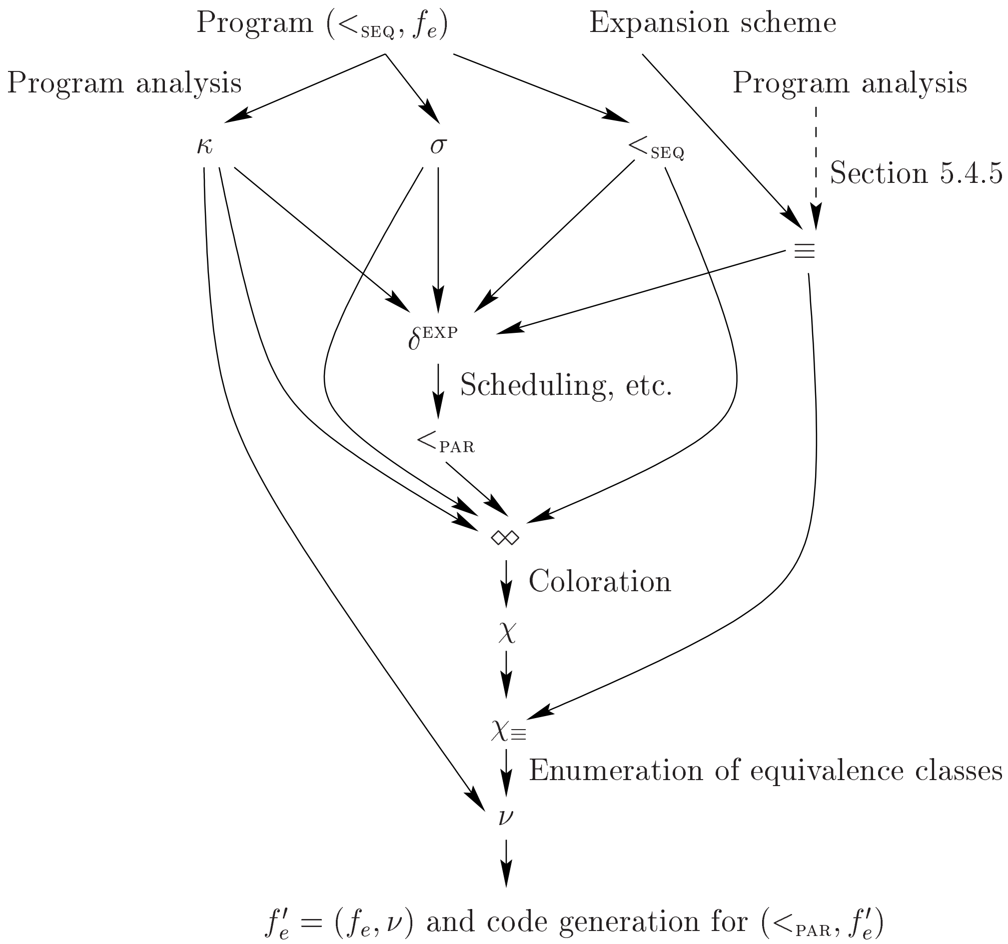


Figure 5.35. Solving the constrained storage mapping optimization problem

the *cyclic* graph coloring algorithm is not efficient enough. If the tile shape is known, one may build a vector of each dimension size, and use it as a "suggestion" for a *block-cyclic* storage mapping. This vector of block sizes is used when replacing the call to Cyclic-Coloring with a call to Near-Block-Cyclic-Coloring in Constrained-Storage-Mapping-Optimization.

5.4.5 Building Expansion Constraints

Our goal here is *not* to choose the right constraint suitable to expand a given program, but this does not mean leaving the user compute relation $\equiv$!

As shown in Section 5.4.2, enforcing the expansion to be *static* corresponds to setting $\equiv = \mathfrak{R}^*$. The constraint is thus built from instancewise reaching definition results (see Section 5.2).

Another example is *privatization*, seen as expansion along *some* surrounding loops, *without renaming*. Consider two accesses u and v writing into the same memory location. After privatization, u and v assign the same location if their iteration vectors coincide on the components associated with privatized loops:

$$u \equiv v \iff \textsc{Iter}(u)[\text{privatized loops}] = \textsc{Iter}(v)[\text{privatized loops}],$$

where $\textsc{Iter}(u)$[privatized loops] holds counters of privatized loops for instance u.

Constrained-Storage-Mapping-Optimization $(program, \kappa, \sigma, \equiv, <_{\text{PAR}})$
  program: an intermediate representation of the program
  κ: the conflict relation
  σ: the reaching definition relation, seen as a function
  $\equiv$: the expansion constraint
  $<_{\text{PAR}}$: the parallel execution order
  returns an intermediate representation of the expanded program

$\bowtie \leftarrow \big((\sigma(\mathbf{R}) \times \mathbf{W}) \cap \not>_{\text{PAR}} \cap >_{\text{SEQ}}\big) \cup \big(\not>_{\text{PAR}} \cap (\sigma \circ (\not<_{\text{PAR}} \cap <_{\text{SEQ}}))\big)$
$\quad\cup\quad \big((\sigma(\mathbf{R}) \times \mathbf{W}) \cap \not<_{\text{PAR}} \cap <_{\text{SEQ}}\big) \cup \big(\not<_{\text{PAR}} \cap (\sigma \circ (\not<_{\text{PAR}} \cap <_{\text{SEQ}}))\big)$
$\chi \leftarrow$ Cyclic-Coloring $(\kappa \cap \bowtie)$
$\chi_{\equiv} \leftarrow \equiv \cup (\chi \setminus \equiv \circ ((\mathbf{W} \times \mathbf{W}) \setminus \chi) \circ \equiv)$
$\rho \leftarrow$ Compute-Representatives $(\kappa^* \cap \chi_{\equiv})$
$\nu \leftarrow$ Enumerate-Representatives (κ^*, ρ)
for each array `A` $\in$ *program*
do $\nu_{\mathtt{A}} \leftarrow$ component-wise maximum of $\nu(u)$ for all write accesses u to `A`
  declaration `A[`*shape*`]` $\leftarrow$ `A`$_{\mathtt{EXP}}$`[`*shape*`,` $\nu_{\mathtt{A}}$`]`
  for each statement S assigning `A` **in** *program*
  do left-hand side `A[`*subscript*`]` of $S \leftarrow$ `A`$_{\mathtt{EXP}}$`[`*subscript*`,` ν`(CurIns)]`
  for each reference *ref* to `A` **in** *program*
  do $\sigma_{/ref} \leftarrow \sigma \cap (\mathbf{I} \times ref)$
    quast $\leftarrow$ Make-Quast $(\nu \circ \sigma_{/ref})$
    map $\leftarrow$ CSMO-Convert-Quast $(quast, ref)$
    ref $\leftarrow$ *map* `(CurIns)`
return *program*

CSMO-Convert-Quast $(quast, ref)$
  quast: the quast representation of the reaching definition function
  ref: the original reference
  returns the implementation of *quast* as a value retrieval code for reference *ref*

switch
  case $quast = \{\perp\}$:
    return *ref*
  case $quast = \{\imath\}$:
    $A \leftarrow$ Array$(\imath)$
    $S \leftarrow$ Stmt$(\imath)$
    $x \leftarrow$ Iter$(\imath)$
    subscript $\leftarrow$ original array subscript in *ref*
    return `A`$_{\mathtt{EXP}}$`[`*subscript*`,` x`]`
  case $quast = \{\imath_1, \imath_2, \ldots\}$:
    return $\phi(\{\imath_1, \imath_2, \ldots\})$
  case $quast =$ ***if*** *predicate* ***then*** $quast_1$ ***else*** $quast_2$:
    return `if` predicate CSMO-Convert-Quast $(quast_1, ref)$
      `else` CSMO-Convert-Quast $(quast_2, ref)$

Building the constraint for *array SSA* is even simpler. Instances of the *same statement* assigning the same memory location must still do so in the expanded program (only variable renaming is performed):

$$u \equiv v \iff \text{Stmt}(u) = \text{Stmt}(v)$$

Now, remember we have defined an extension of reaching definitions, called *reaching definitions of memory locations*. This definition can be used to weaken the static expansion constraint: if the aim of constrained expansion is to reduce run-time overhead due to ϕ functions, then σ^{ML} seems more appropriate than σ to define the constraint. Indeed, if Loop-Nests-ML-SA is used to convert a program to SA form, we have seen that ϕ functions generated by the classical algorithm have disappeared, see the second method in Section 5.1.4. It would thus be interesting to replace

$$\text{Make-Quast}\,(\nu \circ \sigma_{/ref})$$

in line 14 of Constrained-Storage-Mapping-Optimization by

$$\text{Make-Quast}\,(\sigma^{\text{ML}}_{/ref}\,(u, f_e(u)))$$

and to consider the constraint defined by the transitive closure of relation $\mathfrak{W}$

$$\forall v, w \in \mathbf{W}: \quad v\,\mathfrak{W}\,w \iff \exists c \in f(u) : v, w \in \sigma^{\text{ML}}\,(u, c),$$

where f is some conservative approximation of f_e. Maximal expansion according to constraint $\mathfrak{W}^*$ is called *weakened static expansion*. Eventually, setting $\equiv = \mathfrak{W}^*$ combines weakened static expansion with storage mapping optimization.

These practical examples give the insight that building $\equiv$ from the formal definition of an expansion strategy is not difficult. New expansion strategies should be designed and expressed as constraints—statement-by-statement, user-defined, knowledge-based, and especially *architecture dependent* (number of processors, memory hierarchy, communication model) constraints.

5.4.6 Graph-Coloring Algorithm

Our graph coloring problem is almost the same as the one studied by Feautrier and Lefebvre in [LF98], and the core of their solution has been recalled in Section 5.3.5. However, the formulation is slightly different now: it is no longer mixed-up with code generation. An easy work-around would be to redesign the output of algorithm Storage-Mapping-Optimization, as proposed in [Coh99b]: let Stmt(u) (resp. Iter(u)) be the statement (resp. iteration vector) associated with access u, and let NewArray(S) be the name of the new array assigned by S (after partial expansion),

$$\begin{aligned}\forall v, w \in \mathbf{W}: \quad v\,\chi\,w \overset{\text{def}}{\iff} & \ \text{NewArray}(\text{Stmt}(v)) = \text{NewArray}(\text{Stmt}(w)) \\ & \wedge \quad \big(\text{Iter}(v) \mod \mathsf{E}_{\text{Stmt}(v)} = \text{Iter}(w) \mod \mathsf{E}_{\text{Stmt}(w)}\big).\end{aligned}$$

This solution is simple but not practical. We thus present a full algorithm suitable for graph defined by affine relations: Cyclic-Coloring is used on statement instances for our storage mapping optimization purposes. Since the algorithm is general purpose, we consider an interference relation between vectors (of the same dimension). Using this algorithm for statement instances requires a preliminary encoding of statement name inside the iteration vector, and a padding of short vectors with zeroes. We already use this technique when formatting instances to the Omega syntax: see Section 5.2.7 for a practical example.

Remember that Storage-Mapping-Optimization was based on two independent techniques: building of an expansion vector and partial renaming. This decomposition came from the bounded statement number which allowed efficient greedy coloring techniques, and the infinity of iteration vectors which required a specific cyclic coloring. Cyclic-Coloring proceeds in a very similar way, and the reasoning of Section 5.3.5 and [LF98, Lef98] is still applicable to prove its correctness. However, the decomposition into two coloring stages is extended here in considering all *finite dimensions* of the vectors considered: if the vectors related with an interference relation have some dimensions whose components may only take a finite number of values, it is interesting to apply a classical coloring algorithm to these *finite dimensions*. We then build an equivalence relation of vectors that share the same finite dimensions: it is called $finite$ in the Cyclic-Coloring algorithm (the number of equivalence classes is obviously finite). When vectors encode statement instances, it is clear that the last dimension is finite, but some examples may present more finite dimensions, for example with small loops whose bounds are known at compile time. This extension may thus bring more efficient storage mappings that the Storage-Mapping-Optimization algorithm in Section 5.3.4.

Cyclic-Coloring (∞)
∞: the affine interference graph
returns a valid and economical cyclic coloration
$N \leftarrow$ dimension of vectors related with $interfere$
$finite \leftarrow$ equivalence relation of vectors sharing the same finite components
for each class set **in** $finite$
do for $p = 1$ **to** N
  do $working \leftarrow \{(v, w) : v \in set \wedge w \in set$
    $\wedge\ v[1..p] = w[1..p] \wedge v[1..p+1] < w[1..p+1]$
    $\wedge\ \langle S, v\rangle \infty \langle S, w\rangle\}$
    $maxv \leftarrow \{(v, \max_{<_{\text{LEX}}}\{w : (v, w) \in working\})\}$
    $vector[p+1] \leftarrow \max_{<_{\text{LEX}}}\{w - v[p+1] + 1 : (v, w) \in maxv\}$
  $cyclic_{set} \leftarrow v \mod vector$
$interfere \leftarrow \varnothing$
for each set, set' **in** $finite$
do if $(\exists v \in set, v' \in set' : v \infty v')$
  then $interfere \leftarrow interfere \cup \{(set, set')\}$
$coloring \leftarrow$ Greedy-Coloring ($interfere$)
$col \leftarrow \varnothing$
for each set **in** $finite$
do $col \leftarrow col \cup (cyclic_{set}, coloring(set))$
return col

The Near-Block-Cyclic-Coloring algorithm is an optimization ofCyclic-Coloring: it includes an improvement of the technique to efficiently handle graphs associated with tiled programs, as hinted in Section 5.3.6. In this particular case, we consider—as in most tiling techniques—a *perfectly nested* loop nest. Notice the "/" symbol is used for symbolic integer division. The intuitive idea is that a block-cyclic coloring is prefered to the cyclic one of the classical algorithm.

The Near-Block-Cyclic-Coloring algorithm should be seen as a first attempt to compute optimized storage mappings for tiled programs. As shown in Section 5.3.6, the block-cyclic coloring problem is still open for affine interference relations.

NEAR-BLOCK-CYCLIC-COLORING $(\diamond\!\diamond, shape)$
$\diamond\!\diamond$: a symbolic interference graph
shape: a vector of block sizes suggested by a tiling algorithm
returns a valid and economical block-cyclic coloration
$N \leftarrow$ number of nested loops
$quotient \leftarrow \{(x, x) : x \in \mathbb{Z}^N\}$
for $p = 1$ **to** N
do $quotient' \leftarrow quotient \circ$
$\{(x, y) : y[1] = x[1], \ldots, y[p] = x[p]/shape_p, \ldots, y[N] = x[N]\}$
  if $(\nexists z : z \; quotient' \circ \diamond\!\diamond \circ quotient'^{-1} \; z)$
    then $quotient \leftarrow quotient'$
$col \leftarrow$ CYCLIC-COLORING $(quotient \circ \diamond\!\diamond \circ quotient^{-1})$
return $col \circ quotient$

5.4.7 Dynamic Restoration of the Data-Flow

As in Section 5.3.8, Φ-arrays should be chosen in one-to-one mapping with the expanded data structures, and arguments of ϕ functions—i.e. sets of possible reaching definitions—should be updated according to the new storage mapping. The technique is essentially the same: function f_e^{EXP} is used to access Φ-arrays, then relation $\not\kappa$ and function ν are used to recompute the sets of possible reaching definitions:[21] a $\phi(set)$ reference should be replaced by

$$\phi\big(\{v \in set : \nexists w \in set : v <_{\text{SEQ}} w \wedge \neg(v \not\kappa w) \wedge \nu(v) = \nu(w)\}\big).$$

Another optimization is based on the shape of Φ-arrays: since $f_e^{\text{EXP}} = (f_e, \nu)$, the memory location written by a possible reaching definition can be deduced from the array subscript, and the boolean type is now preferred for Φ-arrays elements. This very simple optimization reduces both memory usage and run-time overhead. Algorithm CSMO-IMPLEMENT-PHI summarizes these optimizations.[22]

As hinted in Section 5.1.4, the goal is now to avoid redundancy in the run-time restoration of the data flow. Our technique extends ideas from the algorithms to efficiently place ϕ functions in the SSA framework [CFR+91, KS98]. However, code generation for the online computation of ϕ functions is rather different.

As in the SSA framework, ϕ functions should be placed at the *joins* of the control-flow graph [CFR+91]: there is a join at some *program point* when several control-flow paths merge together. Remember the control-flow graph is *not* the control automaton defined in Section 2.3.1, and a program point is an inter-statement location in the program text [ASU86]. Of course, textual order $<_{\text{TXT}}$ is extended to program points.

Joins are efficiently computed with the *dominance frontier* technique, see [CFR+91] for details. Indeed, the only "interesting" joins are those located on a path from a write w to a use whose set of possible reaching definitions is non empty and holds w. If POINTS is the set of program points, the set of "interesting" joins for an array (or scalar) `A` is

[21] We use $\neg(v \not\kappa w)$ to approximate the relation between writes that *must* assign the same memory location.

[22] For efficiency reasons, an expanded array $\mathtt{A}_{\text{EXP}}$ is partitioned into several sub-arrays, as proposed in Section 5.4.4. To correctly handle this partitioning, some simple—but rather technical—modifications should be made on the algorithm.

CSMO-Implement-Phi (*expanded*)
expanded: an intermediate representation of the expanded program
returns an intermediate representation with run-time restoration code

for each array $\mathtt{A}_{\mathtt{EXP}}[shape]$ **in** expanded
do if there are ϕ functions accessing $\mathtt{A}_{\mathtt{EXP}}$
  then declare an array $\Phi\mathtt{A}_{\mathtt{EXP}}[shape]$ initialized to `false`
    for each read reference ref_ϕ to $\mathtt{A}_{\mathtt{EXP}}$ whose expanded form is $\phi(set)$
    do $sub_\phi \leftarrow$ array subscript in ref_ϕ
      $short \leftarrow \{v \in set : \nexists w \in set : v <_{\text{SEQ}} w \wedge \neg(v \not\prec w) \wedge \nu(v) = \nu(w)\}$
      for each statement s involved in set
      do $ref_s \leftarrow$ write reference in s
        $sub_s \leftarrow$ array subscript in ref_s
        if not already done for s
          then following s insert
            $\Phi\mathtt{A}_{\mathtt{EXP}}[sub_s,\ \nu(\textsc{CurIns}, ref_s)]$ `= true`
      $\phi(set) \leftarrow \mathtt{A}_{\mathtt{EXP}}[\max_{<_{\text{SEQ}}} \{\imath \in short : \Phi\mathtt{A}_{\mathtt{EXP}}[sub_\phi,\ \nu(\imath, ref_\phi)] = \mathtt{true}\}]$
return *expanded*

denoted by $\textsc{Joins}_\mathtt{A}$, and is formally defined by

$$\forall p \in \textsc{Points} : \quad p \in \textsc{Joins}_\mathtt{A} \iff \exists v, u \in \mathbf{I} : \\ v\,\sigma\,u \wedge \textsc{Stmt}(v) <_{\text{TXT}} p <_{\text{TXT}} \textsc{Stmt}(u) \wedge \textsc{Array}(\textsc{Stmt}(u)) = \mathtt{A}. \quad (5.37)$$

For each array (or scalar) `A` in the original program, the idea is to insert at each join j in $\textsc{Joins}_\mathtt{A}$ a pseudo-assignment statement

P_j `A[] = A[];`

which copies the entire structure into itself. Then, the reaching definition relation is extended to these pseudo-assignment statements and the constraint storage-mapping optimization process is performed on the modified program instead of the original one.[23] Application of Constrained-Storage-Mapping-Optimization and then CSMO-Implement-Phi (or an optimized version, see Section 5.1.4) generates an expanded program whose interesting property is the absence of any redundancy in ϕ functions. Indeed, the lexicographic maximum of two instances is never computed twice, since it is done as early as possible in the ϕ function of some pseudo-assignment statement.

However, the expanded program suffers from the overhead induced by array copying, which was not the case for a direct application of Constrained-Storage-Mapping-Optimization and CSMO-Implement-Phi. Knobe and Sarkar encounter a similar problem with SSA for arrays [KS98] and propose several optimizations (mostly based on copy propagation and invariant code motion), but they provide no general method to remove array copies–it is the very nature of SSA to generate temporary variables. Nevertheless, there is such a general method, based on the observation that each pseudo-assignment statement in the expanded program is followed by an Φ-array assignation, by construction of pseudo-assignment statements and the set $\textsc{Joins}_\mathtt{A}$. Consider the following code generation for a pseudo-assignment statement P:

```
for (···) {   // iterate through the whole array
```

[23] Extending the reaching definition relation does not require any other analysis: the sets of possible reaching definitions for pseudo-assignment accesses can be deduced from the original reaching definition relation.

```
P     A_EXP[subscript] = A_EXP[max (set)];
      ΦA_EXP[subscript] = true;
  }
```

Statement P does not compute anything, it only gathers possible values coming from different control paths. The idea is thus to *store instances instead of booleans* and to use `@`-arrays (see Section 5.1.4) instead of `Φ`-arrays. An array `@A_EXP` is initialized to $\perp$, and the array copy is bypassed in updating `@A_EXP[`*subscript*`]` with the maximum in right-hand side of P. The previous code fragment can thus safely be replaced by:

```
for (···) {   // iterate through the whole array
    @A_EXP[subscript] = max (set);
}
```

This technique to remove spurious array copies is implemented in CSMO-EFFICIENTLY-IMPLEMENT-PHI: the optimized generation code algorithm for ϕ functions. Remember that before calling this algorithm, CONSTRAINED-STORAGE-MAPPING-OPTIMIZATION should be applied on the original program *extended with pseudo-assignment statements.*[24]

CSMO-EFFICIENTLY-IMPLEMENT-PHI (*expanded*)
expanded: an intermediate representation of the expanded program
returns an intermediate representation with run-time restoration code

for each array $\texttt{A}_{\texttt{EXP}}$`[`*shape*`]` **in** expanded
do if there are ϕ functions accessing $\texttt{A}_{\text{EXP}}$
  then declare an array $\texttt{@A}_{\texttt{EXP}}$`[`*shape*`]` initialized to $\perp$
    for each read reference ref_ϕ to $\texttt{A}_{\text{EXP}}$ whose expanded form is $\phi(set)$
    do $sub_\phi \leftarrow$ array subscript in ref_ϕ
      $short \leftarrow \{v \in set : \nexists w \in set : v <_{\text{SEQ}} w \wedge \neg(v \not\prec w) \wedge \nu(v) = \nu(w)\}$
      for each statement s involved in set
      do $ref_s \leftarrow$ write reference in s
        $sub_s \leftarrow$ array subscript in ref_s
        if not already done for s
          then following s insert
            $\texttt{@A}_{\texttt{EXP}}\texttt{[}sub_s\texttt{,}\ \nu(\textsc{CurIns}, ref_s)\texttt{]} = \textsc{CurIns}$
      $\phi(set) \leftarrow \texttt{A}_{\texttt{EXP}}\texttt{[max}_{<_{\text{SEQ}}}\ \{\imath \in short : \texttt{@A}_{\texttt{EXP}}\texttt{[}sub_\phi\texttt{,}\ \nu(\imath, ref_\phi)\texttt{]}\}\texttt{]}$
    for each pseudo-assignment P to $\texttt{A}_{\text{EXP}}$ with reference $\phi(set)$
    do $genmax \leftarrow$ code-generation for the lexicographic genmax in set
      right-hand side of `Φ`-array assignment following $p \leftarrow genmax$
      remove statement P
return *expanded*

Eventually, computing the lexicographic maximum of a set—defined in Presburger arithmetics—is a well known problem with very efficient parallel implementations [RF94]. but it is easier and sometimes faster to perform an *online* computation. Let us denote by NEXTJOIN the next instance of the nearest pseudo-assignment statement following CURINS. Computation of the lexicographic maximum in $\phi(set)$ can be performed online in replacing each assignment of the form

$$\texttt{@A}_{\texttt{EXP}}\texttt{[}subscript\texttt{,}\ \nu(\textsc{CurIns})\texttt{]} = \textsc{CurIns}\texttt{;}$$

[24] Same remark regarding partitioning of expanded arrays as for CSMO-IMPLEMENT-PHI.

by

```
@A_EXP[subscript, ν(NextJoin)] = max (@A_EXP[subscript, ν(NextJoin)], CurIns);
```

(ν is defined for instances of NextJoin: it is a pseudo-assignment to A).

Applying CSMO-Efficiently-Implement-Phi and this transformation to the motivating example yields the same result as the SA form in Figure 5.28.

5.4.8 Parallelization after Constrained Expansion

This section aims to characterize *correct parallel execution orders* for a program after *maximal constrained expansion*. The benefit memory expansion is to remove spurious dependences due to memory reuse, but some memory-based dependences may remain after constrained expansion. We still denote by δ_e^{EXP} (resp. δ^{EXP}) the exact (resp. approximate) dependence relation of the *expanded* program with *sequential* execution order $(<_{\text{SEQ}}, f_e^{\text{EXP}})$. As announced in Section 5.4.3, we now give the full computation details for (5.29).

Dependences left by constrained expansion are, as usual, of three kinds.

1. Output dependences due to writes connected to each other by the constraint $\equiv$ (e.g. by $\mathfrak{R}^*$ in the case of MSE).
2. True dependences, from a definition to a read, where the definition either may reach the read or is related (by $\equiv$) to a definition that reaches the read.
3. Anti dependences from a read to a definition where the definition, even if it executes after the read, is related (by $\equiv$) to a definition that reaches the read.

Formally, we thus define δ_e^{EXP} for an execution $e \in \mathbf{E}$ as follows:

$$\begin{aligned}\forall e \in \mathbf{E}, \forall v, w \in \mathbf{A}_e : \quad v\, \delta_e^{\text{EXP}}\, w \iff \quad & v\, \sigma\, w \\ \vee \quad & f_e(v) = f_e(w) \wedge v \equiv w \wedge v <_{\text{SEQ}} w \\ \vee \quad & f_e(v) = f_e(\sigma_e(w)) \wedge v \equiv \sigma_e(w) \wedge v <_{\text{SEQ}} w \\ \vee \quad & f_e(w) = f_e(\sigma_e(v)) \wedge \sigma_e(v) \equiv w \wedge v <_{\text{SEQ}} w\end{aligned}$$

Then, the following definition of δ^{EXP} is the best pessimistic approximation of δ_e^{EXP}, supposing relation κ is the best available approximation of function f_e and σ is the best available approximation of function σ_e:

$$\begin{aligned}\forall v, w \in \mathbf{A} : \quad v\, \delta^{\text{EXP}}\, w \overset{\text{def}}{\iff} \quad & v\, \sigma\, w & (5.38)\\ \vee \quad & v\, \kappa\, w \wedge v \equiv w \wedge v <_{\text{SEQ}} w & (5.39)\\ \vee \quad & \big(\exists u \in \mathbf{W} : u\, \sigma\, w \wedge v\, \kappa\, u \wedge v \equiv u\big) \wedge v <_{\text{SEQ}} w & (5.40)\\ \vee \quad & \big(\exists u \in \mathbf{W} : u\, \sigma\, v \wedge u\, \kappa\, w \wedge u \equiv w\big) \wedge v <_{\text{SEQ}} w & (5.41)\end{aligned}$$

Now, since κ and $\equiv$ are reflexive relations, we observe that (5.38) is already included in (5.40). We may simplify the definition of δ^{EXP}:

$$\begin{aligned}\forall v \in \mathbf{W}, w \in \mathbf{R} : \quad & v\, \delta^{\text{EXP}}\, w \Leftrightarrow \big(\exists u \in \mathbf{W} : \quad u\, \sigma\, w \wedge v\, \kappa\, u \wedge v \equiv u\big) \wedge v <_{\text{SEQ}} w \\ \forall v \in \mathbf{R}, w \in \mathbf{W} : \quad & v\, \delta^{\text{EXP}}\, w \Leftrightarrow \big(\exists u \in \mathbf{W} : \quad u\, \sigma\, v \wedge u\, \kappa\, w \wedge u \equiv w\big) \wedge v <_{\text{SEQ}} w \\ \forall v, w \in \mathbf{W} : \quad & v\, \delta^{\text{EXP}}\, w \Leftrightarrow v\, \kappa\, w \wedge v \equiv w \wedge v <_{\text{SEQ}} w \qquad (5.42)\end{aligned}$$

Eventually, we get an algebraic definition of the dependence relation after maximal constrained expansion:

$$\delta^{\text{EXP}} \quad = \quad (\kappa \cap \equiv) \quad \cup \quad (\kappa \cap \equiv) \circ \sigma \quad \cup \quad \sigma^{-1} \circ (\kappa \cap \equiv). \tag{5.43}$$

The first term describes output dependences, the second one describes flow dependences (including reaching definitions), and the third one describes anti-dependences.

Using this definition, Theorem 2.2 page 81 describes correct parallel execution order $<_{\text{PAR}}$ after maximal constrained expansion. Practical computation of $<_{\text{PAR}}$ is done with scheduling or tiling techniques, see Section 2.5.2.

As an example, we parallelize the convolution program in Figure 5.6 (page 169). The constraint is the one of the *maximal static expansion*. First, we define the sequential execution order $<_{\text{SEQ}}$ within Omega (with conventions defined in Section 5.2.7):

```
Lex := {[i,w,2]->[i',w',2] : 1<=i<=i'<=N && 1<=w,w' && (i<i' || w<w')}
       union {[i,0,1]->[i',w',2] : 1<=i<=i'<=N && 1<=w'}
       union {[i,w,2]->[i',0,1] : 1<=i,i'<=N && 1<=w && i<i'}
       union {[i,0,1]->[i',0,1] : 1<=i<i'<=N}
       union {[i,0,3]->[i',0,3] : 1<=i<i'<=N}
       union {[i,0,1]->[i',0,3] : 1<=i<=i'<=N}
       union {[i,0,3]->[i',0,1] : 1<=i<i'<=N}
       union {[i,w,2]->[i',0,3] : 1<=i<=i'<=N && 1<=w}
       union {[i,0,3]->[i',w',2] : 1<=i<i'<=N && 1<=w'};
```

Second, recall from Section 5.2.7 that all writes are in relation for κ (since the data structure is a scalar variable), and that relation $\mathfrak{R}^*$ is defined by (5.12). We compute δ^{EXP} from (5.43):

```
D := (R union R(S) union S'(R)) intersection Lex;
D;

{[i,w,2] -> [i,w',2] : 1 <= i <= N && 1 <= w < w'} union
{[i,0,1] -> [i,w',2] : 1 <= i <= N && 1 <= w'} union
{[i,0,1] -> [i,0,3] : 1 <= i <= N} union
{[i,w,2] -> [i,0,3] : 1 <= i <= N && 1 <= w}
```

After MSE, it only remains dependences between instances sharing the *same value* of i. It makes the outer loop parallel (it was not the case without expansion of scalar `x`). The parallel program in maximal static expansion is given in Figure 5.14.b.

5.4.9 Back to the Motivating Example

Using the Omega Calculator text-based interface, we describe a step-by-step execution of the expansion algorithm. We have to code instances as integer-valued vectors. An instance $\langle s, i \rangle$ is denoted by vector `[i,..,s]`, where `[..]` possibly pads the vector with zeroes. We number T, S, R with 1, 2, 3 in this order, so $\langle T, i, j \rangle$, $\langle S, i, j, k \rangle$ and $\langle R, i \rangle$ are written `[i,j,0,1]`, `[i,j,k,2]` and `[i,0,0,3]`, respectively.

The result of instancewise reaching definition analysis is written in Omega's syntax:

```
S := {[i,0,0,3]->[i,j,k,2] : 1<=i,j<=M && 1<=k<=N}
     union {[i,j,1,2]->[i,j,0,1] : 1<=i,j<=M}
     union {[i,j,k,2]->[i,j,k-1,2] : 1<=i,j<=M && 2<=k<=N};
```

The conflict and no-conflict relations are trivial here, since the only data structure is a scalar variable: κ is the full relation and $\not\kappa$ is the empty one.

```
Con := {[i,j,k,s]->[i',j',k',s'] : 1<=i,i',j,j'<=M && 1<=k,k'<=N
        && ((s=1 && k=0) || s=2 || (s=3 && j=k=0))
        && ((s'=1 && k'=0) || s'=2 || (s'=3 && j'=k'=0))};
NCon := {[i,j,k,s]->[i',j',k',s'] : 1=2}; # 1=2 means FALSE!
```

As in Section 5.4.1, we choose static expansion as constraint. Relation $\equiv$ is thus defined as $\mathfrak{R}^*$ in Section 5.2.2:

```
S' := inverse S;
R := S(S');
```

No transitive closure computation is necessary since `R` is already transitive. Computing dependences is done according to (5.43) and relation `Con` is removed since it always holds:

```
D := R union R(S) union S'(R);
```

In this case, a simple solution to computing a parallel execution order is the transitive closure computation:

```
Par := D+;
```

We can now compute relation $\bowtie$ in left-hand side of the expansion correctness criterion, call it `Int`.

```
# The "full" relation
Full := {[i,j,k,s]->[i',j',k',s'] : 1<=i,i',j,j'<=M && 1<=k,k'<=N
         && ((s=1 && k=0) || s=2 || (s=3 && j=k=0))
         && ((s'=1 && k'=0) || s'=2 || (s'=3 && j'=k'=0))};

# The sequential execution order
Lex := {[i,j,0,1]->[i',j',0,1] : 1<=i<i'<=M && 1<=j,j'<=M}
       union {[i,j,0,1]->[i',j',k',2] : 1<=i<=i'<=M && 1<=j,j'<=M
              && 1<=k'<=N}
       union {[i,j,k,2]->[i',j',0,1] : 1<=i<i'<=M && 1<=j,j'<=M
              && 1<=k<=N}
       union {[i,j,k,2]->[i',j',k',2] : 1<=i<=i'<=M && 1<=j,j'<=M
              && 1<=k,k'<=N && (i<i' || (j<=j' && (j<j' || k<k')))}
       union {[i,j,0,1]->[i',0,0,3] : 1<=i<=i'<=M}
       union {[i,0,0,3]->[i',j',0,1] : 1<=i<i'<=M}
       union {[i,j,k,2]->[i',0,0,3] : 1<=i<=i'<=M && 1<=j<=M
              && 1<=k<=N}
       union {[i,0,0,3]->[i',j',k',2] : 1<=i<i'<=M && 1<=j'<=M
              && 1<=k'<=N}
       union {[i,0,0,3]->[i',0,0,3] : 1<=i<i'<=M};
ILex := inverse Lex;

NPar := Full - Par;
INPar := inverse NPar;
```

```
Int := (INPar intersection ILex)
       union (INPar intersection S(NPar intersection Lex));
Int := Int union (inverse Int);
```

The result is:

```
Int;

{[i,j,k,2] -> [i',j',k',2] : 1 <= j <= j' <= M
  && 1 <= k <= k' <= N && 1 <= i' < i <= M} union
{[i,j,k,2] -> [i',j',k',2] : 1 <= j < j' <= M
  && 1 <= k' < k <= N && 1 <= i' < i <= M} union
{[i,j,k,2] -> [i',j,k',2] : 1 <= k' < k <= N
  && 1 <= i' < i <= M && 1 <= j <= M} union
{[i,j,1,2] -> [i',j',1,2] : N = 1
  && 1 <= i' < i <= M && 1 <= j' < j <= M} union
{[i,j,k,2] -> [i',j',k',2] : 1 <= k <= k' <= N
  && 1 <= i' < i <= M && 1 <= j' < j <= M && 2 <= N} union
{[i,j,k,2] -> [i',j',k',2] : 1 <= k' < k <= N
  && 1 <= i' < i <= M && 1 <= j' < j <= M} union
{[i,j,k,2] -> [i',j,k',2] : k'-1, 1 <= k <= k'
  && 1 <= i < i' <= M && 1 <= j <= M && k < N} union
{[i,j,k,2] -> [i',j',k',2] : 1, k'-1 <= k <= k'
  && 1 <= i < i' <= M && 1 <= j < j' <= M && k < N} union
{[i,j,k,2] -> [i',j',k',2] : 1 <= i < i' <= M
  && 1 <= j < j' <= M && 1 <= k' < k < N} union
{[i,j,k,2] -> [i',j',k',2] : k'-1, 1 <= k <= k'
  && 1 <= i < i' <= M && 1 <= j' < j <= M && k < N} union
{[i,j,k,2] -> [i',j',k',2] : k-1, 1 <= k' <= k
  && 1 <= j < j' <= M && 1 <= i' < i <= M && k' < N} union
{[i,j,k,2] -> [i',j',k',2] : 1 <= k < k' < N
  && 1 <= i' < i <= M && 1 <= j' < j <= M} union
{[i,j,k,2] -> [i',j',k',2] : 1, k-1 <= k' <= k
  && 1 <= i' < i <= M && 1 <= j' < j <= M && k' < N} union
{[i,j,k,2] -> [i',j,k',2] : k-1, 1 <= k' <= k
  && 1 <= i' < i <= M && 1 <= j <= M && k' < N} union
{[i,j,k,2] -> [i',j',k',2] : 1 <= i < i' <= M
  && 1 <= j < j' <= M && 1 <= k < k' <= N} union
{[i,j,k,2] -> [i',j',k',2] : 1 <= i < i' <= M
  && 1 <= j < j' <= M && 1 <= k' <= k <= N && 2 <= N} union
{[i,j,1,2] -> [i',j',1,2] : N = 1 && 1 <= i < i' <= M
  && 1 <= j < j' <= M} union
{[i,j,k,2] -> [i',j,k',2] : 1 <= i < i' <= M
  && 1 <= k < k' <= N && 1 <= j <= M} union
{[i,j,k,2] -> [i',j',k',2] : 1 <= i < i' <= M
  && 1 <= k < k' <= N && 1 <= j' < j <= M} union
{[i,j,k,2] -> [i',j',k',2] : 1 <= i < i' <= M
  && 1 <= k' <= k <= N && 1 <= j' <= j <= M}
```

A quick verification shows that

```
Int intersection {[i,j,k,2]->[i,j,k',2]};
```

and

```
Int intersection {[i,j,0,1]->[i,j,k',2] : k' != 0};
```

are both empty. It means that $\langle T, i, j\rangle$ and $\langle S, i, j, k\rangle$ should share the same color for all $1 \leq k \leq N$ (R does not perform any write). However, the sets $W_0^T(v)$, $W_0^S(v)$ (for the `i` loop), $W_1^T(v)$, $W_1^S(v)$ (for the `j` loop) hold all accesses w executing after v. Then, different i or j enforces different color for $\langle T, i, j\rangle$ and $\langle S, i, j, k\rangle$. Application of the graph coloring algorithm thus yields the following definition of the coloring relation:

```
Col := {[i,j,0,1]->[i,j,k,2] : 1<=i,j<=M && 1<=k<=N}
       union {[i,j,k,2]->[i,j,k',2] : 1<=i,j<=M && 1<=k,k'<=N};
```

We now compute relation $\chi_{\equiv}$, thanks to (5.35):

```
Eco := R union (Col-R(Full-Col(R)));
```

We choose the representative of each equivalence class as the lexicographic minimum (relation κ always holds and has been removed):

```
Rho := Eco-Lex(Eco);
```

The result is:

```
Rho;

{[i,j,0,1] -> [i,j,0,1] : 1 <= i <= M && 1 <= j <= M} union
{[i,j,k,2] -> [i,j,0,1] : 1 <= i <= M && 1 <= j <= M && 1 <= k <= N}
```

The labeling scheme is obvious: the last two dimensions are stripped off from `Rho`. The resulting function ν is thus

$$\nu(\langle T, i, j\rangle) = (i, j) \qquad \text{and} \qquad \nu(\langle S, i, j, k\rangle) = (i, j).$$

Following the lines of CONSTRAINED-STORAGE-MAPPING-OPTIMIZATION, we have computed the same storage mapping as in Figure 5.31.

5.5 Parallelization of Recursive Programs

The last contribution of this work is about automatic parallelization of recursive programs. This topic has received little interest from the compilation community, but the situation is evolving thanks to new powerful multi-threaded environments for efficient execution of programs with control parallelism. When dealing with shared-memory architectures and software-emulated shared memory machines, tools like Cilk [MF98] provide a very suitable programming model for automatic or semi-automatic code generation [RR99].

Now, what programming model should we consider for parallel code generation? First, it it still an open problem to compute a schedule from a dependence relation described by a *transducer*. This is of course a strong argument against *data parallelism* as a model of choice for parallelization of recursive programs. Moreover, we have seen in Section 1.2

that the *control parallel* paradigm was well suited to express parallel execution in recursive programs. In fact, this assertion is true when most iterative computations are implemented with recursive calls, but not when parallelism is located within iterations of a loop. Since loops can be rewritten as recursive procedure calls, we will stick to control parallelism in the following.

Notice we have studied powerful expansion techniques for loop nests, but no practical algorithm for recursive structures has been proposed yet. We thus start with an investigation of specific aspects of expanding recursive programs and recursive data structures in Section 5.5.1. Then we present in Section 5.5.2 a simple algorithm for single-assignment form conversion of any code that fit into our program model: the algorithm can be seen as a practical realization of ABSTRACT-SA, the abstract algorithm for SA-form conversion (page 157). Then, a privatization technique for recursive programs is proposed in Section 5.5.4; and some practical examples are studied in Section 5.5.5. We also give some perspectives about extending maximal static expansion or storage mapping optimization to this larger class of programs.

The rest of this section addresses generation of parallel recursive programs. Section 5.5.6 starts with a short state of the art on parallelization techniques for recursive programs, then motivates the design of a new algorithm based on instancewise dataflow information. In Section 5.5.7, we present an improvement of the statementwise algorithm which allows *instancewise* parallelization of recursive programs: whether some statements execute in parallel or in sequence can be dependent on the instance of these statements—but it is still decided at *compile-time*. This technique is also completely novel in parallelization of recursive programs.

5.5.1 Problems Specific to Recursive Structures

Before proposing a general solution for SA-form conversion of recursive programs, we investigate several issues which make the problem more difficult for recursive control and data structures. Recall that elements in data structures in single-assignment form are in one-to-one mapping with control words. Thus, the preferred layout of an expanded data structure is a *tree*. Expanded data structures can sometimes be implemented with arrays: it is the case when only loops and simple recursive procedures are involved, and when loops and recursive calls are not "interleaved"—program `Queens` is such an example. But automatic recognition of such programs and effective design of a specific expansion technique are left for future work. We will thus always consider that *expanded data structures are trees whose edges are labeled by statement names.*

Management of Recursive Data-Structures

Compared to arrays, lists and trees seems much less easy to access and traverse: they are indeed not *random access data structures*. For example, the abstract algorithm ABSTRACT-SA (page 157) for SA-form conversion uses the notation $\mathtt{D}_{\mathtt{EXP}}$`[CurIns]` to refer the access of an element index by word $\imath$ in a data structure $\mathtt{D}_{\mathrm{EXP}}$. But when $\mathtt{D}_{\mathrm{EXP}}$ is a tree, what does it mean? How is it implemented? Is it efficient?

There is a quick answer to all these questions: the tree is traversed from its root using pointer dereferences along letters in CURINS, the result is of course very costly at runtime. A more clever analysis shows that CURINS is not a random word: it is the *current control word*. Its "evolution" during program execution is fully predictable: it can be seen as a different *local* variable in each program statement, a new letter being added at each block entry.

The other problem with recursive data structures is memory allocation. Because they cannot be allocated at compile-time in general, a very efficient memory management technique should be used to reduce the run-time overhead. We thus suppose that an automatic scheme for grouping `malloc`s or `new`s is implemented, possibly at the C-compiler or operating system level.

Eventually, both problems can be solved with a simple and efficient code generation algorithm. The idea is the following: suppose a recursive data structure indexed by CurIns must be generated by algorithm Abstract-SA; each time a block is entered, a new element of the data structure is allocated and the pointer to the last element—stored in a local variable—is dereferenced accordingly. This technique is implemented in Recursive-Programs-SA.

About Accuracy and Versatility

When trying to extend maximal static expansion and storage mapping optimization to recursive programs, two kind of problems immediately arise:

- transductions are not as versatile as affine relations, because some critical algebraic operations are not decidable and require conservative approximations;

- the results of dependence and reaching definition analyses are not always as precise as one would expect, because of the lack of expressiveness of rational and one-counter transductions.

These two points are of course limiting the applicability of "evolved" expansion techniques which intensively rely on algebraic operations on sets and relations.

In addition, a few critical operations useful to "evolved" expansion techniques are lacking, e.g., the class of left-synchronous relations is *not* closed under transitive closure. Conversely, the problem of enumerating equivalence classes seems rather easy because the lexicographical selection of a left-synchronous transduction is left-synchronous, see Section 3.4.3; a remaining problem would be to label the class representatives...

We are not aware of any result about coloring graphs of rational relations, but optimality should probably not hoped for, even for recognizable relations. Graph-coloring algorithms for rational relations would of course be useful for storage mapping optimization; but recall from Section 5.3.2 that many algebraic operations are involved in the expansion correctness criterion, and most of these operations are undecidable for rational relations.

The last point is that we have not found enough codes that both fit into our program model and require expansion techniques more "evolved" than single-assignment form or privatization. But this problem is more with the program model restrictions than with the applicability of static expansion and storage mapping optimization.

5.5.2 Algorithm

Algorithm Recursive-Programs-SA is a first attempt to give a counterpart of algorithm Loop-Nests-SA for recursive programs. It works together with Recursive-Programs-Implement-Phi to generate the code for ϕ functions. Expanded data structures all have the same type, `ControlType`, which is basically a tree type associated with the language L_{CTRL} of control words. It can be implemented using recursive types and sub-types, or simply with as many pointer fields as statement labels in Σ_{CTRL}. An additional field in `ControlType` stores the element value, it has the same type as original data structure elements, and it is called `value`.

Recursive-Programs-SA $(program, \sigma)$
    program: an intermediate representation of the program
    σ: a reaching definition relation, seen as a function
    returns an intermediate representation of the expanded program
    define a tree type called `ControlType` whose elements are indexed in L_{CTRL}
    for each data structure `D` **in** *program*
    do define a data structure $\mathtt{D}_{\mathtt{EXP}}$ of type `ControlType`
        define a global pointer variable $\mathtt{D}_{\mathtt{LOCAL}}$ `=` $\mathtt{\&D}_{\mathtt{EXP}}$
        for each procedure **in** *program*
        do insert a new argument $\mathtt{D}_{\mathtt{LOCAL}}$ in the first place
        for each call to a procedure p **in** *program*
        do insert $\mathtt{D}_{\mathtt{LOCAL}}$`->`$p$ `= new ControlType ()` before the call
            insert a new argument $\mathtt{D}_{\mathtt{LOCAL}}$`->`$p$ in the first place
        for each non-procedure block b **in** *program*
        do insert $\mathtt{D}_{\mathtt{LOCAL}}$`->`$b$ `= new ControlType ()` at the top of b
            define a local pointer variable $\mathtt{D}_{\mathtt{LOCAL}}$ `=` $\mathtt{D}_{\mathtt{LOCAL}}$`->`$b$
        for each statement s assigning `D` **in** *program*
        do left-hand side of $s \leftarrow$ $\mathtt{D}_{\mathtt{LOCAL}}$`->value`
        for each reference *ref* to `D` **in** *program*
        do $ref \leftarrow \phi(\sigma(\textsc{CurIns}, ref))$
    return *program*

A simple optimization to spare memory consists in removing all "useless" fields from `ControlType`, and every pointer update code in the associated program blocks and statements. By useless, we mean statement labels which are not useful to distinguish between different memory locations, i.e. which cannot be replaced by another label and yield another instance of an assignation statement to the considered data structure. Applied to program `Queens`, only three labels can be considered to define the fields of `ControlType`: Q, a, and b; all other labels are unnecessary to enforce the single-assignment property. This optimization should of course be applied on a data structure per data structure basis, to take benefit of the locality of data structure usage in programs.

One should notice that every read reference requires a ϕ function! This is clearly a big problem for efficient code generation, but detecting exact results and computing reaching definitions at run-time is not as easy as in the case of loop nests. In fact, a part of the algorithm is even "abstract": we have not discussed yet how the argument of the ϕ can be computed. To simplify the exposition, all these issues are addressed in the next section.

Of course, algorithm Recursive-Programs-Implement-Phi generates the code for Φ-structures $\Phi\mathtt{D}_{\text{EXP}}$ using the same techniques as the SA-form algorithm. These Φ-structures store addresses of memory locations, computed from the original write references in assignment statements. Each ϕ function requires a traversal of Φ-structures to compute the exact reaching definition at run-time: the maximum is computed recursively from the root of $\Phi\mathtt{D}_{\text{EXP}}$, and the appropriate element value in $\mathtt{D}_{\text{EXP}}$ is returned. This computation of the maximum can be done in parallel, as usual for reduction operations on trees.

```
Recursive-Programs-Implement-Phi (expanded)
    expanded: an intermediate representation of the expanded program
    returns an intermediate representation with run-time restoration code
    for each expanded data structure D_EXP in expanded
    do if there are φ functions accessing D_EXP
         then define a data structure ΦD_EXP of type ControlType
              define a global pointer variable ΦD_LOCAL = &ΦD_EXP
              for each procedure in program
              do insert a new argument ΦD_LOCAL in the first place
              for each call to a procedure p in program
              do insert ΦD_LOCAL->p = new ControlType () before the call
                 insert a new argument ΦD_LOCAL->p in the first place
              for each non-procedure block b in program
              do insert ΦD_LOCAL->b = new ControlType () at the top of b
                 define a local pointer variable ΦD_LOCAL = ΦD_LOCAL->b
                 insert ΦD_LOCAL->value = NULL
              for each read reference ref_φ to D_EXP whose expanded form is φ(set)
              do for each statement s involved in set
                 do ref_s ←write reference in s
                    if not already done for s
                       then following s insert ΦD_LOCAL->value = &ref_s
                 φ(set) ← { traverse D_EXP and ΦD_EXP in lexicographic order
                              using pointers D_LOCAL and ΦD_LOCAL respectively
                                if (ΦD_LOCAL->value == &ref_φ) maxloc = D_LOCAL;
                            maxloc->value; }
    return expanded
```

Two problems remain with ϕ function implementation.

- The tree traversal does not use the *set* argument of ϕ functions at all! Indeed, testing for membership in a rational language is not a constant-time problem, and it is even not linear in general for algebraic languages. This point is also related with run-time computation of sets of reaching definitions: it will be discussed in the next section.

- Several ϕ functions may induce many redundant computations, since the maximum must everytime be computed on the whole structure, not taking benefit of the previous results. This problem was solved for loop nests using a complex technique integrated with constrained storage mapping optimization (see Section 5.4.7), but no similar technique for recursive programs is available.

5.5.3 Generating Code for Read References

In the last section, all read accesses were implemented with ϕ functions. This solution ensures correctness of the expanded program, but it is obviously not the most efficient. If we know that the reaching definition relation σ is a partial function (i.e. the result is exact), we can hope for an efficient run-time computation of its value, as it is the case for loop nests (with the quast representation). Sadly, this is not as easy in general: some rational functions cannot be computed for a given input in linear time, and it is even worse for algebraic functions.

The class of *sequential functions* is interesting for this purpose, since it is decidable and allows efficient online computation, see Section 3.3.3. Because for every state and input letter, the output letter and next state are known unamiguously, we can compute sequential functions together with pointer updates for expanded data structures. This technique can be easily extended to a *sub-sequential function* $(\mathcal{T}, \rho)$, in adding the pointer updates associated with function ρ (from states to words, see Definition 3.10 page 100). The class of sub-sequential transductions is decidable in polynomial time among rational transductions and functions [BC99b]. This online computation technique is detailed in algorithm Recursive-Programs-Online-SA, for sub-sequential reaching defintion transductions. An extension to *online rational transduction* would also be possible, without significantly increasing the run-time computation cost, but decidability is not known for this class.

Dealing with algebraic functions is less enthusiastic, because deciding whether an algebraic relation is a function is rather unlikely, and it is the same for the class of online algebraic transductions. But supposing we are lucky enough to know that an algebraic transduction is online (hence a partial function), we can implement efficiently the run-time computation, with the same technique as before: the next state, output label, and stack operation is never ambiguous.

A similar technique can be used to optimize the tree traversal in the implementation of $\phi(set)$ by algorithm Recursive-Programs-Implement-Phi. Computing a *left-synchronous* approximation of the reaching definition transduction (even in the case of an algebraic transduction), one may use the closure under *prefix-selection* (see Section 3.4.3 and especially Proposition 3.11) to select the *topmost node* in $\mathtt{D}_{\mathtt{EXP}}\mathtt{[}set\mathtt{]}$ and $\mathtt{\Phi D}_{\mathtt{EXP}}\mathtt{[}set\mathtt{]}$. These topmost nodes can be used instead of the root of the trees to initiate the traversal. To be computed at run-time, however, the rational function implementing the prefix-selection of σ (approximate in general) must be sub-sequential. Another approach consists in computing an approximation of the union of all possible sets of reaching definitions involved in a given ϕ function. The result is rational (resp. algebraic) if the reaching definition transduction is rational (resp. algebraic), thanks to Nivat's Theorem 3.6 (resp. Evey's Theorem 3.24), and it can be used to restrict the tree traversal to a smaller domain. Both approaches can be combined to optimize the ϕ function implementation.

To conclude this discussion on run-time computation of reaching definitions, only the case of sub-sequential functions is very clear: it allows efficient online computation with algorithm Recursive-Programs-Online-SA. In all other cases—which includes all cases of algebraic transductions—we think that no real alternative to ϕ functions is available. In practice, Recursive-Programs-Online-SA should be applied to the largest subset of data structures and read references on which σ is sub-sequential, and Recursive-Programs-SA is used for the rest of the program. It is perhaps one of the greatest failures of our framework, since we computed an interesting information—reaching definitions—which we are unable to use in practice. This is also a discouraging argument for extending static expansion to recursive programs: what is the use of removing ϕ functions if the reaching definition information fails to give the value we are looking for at a lower cost? Finally, ϕ functions may be so expensive to compute that conversion to single-assignment form should be reconsidered, in favor of other expansion schemes. In this context, a very interesting alternative is proposed in the next section.

Eventually, looking at our motivating examples in Chapter 4, or thinking about most practical examples of recursive programs using trees and other pointer-based data structures, one common observation can be made: there is "not so many" memory reuse—if

Recursive-Programs-Online-SA $(program, \sigma)$
program: an intermediate representation of the program
σ: a sub-sequential reaching definition transduction
returns an intermediate representation of the expanded program
define a tree type called `ControlType` whose elements are indexed in L_{CTRL}
build $(\mathcal{T}, \rho)$ from σ where $\mathcal{T} = (Q, \{q_0\}, F, E)$ is sequential and $\rho : Q \to \Sigma^*_{\text{CTRL}}$
build a "next state" function $\alpha : Q \times \Sigma_{\text{CTRL}} \to Q$ from $\mathcal{T}$
build a "next output" function $\beta : Q \times \Sigma_{\text{CTRL}} \to \Sigma^*_{\text{CTRL}}$ from $\mathcal{T}$
for each data structure `D` **in** *program*
do declare a data structure $\mathtt{D}_{\mathtt{EXP}}$ of type `ControlType`
    define a global pointer variable $\mathtt{D}_{\mathtt{LOCAL}}$ `=` $\&\mathtt{D}_{\mathtt{EXP}}$
    define a global pointer variable $\mathtt{D}^{\sigma}_{\mathtt{LOCAL}}$ `=` $\&\mathtt{D}_{\mathtt{EXP}}$
    define a global "state" variable $\mathtt{D}^{Q}_{\mathtt{LOCAL}}$ `=` q_0
    for each procedure **in** *program*
    do insert a new argument $\mathtt{D}_{\mathtt{LOCAL}}$ in the first place
        insert a new argument $\mathtt{D}^{\sigma}_{\mathtt{LOCAL}}$ in the second place
        insert a new argument $\mathtt{D}^{Q}_{\mathtt{LOCAL}}$ in the third place
    for each call to a procedure p **in** *program*
    do insert $\mathtt{D}_{\mathtt{LOCAL}}$`->`$p$ `= new ControlType ()` before the call
        insert a new argument $\mathtt{D}_{\mathtt{LOCAL}}$`->`$p$ in the first place
        insert a new argument $\mathtt{D}^{\sigma}_{\mathtt{LOCAL}}$`->`$\beta(\mathtt{D}^{Q}_{\mathtt{LOCAL}}, p)$ in the second place
        insert a new argument $\alpha(\mathtt{D}^{Q}_{\mathtt{LOCAL}}, p)$ in the third place
    for each non-procedure block b **in** *program*
    do insert $\mathtt{D}_{\mathtt{LOCAL}}$`->`$b$ `= new ControlType ()` at the top of b
        define a local pointer variable $\mathtt{D}_{\mathtt{LOCAL}}$ `=` $\mathtt{D}_{\mathtt{LOCAL}}$`->`$b$
        define a local pointer variable $\mathtt{D}^{\sigma}_{\mathtt{LOCAL}}$ `=` $\mathtt{D}^{\sigma}_{\mathtt{LOCAL}}$`->`$\beta(\mathtt{D}^{Q}_{\mathtt{LOCAL}}, b)$
        define a local pointer variable $\mathtt{D}^{Q}_{\mathtt{LOCAL}}$ `=` $\alpha(\mathtt{D}^{Q}_{\mathtt{LOCAL}}, b)$
    for each statement s assigning `D` **in** *program*
    do left-hand side of $s \leftarrow \mathtt{D}_{\mathtt{LOCAL}}$`->value`
    for each reference ref to `D` **in** *program*
    do $ref \leftarrow \mathtt{D}^{\sigma}_{\mathtt{LOCAL}}$`->`$\rho(\mathtt{D}^{Q}_{\mathtt{LOCAL}})$`->value`
return *program*

not zero memory reuse—in these programs! This *late* but simple discovery is a strong argument against memory expansion techniques for recursive tree programs: they may simply be useless. In fact, many tree programs already have a high level of parallelism and do not need to be expanded. This is very disappointing that the best results of our single-assignment technique are likely to be very rarely useful in practice. In the case of recursive array programs, expansion is still a critical issue for parallelisation, like for the `Queens` program in Chapter 4.

5.5.4 Privatization of Recursive Programs

We have seen that SA-form conversion is not practical for all recursive programs. It was already the case for loop nests, but the problem is more obvious here. However, SA-form is probably not the most suitable method to extract parallelism from recursive programs. Because of the heavy use of procedures and functions, looking at expansion as a transformation of global data structures into local ones is much more profitable. This idea happens to be very similar to the principles of *array privatization* for loop nests, and we use the same word here. A general privatization technique can be defined for unrestricted recursive programs, but *copy-out* code is necessary to update the data structures of the calling procedure. In a parallel execution, this often requires additional synchronizations, and the overhead of such an expansion is likely to be very high. Further study is left for future work.

We will restrict ourselves to the case of reaching definition relations σ which satisfy the VPA property defined in Section 4.3.4 (for reaching definition analysis purposes): forall $u, v \in L_{\text{CTRL}}$, if $v \, \sigma \, u$ then v is an ancestor of u, i.e. $\exists w_1, w_2 \in L_{\text{CTRL}}, s \in \Sigma_{\text{CTRL}} : v = w_1 s \wedge u = w_1 w_2$ (and $v <_{\text{LEX}} u$, which is trivial since $v \, \sigma \, u$). This property is enforced in many important classes of recursive programs: all divide-and-conquer execution schemes, most dynamic-programming implementations, many sorting algorithms...

Now, the *privatization* technique for VPA programs is very simple: every global data structure (probably an array) to be expanded is made local to each procedure in the program, and the appropriate *copy-in* code of the whole structure is inserted at the beginning point of each procedure. Notice no *copy-out* is needed since it would involve reaching definitions from *non-ancestor* instances. A program privatized in that sense is generally less expanded than SA-form[25], and the parallelism extracted by privatization can be found at function calls only: instead of waiting for the function's return, one may run each function call in parallel and insert synchronizations only when the result of a function is needed.

This technique may appear somewhat expensive because of the data structure copying, but the same optimization that worked for loop nests can be applied here [TP93, MAL93, Li92]: privatization can be done on a *processor* basis instead, and copy-in is only performed when a procedure call is made *accross* processors. We implemented this optimization for program `Queens`, using Cilk's "fast" and "slow" implementations of parallel procedures, the "slow" one being called only when a processor "catches" new work [MF98]. Further discussion about parallelization of expanded programs is delayed to Section 5.5.6.

5.5.5 Expansion of Recursive Programs: Practical Examples

We applied single-assignment algorithm RECURSIVE-PROGRAMS-SA to program `Queens`. The result is shown in Figure 5.36. The `ControlType` structure has been optimized in keeping only fields which enforce the single-assignment form property. It is implemented with a C++ template-like syntax to handle both $\mathtt{D}_{\text{EXP}}$ and Φ-structure $\Phi\mathtt{D}_{\text{EXP}}$:

```
struct ControlType<T> {
  T value;
  ControlType<T> *Q;
  ControlType<T> *a;
  ControlType<T> *b;
};
```

Notice that the input automaton for the reaching definition transducer of procedure `Queens` is not deterministic. This ruins any hope to efficiently compute reaching definitions at run-time and to remove the ϕ function, despite the fact our analysis technique

[25] As a technical remark, this is not always true because we copy the whole data structures and not each element. In some tricky cases, privatization can require more memory than SA-form!

```
    int A[n];

    ControlType<int> A_EXP = new ControlType<int> ();
    ControlType<int> *A_LOCAL = &A_EXP;
    ControlType<int*> ΦA_EXP = new ControlType<int*> ();
    ControlType<int*> *ΦA_LOCAL = &ΦA_EXP;

P   void Queens (ControlType<int> *A_LOCAL, ControlType<int*> *ΦA_LOCAL,
                 int n, int k) {
I     if (k < n) {
A/a     for (int i=0; i<n; i++) {
          A_LOCAL->b = new ControlType<int> ();
          ΦA_LOCAL->b = new ControlType<int*> ();
          ControlType<int> *A_LOCAL = A_LOCAL->a;
          ControlType<int*> *ΦA_LOCAL = ΦA_LOCAL->a;
B/b       for (int j=0; j<k; j++) {
            A_LOCAL->b = new ControlType<int> ();
            ΦA_LOCAL->b = new ControlType<int*> ();
            ControlType<int> *A_LOCAL = A_LOCAL->b;
            ControlType<int*> *ΦA_LOCAL = ΦA_LOCAL->b;
r           ··· = ··· φ(σ(CurIns, A[j])) ···;
          }
J         if (···) {
s           A_LOCAL->value = ···;
            ΦA_LOCAL->value = &(A[k]);
            A_LOCAL->Q = new ControlType<int> ();
            ΦA_LOCAL->Q = new ControlType<int*> ();
Q           Queens (A_LOCAL->Q, ΦA_LOCAL->Q, n, k+1);
          }
        }
      }
    }

    int main () {
F     Queens (A_LOCAL, ΦA_LOCAL, n, 0);
    }
```

Figure 5.36. Single-assignment form conversion of program `Queens`

computed an exact result! The tree traversal associated with the ϕ function has not been implemented in Figure 5.36, but it does not require a full tranversal of $\mathtt{D}_{\text{EXP}}$: because only ancestors are possible reaching definitions (property VPA), the computation of the maximum can be made on the path from the root (i.e. $\mathtt{\&D}_{\text{EXP}}$) to the current element (i.e. $\mathtt{\&D}_{\text{LOCAL}}$). This is implemented most efficiently with pointers to the parent node in `ControlType`, stopping at the *first* ancestor in dependence (i.e. the deepest ancestor in dependence). An effective implementation of statement r is given in Figure 5.37. The `maxatloc != NULL` test is necessary in general, when $\perp$ can be a possible reaching definition, but it could indeed be removed in our case since execution of ancestors is guaranteed. The appropriate construction of the `parent` field in `ControlType` is assumed in the rest of the code.

```
r  {
     ControlType<int> *maxloc = D_LOCAL;
     ControlType<int*> *maxatloc = ΦD_LOCAL;
     while (maxatloc != NULL && maxatloc->value != &(A[j])) {
       maxloc = maxloc->parent;
       maxatloc = maxatloc->parent;
     }
     ··· = ··· maxloc->value ···;
   }
```

Figure 5.37. Implementation of the read reference in statement r

We also experimented the privatization technique since property VPA is satisfied for program `Queens`, see Figure 5.38. An additional optimization has been performed: only the k first elements of array `A` are copied, because the others are not used. This result can be obtained thanks to static analyses of variables [CH78]. Parallelization of the privatized form is studied in Section 5.5.6.

5.5.6 Statementwise Parallelization

We start with two motivating examples to show what we want to achieve, then discuss the results of classical static analyses on such examples, before we present our statementwise parallelization algorithm.

Motivating Example

Our first example is the `BST` program introduced in Section 2.3. Instancewise dependence analysis has been performed in Section 4.4 and the result is the rational transducer in Figure 4.9. Because the two recursive calls involve dereferences of pointer `p` along two distinct edges, and because the underlying data structure is a *tree*, we know that all accesses performed after the first call are independent from accesses performed after the second one. Both conditional statements I_1 and J_1 can thus be executed asynchronously (recall that an implicit synchronization is supposed at the return point of procedure `BST`, see Section 1.2). The parallel version is given by Figure 5.39.

Our second example maps two functions on a list, one on even elements and the other on the odd ones, see program `Map` in Figure 5.40. The result of our analysis for this program is that there are no dependences between instances of s and t. This allows parallel execution of s and t, and their respective function calls to `Even` and `Odd`.

Let us compare the effectiveness of related parallelization techniques with the expected results on these two motivating examples. Hendren et al. propose in [HHN94] a dependence test for recursive programs with pointer-based data structures. Their technique does not handle arrays (seen as pointer arithmetics in that case). But since it handles

```
    int A[n];

P   void Queens (int A[n], int n, int k) {
      int B[n];
      memcpy (B, A, k * sizeof (int));
I     if (k < n) {
A/a     for (int i=0; i<n; i++) {
B/b       for (int j=0; j<k; j++) {
r           ··· = ··· B[j] ···;
          }
J         if (···) {
s           B[k] = ···;
Q           Queens (B, n, k+1);
          }
        }
      }
    }

    int main () {
F     Queens (A, n, 0);
    }
```

Figure 5.38. Privatization of program `Queens`

a wide range of recursive data structures, including directed acyclic graphs and doubly-linked lists, it is more general than our technique in that domain. Because their pointer aliasing abstraction is based on *path expressions* which are pairs of regular expressions on the edge names, the `BST` program is actually parallelized with their technique. But the `Map` procedure is not, since their path expressions cannot capture the evenness of dereference numbers. The very precise alias analysis by Deutsch [Deu94] would allow parallelization of the two examples because Kleene stars are there replaced by named counters constrained with systems of affine equations. More usual flow-sensitive and context-sensitive alias analyses [LRZ93, EGH94, Ste96] would generally succeed for `BST` and fail for `Map`.

Algorithm

We now present an algorithm for statementwise parallelization of recursive programs, based on the results of our dependence analysis. Let $(\Sigma_{\text{CTRL}}, E)$ be the *dual control flow graph* [ASU86] of the program—i.e. the *dual graph* of the control flow graph—whose nodes are statements instead of program points, and whose edges are program points instead of statements. We define a *synchronization graph* $(\Sigma_{\text{CTRL}}, E')$ as a sub-graph of $(\Sigma_{\text{CTRL}}, E)$ such that every edge in E' is associated with a synchronization barrier. Supposing that all sequential compositions of statements are replaced by asynchronous executions, a synchronization graph must ensure that there are enough synchronization points to preserve the original program semantics. Thanks to Bernstein's conditions, this is ensured by the following condition: let $S, T \in \Sigma_{\text{CTRL}}$ be two program statements, $ST \in E$, and B be the

```
P   void BST (tree *p) {
I1    spawn if (p->l!=NULL) {
L       BST (p->l);
I2      if (p->value < p->l->value) {
a         t = p->value;
b         p->value = p->l->value;
c         p->l->value = t;
        }
      }
J1    spawn if (p->r!=NULL) {
R       BST (p->r);
J2      if (p->value > p->r->value) {
d         t = p->value;
e         p->value = p->r->value;
f         p->r->value = t;
        }
      }
    }

    int main () {
F     if (root!=NULL) BST (root);
    }
```

Figure 5.39. Parallelization of program `BST`

```
    void Map (List *p, List *q) {
s     p->value = Even (p->value);
t     q->value = Odd (q->value);
      if (···) {
        Map (p->next->next, q->next->next);
    }

    int main () {
      Map (list, list->next);
    }
```

Figure 5.40. Second motivating example: program `Map`

innermost block surrounding both S and T,

$$ST \in E' \iff \exists v, w \in L_{\text{CTRL}}, u, x', y' \in \Sigma^*_{\text{CTRL}}, x, y \in (\Sigma_{\text{CTRL}} \setminus \{B\})^* : \\ v = uBxSx' \wedge w = uByTy' \wedge v\,\delta\,w \vee w\,\delta\,v. \quad (5.44)$$

Indeed, executing $uBxS$ and $uByT$ in parallel induces parallel execution of all their descendants—coarse grain parallelization—and prefix u should be chosen as long as possible, hence the restriction of x and y to non-B labels. Algorithm Statementwise-Parallelization is based on this equation to generate a parallel program with the required synchronizations. It is interesting to notice that

$$v\,\delta\,w \vee w\,\delta\,v \iff v\,\kappa\,w \wedge (v \in \mathbf{W} \vee w \in \mathbf{W}),$$

which means that intersection with the lexicographic order is not necessary: conflict relation κ can be used instead of the dependence one to describe statements that may execute in parallel. Because $(\Sigma^*_{\text{CTRL}}B(\Sigma_{\text{CTRL}} \setminus \{B\})^* S \Sigma^*_{\text{CTRL}} \times \Sigma^*_{\text{CTRL}}B(\Sigma_{\text{CTRL}} \setminus \{B\})^* T \Sigma^*_{\text{CTRL}})$ in Statementwise-Parallelization is a recognizable language, its intersection with *depend* can be computed exactly. These two remarks show that computing the synchronization graph for a recursive program can be done without any approximation in most cases: the conflict relation is approximate only for multi-dimensional arrays. Notice that this algorithm does not perform any statement reordering inside a program block; this issue is left for future work.

Statementwise-Parallelization $(program, \kappa)$
    program: an intermediate representation of the program
    κ: the conflict relation to be satisfied by all parallel execution orders
    returns a parallel implementation of *program*

$depend \leftarrow \kappa \cap ((\mathbf{W} \times \mathbf{R}) \cup (\mathbf{R} \times \mathbf{W}) \cup (\mathbf{W} \times \mathbf{W}))$
$(\Sigma_{\text{CTRL}}, edges) \leftarrow$ dual control flow graph of *program*
for each ST **in** *edges*
do $B \leftarrow$ innermost block surrounding both S and T
    $synchro \leftarrow depend \cap (\Sigma^*_{\text{CTRL}}B(\Sigma_{\text{CTRL}} \setminus \{B\})^* S \Sigma^*_{\text{CTRL}}$
                $\times\, \Sigma^*_{\text{CTRL}}B(\Sigma_{\text{CTRL}} \setminus \{B\})^* T \Sigma^*_{\text{CTRL}})$
    if $synchro \neq \varnothing$
        then insert a `sync` statement at program point associated with ST
    insert a `spawn` keyword before every statement
return *program*

Of course, several `spawn` keywords may be useless or misplaced regarding the parallel programming environment: Cilk only allows asynchronous procedure calls, not asynchronous execution at the statement level, and several environments do not support nested parallelism. When a `spawn`ed statement is immediately followed by a `sync`, both keywords can be removed since such a construct is equivalent to sequential execution. In addition, powerful methods have been crafted to optimize the number of synchronization points and shrink the critical-path, see for example [Rin97]. Application of Statementwise-Parallelization on the two motivating examples yields the expected results.

Eventually, the parallelization technique proposed by Feautrier in [Fea98] would find a similar result on both motivating examples, since they are based on an *instancewise dependence test* (but automatic computation of storage mappings is not handled in [Fea98]).

Statementwise Parallelization via Memory Expansion

Our running example is now program `Queens`, already studied in the previous chapters. This program does not hold any parallel loop (the inner-loop looks parallel but memory dependences on the "$\cdots$" parts actually hampers parallelization). We will consider the reaching definition information computed in Section 4.5, i.e. the one-counter transducer in Figure 4.15, and the privatized `Queens` program proposed in Section 5.5.5, see Figure 5.38.

Recall that reaching definition relation σ of program `Queens` satisfied the VPA property: this guarantees that the reaching definition relation can be used as dependence information to decide whether a procedure call can be executed asynchronously or not. The result is that the recursive call can be made asynchronous, see Figure 5.41. Starting from the single-assignment form version of program `Queens`(see Figure 5.36), no more parallelism would have been extracted but the overhead due to ϕ function computation would make the parallel program unpractical.

```
    int A[n];

P   void Queens (int A[n], int n, int k) {
      int B[n];
      memcpy (B, A, k * sizeof (int));
I     if (k < n) {
A/a     for (int i=0; i<n; i++) {
B/b       for (int j=0; j<k; j++) {
r           ··· = ··· B[j] ···;
          }
J         if (···) {
s           B[k] = ···;
Q           spawn Queens (B, n, k+1);
          }
        }
      }
    }

    int main () {
F     Queens (A, n, 0);
    }
```

Figure 5.41. Parallelization of program `Queens` *via privatization*

The algorithm to achieve this result automatically is simple. First choose between single-assignment form and privatization; Second, apply algorithm STATEMENTWISE-PARALLELIZATION using the *reaching-definition* relation as *dependence relation for the expanded program*. However, if privatization is chosen, only asynchronous calls to privatized procedures are provably correct (they preserve the original program semantics), all other asynchronous and parallel constructs should be removed from the generated code; this is because some memory-based dependences between instances of non-procedure statements may remain.

Some experiments have been performed with the Cilk environment [MF98] on a 32 processor SGI Origin 2000. The results in Figure 5.42 corresponds to the execution time and to the speed-up of the parallel version compared to the *sequential non-privatized* one (without Cilk overhead and without array copying). The program was run with 13 queens only, to demonstrates both the efficiency of the Cilk run-time and the low overhead induced by the expansion of program `Queens`. Performance is very good up to 16 processors, then it degrades for 32 processors.

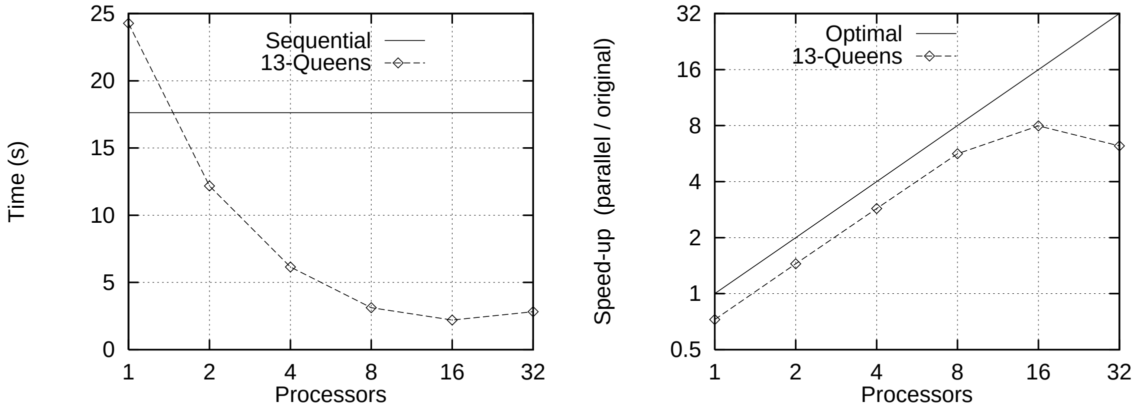


Figure 5.42. Parallel resolution of the n-Queens problem

Notice that the privatized `Queens` program can itself be the matter of a comparison with other parallelization techniques. It happens that analyses for pointer arithmetics (seen as a particular implementation of arrays) used by Rugina and Rinard in [RR99] are unable to parallelize the program. Indeed, the *ordering* analysis shows that $j < k$, which means that for a given iteration of the outer loop, the procedure call can be executed asynchronously with the next iterations. However, the *inter-procedural region expression analysis* computes a fix-point over recursive calls to procedure `Queens` which cannot capture the fact that only the k first elements of array `A` are useful: subsequent recursive calls are thus supposed to read the *whole* array `A`, which is not the case in practice.

5.5.7 Instancewise Parallelization

This last section investigates parallelization of recursive programs at the statement instance level. This common technique for loop nest parallelization is completely new for recursive programs. Notice we do *not* propose a run-time parallelization technique for recursive programs: we describe at *compile-time* the sets of run-time instances which can be executed asynchronously.

Motivating Example

We study the procedure `P` example in Figure 5.43.a. Pointer arguments `p` and `q` are identical in the first call: they are set to the root of a binary tree structure.

Because `p` and `q` may be aliased during the whole execution, any dependence *test*—instancewise or not—would return the same result: no parallelism can be found in this program. However, a more precise observation shows that when the current control word w contains both a and b or both c and d, `p` and `q` may never be aliased again in all *descendants* of w (words such as w is a strict prefix). This proves the correctness of the abstract parallelization of procedure `P` in Figure 5.43.b (recall that CurIns stands for the run-time value of the control word). As soon as both branches of the same conditional have been taken, all recursive calls can be executed asynchronously. This yields in practice a huge amount of parallelism—an average logarithmic parallel complexity.

Eventually, this motivating example shows the need for an instancewise parallelization technique for recursive programs. Of course, such a technique requires more information

```
P  void P (int *p, int *q) {
s    p->v = ···;
t    q->v = ···;
a    if (···) P (p->l, q);
b    else P (p, q->r);
c    if (···) P (p->r, q);
d    else P (p, q->l);
}

   int main () {
F    P (tree, tree);
   }
```

Figure 5.43.a. Procedure `P`

```
P  void P (int *p, int *q) {
s    p->v = ···;
t    q->v = ···;
a    if (···) spawn P (p->l, q);
b    else spawn P (p, q->r);
     if (CurIns ∈ (a + d)* + (b + c)*) sync
c    if (···) spawn P (p->r, q);
d    else spawn P (p, q->l);
   }

   int main () {
F    P (tree, tree);
   }
```

Figure 5.43.b. Abstract parallelization of `P`

Figure 5.43. Instancewise parallelization example

than a simple dependence test: a precise description of the instances in dependence is the key for instancewise parallelism detection.

Algorithm

We now present an algorithm to automatically detect instancewise parallelism in recursive programs, and to generate the parallel code. This technique naturally extends the previous statementwise algorithm, but synchronization statements are now *guarded* by membership of the current run-time instance to *rational* subsets of L_{CTRL}—the whole language of control words. The idea consists in guarding every `sync` statement with the domain of relation *synchro* in STATEMENTWISE-PARALLELIZATION. In the case of algebraic relations, this domain is an algebraic language and membership may not be decided efficiently, we then compute a rational approximation of the domain before generating the code.

Instancewise parallelization algorithm INSTANCEWISE-PARALLELIZATION is based on the statementwise version, and it generates a "next state" function $alpha : Q \times \Sigma_{\text{CTRL}} \to Q$ for online computation of the CURINS $\in$ *set* condition. This function is usually implemented with a two-dimensional array, see the example below.[26]

The result of INSTANCEWISE-PARALLELIZATION applied to procedure `P` is shown in Figure 5.44. It is basically the same parallelization as the abstract code in Figure 5.43.a, but the synchronization condition is now fully implemented: the deterministic automata used for online recognition of $(a + d)^* + (b + c)^*$ is given in Figure 5.44.b. Transitions are stored in array `next`, the first dimension is indexed by state numbers and the second by statement labels.

Notice the parallelization technique proposed by Feautrier in [Fea98] would also fail on this example, because it is a dependence *test* only: it cannot be used to compute at compile-time which instances of procedure `P` allow asynchronous execution of the recursive

[26] An extension to *deterministic* algebraic languages would be rather easy to design, and would sometimes give better results for recursive programs with arrays. Nevertheless, it requires computation of a deterministic approximation of an algebraic language, which is much more difficult than a rational approximation.

Instancewise-Parallelization $(program, \kappa)$
  program: an intermediate representation of the program
  κ: the conflict relation to be satisfied by all parallel execution orders
  returns a parallel implementation of *program*

$depend \leftarrow \kappa \cap ((\mathbf{W} \times \mathbf{R}) \cup (\mathbf{R} \times \mathbf{W}) \cup (\mathbf{W} \times \mathbf{W}))$
$(\Sigma_{\text{CTRL}}, edges) \leftarrow$ dual control flow graph of *program*
for each ST **in** *edges*
do $B \leftarrow$ innermost block surrounding both S and T
  $synhro \leftarrow depend \cap (\Sigma^*_{\text{CTRL}} B(\Sigma_{\text{CTRL}} - \{B\})^* S \Sigma^*_{\text{CTRL}}$
    $\times\, \Sigma^*_{\text{CTRL}} B(\Sigma_{\text{CTRL}} - \{B\})^* T \Sigma^*_{\text{CTRL}})$
  $set \leftarrow$domain of relation *synchro*
  if $set \neq \varnothing$
   then if *set* is algebraic
      then $set \leftarrow$rational approximation of *set*
     $(Q, \{q_0\}, F, E) \leftarrow$determinization of *set*
     compute a "next state" function α from $(Q, \{q_0\}, F, E)$
     define a global variable `state =` q_0
     for each procedure **in** *program*
     do insert a new argument *state* in the first place
     for each call to a procedure p **in** *program*
     do insert a new argument $\alpha($`state`$, p)$ in the first place
     for each non-procedure block b **in** *program*
     do define a local variable `state =` $\alpha($`state`$, b)$
     insert "`if (state` $\in F$`) sync`" at program point associated with ST
  insert a `spawn` keyword before every statement
return *program*

calls.

5.6 Conclusion

In this chapter, we studied automatic parallelization techniques based on memory expansion. Expanding data structures is a classical optimization to cut memory-based dependences. The first problem is to ensure that all reads refer to the correct memory location, in the generated code. When control and data flow cannot be known at compile-time, run-time computations have to be done to find the identity of the correct memory location. The second problem is that converting programs to single-assignment form is too costly, in terms of memory usage.

When dealing with unrestricted nests of loops and arrays, we have tackled both problems. We proposed a general method for *static expansion* based on instancewise reaching definition information, a robust run-time *data-flow restoration* scheme, and a versatile *storage mapping optimization* technique. Our techniques are either novel or generalize previous work to unrestricted nests of loops. Eventually, all these techniques were combined in a simultaneous expansion and parallelization framework, based on *expansion constraints*. Many algorithms were designed, from single-assignment conversion to constrained storage mapping optimization and efficient data-flow restoration. This work advocates for the use of constrained expansion in parallelizing compilers. The goal is now to design pragmatic constraints and to propose a real bi-criteria optimization algorithm

```
   int state = 0;
   int next[4, 4] = {{1,2,2,1}, {1,3,3,1}, {2,3,3,2}, {3,3,3,3}};

P  void P (int state, int *p, int *q) {
s    p->v = ···;
t    q->v = ···;
a    if (···)
       spawn P (next[state, 0], p->l, q);
b    else
       spawn P (next[state, 1], p, q->r);
     if (state == 3) sync
c    if (···)
       spawn P (next[state, 2], p->r, q);
d    else
       spawn P (next[state, 3], p, q->l);
   }

   int main () {
F    P (state, tree, tree);
   }
```

Figure 5.44.a. Parallel code

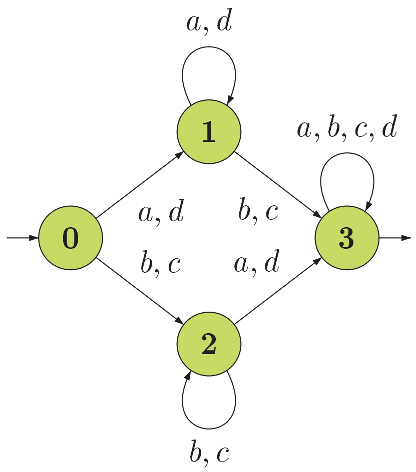


Figure 5.44.b. Automaton to decide synchronization at run-time

Figure 5.44. Automatic instancewise parallelization of procedure `P`

for expansion overhead and parallelism extraction.

The second part of this chapter discussed *parallelization of recursive programs*. We investigated memory expansion of recursive programs, which is a *new issue* in automatic parallelization. Single-assignment and privatization were extended to recursive programs, based on the rational and algebraic transduction results of our analysis for recursive programs. Difficult problems related with online computation of reaching definitions and run-time data-flow restoration where investigated. Extending constrained expansion and storage mapping optimization to recursive programs is left for future work, but several unresolved issues for simpler expansion schemes must be investigated first. Eventually, we showed that the rational or algebraic transductions returned by dependence analysis could be used to extract control parallelism. A simple algorithm to decide whether two statements can be executed in parallel has been designed and applied to an example—in combination with the privatization technique. This algorithm achieves better results than most existing techniques, because it is based on a very precise—and instancewise—dependence information. These good results motivate further researches in dependence analysis of recursive programs. Another contribution is the algorithm for *instancewise parallelization*: it decides at compile-time whether two instances of a statement can be executed in parallel or not. Common in the case of nested loops, this technique is completely *new for recursive programs*. However, algorithms proposed are still rather primitive: they neither perform statement reordering nor integrate architecture parameters such as the minimal grain of parallel tasks. Fortunately, these issues have been widely studied in more classical parallelization frameworks and we hope that the same solutions would apply to our own framework.

Future work is threefold. First, improve optimization of the generated code and study—both theoretically and experimentally—the effect of ϕ functions on parallel code performance. Second, study how comprehensive parallelization techniques can be plugged into the constrained storage mapping optimization framework: reducing memory usage is a good thing, but choosing the right parallel execution order is another. Third, proceed in an extensive study of the applicability of memory expansion techniques for parallelization of recursive programs.

Chapter 6

Conclusion

We now conclude this thesis by a summary of the main results and contributions, followed by a discussion of perspectives and future works.

6.1 Contributions

Our main contributions can be divided into four closely related parts. The first three parts address automatic parallelization and are summarized in the next table, and the fourth one is about rational and algebraic transductions. Not all contributions in this table are well matured and ready to use results: most of the work about recursive programs should be seen as a first attempt to extend instancewise analysis and transformation techniques to a larger class of programs.

	Affine loop nests with arrays	Unrestricted loop nests with arrays	Recursive programs with arrays and trees
Instancewise dependence analysis	[Bra88, Ban88] [Fea88a, Fea91, Pug92]	[BCF97, Bar98] [WP95, Won95]	[Fea98],[1] Chapter **4**, published in [CC98][2]
Instancewise reaching definition analysis	[Fea88a, Fea91, Pug92] [MAL93]	[CBF95, BCF97, Bar98] [WP95, Won95]	Chapter **4**, published in [CC98][2]
Single-assignment form	[Fea88a, Fea91]	[Col98], Sections **5.1** and **5.4**	Section **5.5**
Maximal static expansion	Sections **5.2** and **5.4**, published in [BCC98, Coh99b, BCC00]		*open problem*
Storage mapping optimization	[LF98, Lef98] [SCFS98, CDRV97]	Sections **5.3** and **5.4**, published in [CL99, Coh99b]	*open problem*
Instancewise parallelization	[Fea92, CFH95] [DV97]	[GC95, CBF95] [Col95b]	Section **5.5**

Let us now review every contribution in more detail.

[1]Dependence *test* for *trees* only.
[2]For *arrays* only.

Control and Data Structures: Beyond the Polyhedral Model In Chapter 2, we defined a program model and mathematical abstractions for statement instances and elements of data structures. This framework was used throughout this work to give a formal presentation of our techniques, especially when dealing with recursive control and data structures.

Novel instancewise dependence and reaching definition analyses for recursive programs were proposed in Chapter 4, based on formal language theory, and more precisely on rational and algebraic transductions. Using a new definition of induction variables in recursive programs, we could capture the effect of every run-time instance of a statement in a rational or algebraic transduction. Because conditionals and loop bounds are unrestricted, we could achieve only approximate results in general. A summary of program model restrictions and a comparison with other dependence and reaching definition analyses concludes this work.

However, when designing algorithms for nested loops and arrays—a special case of the program model—we sticked to the classical iteration vector framework, and we took benefit of the wealth of algorithms to work with affine relations in Presburger arithmetics.

Memory Expansion: New Techniques to Solve New Problems Parallelization via memory expansion is an old technique, but the recent extension of instancewise reaching definition analyses to programs with conditionals, complex data structure references—e.g. non-affine array subscripts—or recursive calls raises new questions. The first one is to ensure that read accesses in the expanded program refer to the correct memory location; the second is that existing techniques for memory expansion have to be extended to fit the new program models.

We addressed both questions in the first four sections of Chapter 5, when dealing with unrestricted nested loops and arrays. A new technique to reduce the run-time overhead of memory expansion has been proposed, and another technique to reduce memory usage has been extended to unrestricted loop nests. Combination of the two techniques has also been studied. Eventually, we designed several algorithms to optimize run-time restoration of the flow of data (when it is mandatory). We also discussed experimental results on a shared-memory architecture.

Memory expansion for recursive programs is a completely new topic, and we discovered that the mathematical abstraction for reaching definitions—rational and algebraic transductions—may incur a severe run-time overhead. Nevertheless, in a few particular cases we could design algorithms to generate low-overhead expanded recursive programs.

Parallelism: Extending Classical Techniques Our new dependence analysis technique has been shown useful to parallelizing recursive programs. It demonstrates the applicability of rational and algebraic transductions, thanks to their decidable properties. The first algorithm we presented is similar to existing parallelization methods for recursive programs, but it takes benefit of the additional information captured by our analysis to achieve better results in general. Another algorithm addresses instancewise parallelization of recursive programs: this new technique is made possible by the instancewise information captured in rational and algebraic transductions. A few experimental results were discussed, combining expansion and parallelization on a well known recursive program.

Formal Language Theory: Several Contributions and Applications The last results of this work do not belong to compilation. They are mostly found in the third section of Chapter 3—presenting useful mathematical abstractions—and some in the following sections. We designed a sub-class of rational transductions with boolean algebra structure and many other interesting properties. We showed that this class is not decidable among rational transductions, but conservative approximation techniques allow to take benefit of these properties in the whole class of rational transductions. We also presented some new results about composition of rational transductions over non-free monoids and investigated approximation of algebraic transductions.

6.2 Perspectives

Many questions arose along this thesis, and our results motivate more interesting studies than it solves problems. We start with questions related with recursive programs, then discuss future work in the polyhedral model.

First of all, looking for the good mathematical abstraction to capture instancewise properties appeared once more as a critical issue. Rational and algebraic transductions have been successful in many cases, but the lack of expressiveness has often limited their applications. Reaching definition analysis has most suffered of these limitations, as well as integration of conditional expressions and loop bounds in dependence analysis. In this context, we would like to consider more than one counter in a transducer, and still be able to decide emptiness and other useful properties. We are thus very interested in the work by Comon and Jurski [CJ98] on deciding the emptiness for a sub-class of multi-counter languages, and more generally in studies about system verification based on restricted classes of Minsky machines, such as timed automata. In addition, using several counters would allow us to extend one of the major ideas underlying fuzzy array dataflow analysis [CBF95]: inserting new parameters to capture properties of non-affine expressions and improve precision.

Moreover, we believe that decidability of the mathematical abstraction is not the most important thing for program analysis: a few good approximate results are often sufficient. In particular, we discovered when studying deterministic and left-synchronous relations that a nice sub-class with good decidability properties cannot be used in our framework without an efficient approximation method. Improving our techniques to resynchronize rational transducers and approximate them by left-synchronous ones is thus an important issue. We also hope that this demonstrates the high mutual interest of cooperations between theoretical computer scientists and compilation researchers.

Besides these formal aspects, another research issue is to alleviate as many restrictions as possible in the program model. As hinted before, the best way consists in looking for a graceful degradation of our results using approximation techniques. This idea has been investigated in a similar context [CBF95], and studying its applicability to recursive programs is an interesting future work. Another idea would be to perform induction variable computation on execution traces (instead of control words)—allowing induction variable update in every program statement—then to deduce approximate information on control words; relying on abstract interpretation techniques [CC77] would perhaps be helpful in proving the correctness of our approximations.

The interest of memory expansion for recursive programs is still unclear, because of the high overhead to compute reaching definitions at run-time—either exactly or with ϕ functions. Pragmatic techniques similar to privatization—i.e. making a global variable local to each procedure—seem more promising, but require further study. Working on an extension of maximal static expansion and storage mapping optimization to recursive programs is perhaps too early in this context, but transitive closure, class enumeration and graph coloring techniques for rational and algebraic transductions are interesting open problems.

We have not addressed the problem of scheduling recursive programs, because the way to assign sets of run-time instances to logical execution dates is unknown. Building a rational transducer from dates to instances is perhaps a good idea, but the problem of generating the code to enumerate the precise sets of instances becomes rather difficult. Besides these technical reasons, most parallelism in recursive programs can already been exploited by control parallel techniques, and the need for a data parallel execution model is not obvious.

In addition to motivating a large part of our work on recursive programs, techniques from the polyhedral model cover an important part of this thesis. An major goal throughout his work was to keep some distance with the mathematical representation of affine relations. One drawback of this point of view is the increased difficulty to build optimized algorithms ready to be used in a compiler, but the big advantage is the generality of the approach. Among the technical problems that should be improved in both maximal static expansion and storage mapping optimization, the most important are the following.

Many algorithms for run-time restoration of the data flow have been designed, but practical experience with parallelization of loop nests with unpredictable control flow and non-affine array subscripts is still very low. Because the SSA framework [CFR+91] is mainly used as an intermediate representation, ϕ functions are rarely implemented in practice. Generating an efficient data-flow restoration code is thus a rather new problem.

No parallelizing compiler for unrestricted nested loops has been designed. As a result, a large scale experiment has never been performed. To apply precise analysis and transformation techniques to real programs, an important work in optimizing the techniques must be done. The main ideas would be code partitioning [Ber93] and extending our techniques to hierarchical dependence graphs, array regions [Cre96] or hierarchical schedules [CW99].

A parallelizing compiler must be able to tune automatically a large number of parameters: run-time overhead, parallelism extraction, parallelization grain, copy-in and copy-out, schedule latency, memory hierarchy, memory usage, placement of computations and communications... And we have seen that the optimization problem is even more complex for non-affine loop nests. Our constrained expansion framework allows simultaneous optimization of some parameters related with memory expansion, but this is only a first step.

Bibliography

[AB88] J.-M. Autebert and L. Boasson. *Transductions rationnelles.* Masson, Paris, France, 1988.

[AFL95] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *ACM Symp. on Programming Language Design and Implementation (PLDI'95)*, pages 174–185, La Jolla, California, USA, June 1995.

[AI91] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loop. In *3^rd ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'91)*, pages 39–50, June 1991.

[AK87] 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.

[Ala94] M. Alabau. *Une expression des algorithmes massivement parallèles à structures de données irrégulières.* PhD thesis, Université Bordeaux I, September 1994.

[Amm92] Z. Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Trans. on Software Engineering*, 18(3):237–251, March 1992.

[AR94] R. Andonov and S. Rajopadhye. A sparse knapsack algo-tech-cuit and its synthesis. In *Int. Conf. on Application-Specific Array Processors (ASAP'94)*, pages 302–313, San-Francisco, California, USA, August 1994. IEEE Computer Society Press.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Bak77] B. S. Baker. An algorithm for structuring programs. *Journal of the ACM*, 24:98–120, 1977.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, USA, 1988.

[Ban92] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, Boston, USA, 1992.

[Bar98] D. Barthou. *Array Dataflow Analysis in Presence of Non-affine Constraints.* PhD thesis, Université de Versailles, France, February 1998. `http://www.prism.uvsq.fr/~bad/these.html`.

[BBA98] H. Bourzoufi, B. Sidi Boulenouar, and R. Andonov. A tiling approach for solving dynamic programming knapsack problem recurrences. In *Rencontres francophones du parallélisme (RenPar'10)*, Strasbourg, France, June 1998.

[BC99a] M. P. Béal and O. Carton. Asynchronous sliding block maps. Technical Report IGM 99-06, Institut Gaspard Monge, Université de Marne-la-Vallée, France, 1999.

[BC99b] M.-P. Béal and O. Carton. Determinization of transducers over finite and infinite words. Technical Report (to appear), Institut Gaspard Monge, Université de Marne-la-Vallée, France, 1999.

[BCC98] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. In *25^{th}ACM Symp. on Principles of Programming Languages*, pages 98–106, San Diego, California, USA, January 1998.

[BCC00] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. *Int. Journal of Parallel Programming*, June 2000. To appear.

[BCF97] D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40:210–226, 1997.

[BDRR94] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? In *Scalable High-Performance Computing Conf.*, pages 568–576, Knoxville, Tennessee, USA, May 1994. IEEE Computer Society Press.

[BE95] W. Blume and R. Eigenmann. Symbolic range propagation. In *Proc. of the 9^{th}Int. Parallel Processing Symp. (IPPS'95)*, pages 357–363, Santa Barbara, California, USA, April 1995. IEEE Computer Society Press.

[BEF$^+$96] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[Ber79] J. Berstel. *Transductions and Context-Free Languages.* Teubner, Stuttgart, Germany, 1979.

[Ber93] J.-Y. Berthou. *Contruction d'un paralléliseur de logiciels scientifiques de grande taille guidée par des mesures de performances.* PhD thesis, Université Pierre et Marie Curie (Paris VI), France, October 1993.

[BH77] M. Blattner and T. Head. Single valued a-transducers. *Journal of Comput. and System Sci.*, 15:310–327, 1977.

[Bra88] T. Brandes. The importance of direct dependences for automatic parallelization. In *ACM Int. Conf. on Supercomputing*, pages 407–417, St. Malo, France, July 1988.

[CBC93] J.-D. Choi, M. Burke, and P. Carlini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20^{th}ACM Symp. on Principles of Programming Languages (PoPL'93)*, pages 232–245, Charleston, South Carolina, USA, January 1993.

[CBF95] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *ACM Symp. on Principles and Practice of Parallel Programming*, pages 92–102, Santa Barbara, California, USA, July 1995.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *4^{th}ACM Symp. on Principles of Programming Languages*, pages 238–252, Los Angeles, California, USA, January 1977.

[CC98] A. Cohen and J.-F. Collard. Instance-wise reaching definition analysis for recursive programs using context-free transductions. In *Parallel Architectures and Compilation Techniques*, pages 332–340, Paris, France, October 1998. IEEE Computer Society Press. (IEEE award for the best student paper).

[CCG96] A. Cohen, J.-F. Collard, and M. Griebl. Data-flow analysis of recursive structures. In *Proc. of the 6^{th}Workshop on Compilers for Parallel Computers*, pages 181–192, Aachen, Germany, December 1996.

[CDRV97] P.-Y. Calland, A. Darte, Y. Robert, and Frédéric Vivien. Plugging anti and output dependence removal techniques into loop parallelization algorithms. *Parallel Computing*, 23(1–2):251–266, 1997.

[CFH95] L. Carter, J. Ferrante, and S. Flynn Hummel. Efficient multiprocessor parallelism via hierarchical tiling. In *SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[CFR+91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. 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.

[CFR95] J.-F. Collard, P. Feautrier, and T. Risset. Construction of DO loops from systems of affine constraints. *Parallel Processing Letters*, 5(3), 1995.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5^{th}ACM Symp. on Principles of Programming Languages*, pages 84–96, January 1978.

[Cho77] C. Choffrut. Une caractérisation des fonctions séquentielles et des fonctions sous-séquentielles en tant que relations rationnelles. *Theoretical Computer Science*, 5:325–338, 1977.

[CI96] B. Creusillet and F. Irigoin. Interprocedural array region analyses. *Int. Journal of Parallel Programming*, 24(6):513–546, December 1996.

[CJ98] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In A. Hu and M. Vardi, editors, *Proc. Computer Aided Verification*, volume 1427 of *LNCS*, pages 268–279, Vancouver, Britich Columbia, Canada, 1998. Springer-Verlag.

[CK98] J.-F. Collard and J. Knoop. A comparative study of reaching definitions analyses. Technical Report 1998/22, Laboratoire PRiSM, Université de Versailles, France, 1998.

[CL99] A. Cohen and V. Lefebvre. Optimization of storage mappings for parallel programs. In *EuroPar'99*, number 1685 in LNCS, pages 375–382, Toulouse, France, September 1999. Springer-Verlag.

[Cla96] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *ACM Int. Conf. on Supercomputing*, pages 278–295. ACM Press, 1996.

[Coh97] A. Cohen. Analyse de flot de données de programmes récursifs à l'aide de grammaires hors-contexte. In *Rencontres francophones du parallélisme (RenPar'9)*, Lausanne, Suisse, May 1997. (IEEE award for the best french-speaking student paper).

[Coh99a] A. Cohen. Analyse de flot de données pour programmes récursifs à l'aide de langages algébriques. *Technique et science informatiques*, 18(3):323–343, 1999.

[Coh99b] A. Cohen. Parallelization via constrained storage mapping optimization. In *Int. Symp. on High Performance Computing (ISHPC'99)*, number 1615 in LNCS, pages 83–94, Kyoto, Japan, May 1999. Springer-Verlag.

[Col94a] J.-F. Collard. Code generation in automatic parallelizers. In C. Girault, editor, *Proc. of the Int. Conf. on Applications in Parallel and Distributed Computing, IFIP W.G. 10.3*, pages 185–194, Caracas, Venezuela, April 1994. North Holland.

[Col94b] J.-F. Collard. Space-time transformation of while-loops using speculative execution. In *Scalable High Performance Computing Conf.*, pages 429–436, Knoxville, Tennessee, USA, May 1994. IEEE Computer Society Press.

[Col95a] J.-F. Collard. Automatic parallelization of while-loops using speculative execution. *Int. Journal of Parallel Programming*, 23(2):191–219, April 1995.

[Col95b] J.-F. Collard. *Parallélisation automatique des programmes à contrôle dynamique*. PhD thesis, Université Pierre et Marie Curie (Paris VI), France, January 1995.
`http://www.prism.uvsq.fr/~jfc/memoire.ps`.

[Col98] J.-F. Collard. The advantages of reaching definition analyses in Array (S)SA. In *11th Workshop on Languages and Compilers for Parallel Computing*, number 1656 in LNCS, pages 338–352, Chapel Hill, North Carolina, USA, August 1998. Springer-Verlag.

[Cou81] P. Cousot. *Semantic foundations of programs analysis*. Prentice-Hall, 1981.

[Cre96] B. Creusillet. *Array Region Analyses and Applications*. PhD thesis, École Nationale Supérieure des Mines de Paris (ENSMP), Paris, France, December 1996.

[CW99] J. B. Crop and D. K. Wilde. Scheduling structured systems. In *EuroPar'99*, LNCS, pages 409–412, Toulouse, France, September 1999. Springer-Verlag.

[Deu90] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *17^{th}ACM Symp. on Principles of Programming Languages (PoPL'90)*, pages 157–168, San Francisco, California, USA, January 1990.

[Deu92] A. Deutsch. *Operational Models of Programming Languages and Representations of Relations on Regular Languages with Application to the Static Determination of Dynamic Aliasing Properties of Data.* PhD thesis, École Polytechnique, France, April 1992.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *ACM Symp. on Programming Language Design and Implementation (PLDI'94)*, pages 230–241, Orlando, Florida, USA, June 1994.

[DGS93] E. Duesterwald, R. Gupta, and M.-L. Soffa. A practical data flow framework for array reference analysis and its use in optimization. In *ACM Symp. on Programming Language Design and Implementation (PLDI'93)*, pages 68–77, Albuquerque, New Mexico, USA, jun 1993.

[DV97] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *Int. Journal of Parallel Programming*, 25(6):447–496, December 1997.

[EGH94] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM Symp. on Programming Language Design and Implementation (PLDI'94)*, pages 242–256, June 1994.

[Eil74] S. Eilenberg. *Automata, Languages and Machines*, volume A. Academic Press, 1974.

[EM65] C. C. Elgot and J. E. Mezei. On relations defined by generalized finite automata. *IBM Journal of Research and Development*, pages 45–68, 1965.

[FB98] P. Feautrier and P. Boulet. Scanning polyhedra without do-loops. In *Parallel Architectures and Compilation Techniques (PACT'98)*, Paris, France, October 1998. IEEE Computer Society Press.

[Fea88a] P. Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing*, pages 429–441, St. Malo, France, July 1988.

[Fea88b] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[Fea91] P. Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53, February 1991.

[Fea92] P. Feautrier. Some efficient solution to the affine scheduling problem, part II, multidimensional time. *Int. Journal of Parallel Programming*, 21(6):389–420, December 1992. See also Part I, One Dimensional Time, 21(5):315–348.

[Fea98] P. Feautrier. A parallelization framework for recursive tree programs. In *EuroPar'98*, LNCS, Southampton, UK, September 1998. Springer-Verlag.

[FM97] P. Fradet and D. Le Metayer. Shape types. In *24thACM Symp. on Principles of Programming Languages (PoPL'97)*, pages 27–39, Paris, France, January 1997.

[FS93] C. Frougny and J. Sakarovitch. Synchronized relations of finite words. *Theoretical Computer Science*, 108:45–82, 1993.

[GC95] M. Griebl and J.-F. Collard. Generation of synchronous code for automatic parallelization of `while` loops. In S. Haridi, K. Ali, and P. Magnusson, editors, *EuroPar'95*, volume 966 of *LNCS*, pages 315–326. Springer-Verlag, 1995.

[GH95] R. Ghiya and L. J. Hendren. Connection analysis: A practical interprocedural heap analysis for c. In *8thWorkshop on Languages and Compilers for Parallel Computing*, number 1033 in LNCS, Columbus, Ohio, USA, August 1995. Springer-Verlag.

[GH96] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *23rdACM Symp. on Principles of Programming Languages (PoPL'96)*, pages 1–15, St. Petersburg Beach, Florida, USA, January 1996.

[GL97] M. Griebl and C. Lengauer. The loop parallelizer LooPo — announcement. *LNCS*, 1239:603–607, 1997.

[Gup98] R. Gupta. A code motion framework for global instruction scheduling. In *Int. Conf on Compiler Construction (CC'98)*, pages 219–233, 1998.

[H^{+}96] M. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[Har89] W. L. Harrison. The interprocedural analysis and automatic parallelisation of scheme programs. *Lisp and Symbolic Computation*, 2(3):176–396, October 1989.

[HBCM94] M. Hind, M. Burke, P. Carini, and S. Midkiff. An empirical study of precise interprocedural array analysis. *Scientific Programming*, 3(3):255–271, 1994.

[HHN92] L. J. Hendren, J. Hummel, , and A. Nicolau. Abstractions for recursive pointer data structures: improving the analysis and transformation of imperative programs. In *ACM Symp. on Programming Language Design and Implementation (PLDI'92)*, pages 249–260, San Francisco, Calfifornia, USA, June 1992.

[HHN94] J. Hummel, L. J. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *ACM Symp. on Programming Language Design and Implementation (PLDI'94)*, pages 218–229, Orlando, Florida, USA, June 1994.

[HP96] M. Haghighat and C. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Trans. on Programming Languages and Systems*, 18(4):477–518, July 1996.

[HTZ+97] L. J. Hendren, X. Tang, Y. Zhu, S. Ghobrial, G. R. Gao, X. Xue, H. Cai, and P. Ouellet. Compiling C for the EARTH multithreaded architecture. *Int. Journal of Parallel Programming*, 25(4):305–338, August 1997.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[IJT90] F. Irigoin, P. Jouvelot, and R. Triolet. Overview of the PIPS project. In P. Feautrier and F. Irigoin, editors, *2nd Int. Workshop on Compilers for Parallel Computers*, pages 199–212, Paris, December 1990.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *15th ACM Symp. on Principles of Programming Languages (PoPL'88)*, pages 319–328, San Diego, California, USA, January 1988.

[JM82] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. *ACM Press*, 1982.

[Kar92] G. Karner. Nivat's theorem for pushdown transducers. *Theoretical Computer Science*, 97:245–262, 1992.

[KPRS96] W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its applications. *Int. Journal of Parallel Programming*, 24(6):579–598, 1996.

[KRS94] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1117–1155, 1994.

[KS92] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Proc. of the 4th Int. Conference on Compiler Construction (CC'92)*, number 641 in LNCS, Paderborn, Germany, 1992.

[KS93] N. Klarlund and M. I. Schwartzbach. Graph types. In *20th ACM Symp. on Principles of Programming Languages (PoPL'93)*, pages 196–205, Charleston, South Carolina, USA, January 1993.

[KS98] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *25th ACM Symp. on Principles of Programming Languages*, pages 107–120, San Diego, California, USA, January 1998.

[KSV96] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):268–299, May 1996.

[KU77] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309–317, 1977.

[Lef98] V. Lefebvre. *Restructuration automatique des variables d'un programme en vue de sa parallélisation*. PhD thesis, Université de Versailles, France, February 1998.
`http://www.prism.uvsq.fr/~vil/these.ps.gz`.

[LF98] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3):649–671, 1998.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *ACM Symp. on Programming Language Design and Implementation (PLDI'88)*, pages 21–34, 1988.

[Li92] Z. Li. Array privatization for parallel execution of loops. In *ACM Int. Conf. on Supercomputing*, pages 313–322, Washington, District of Columbia, USA, July 1992. ACM Press.

[LL97] A. W. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *24thACM Symp. on Principles of Programming Languages*, pages 201–214, Paris, France, jan 1997.

[LRZ93] W. A. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *ACM Symp. on Programming Language Design and Implementation (PLDI'93)*, pages 56–67, Albuquerque, New Mexico, USA, June 1993.

[MAL93] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array dataflow analysis and its use in array privatization. In *20thACM Symp. on Principles of Programming Languages*, pages 2–15, Charleston, South Carolina, USA, January 1993.

[Mas93] F. Masdupuy. Semantic analysis of interval congruences. In D. Børner, M. Broy, and I. V. Pottosin, editors, *Int. Conf. on Formal Methods in Programming and their Applications*, volume 735 of *LNCS*, pages 142–155, Academgorodok, Novosibirsk, Russia, June 1993. Springer-Verlag.

[MF98] K. H. Randall M. Frigo, C. E. Leiserson. The implementation of the Cilk-5 multithreaded language. In *ACM Symp. on Programming Language Design and Implementation (PLDI'98)*, pages 212–223, Montreal, Canada, June 1998.

[Mic95] O. Michel. Design and implementation of 81/2, a declarative data-parallel language. Technical Report 1012, Laboratoire de Recherche en Informatique, Université Paris Sud (Paris XI), France, 1995. Contains paper Group-based Fields with J.-L. Giavitto and Jean-Paul Sansonnet, Proc. of the Parallel Symbolic Languages and Systems, October 1995.

[Min67] M. Minsky. *Computation, Finite and Infinite Machines.* Prentice-Hall, 1967.

[MP94] V. Maslov and W. Pugh. Simplifying polynomial constraints over integers to make dependence analysis more precise. Technical Report CS-TR-3109.1, U. of Maryland, February 1994.

[MT90] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementation.* John Wiley and Sons, 1990.

[Muc97] S. S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann, 1997.

[Par66] R. J. Parikh. On context-free languages. *Journal of the ACM*, 13(4):570–581, 1966.

[PD96] G. R. Perrin and A. Darte, editors. *The Data Parallel Programming Model*. Number 1132 in LNCS. Springer-Verlag, 1996. For scheduling issues, see “Automatic Parallelization in the Polytope Model”, pages 79–103.

[PS98] M. Pelletier and J. Sakarovitch. On the representation of finite deterministic 2-tape automata. Technical Report 98 C 002, École Nationale Supérieure des Télécommunications (ENST), Paris, France, May 1998. To appear in Theoretical Computer Science.

[Pug92] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):27–47, August 1992.

[QR99] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. Technical Report 1228, Institut de Recherche en Informatique et Systèmes Aléatoires, Université de Rennes, France, January 1999.

[RF94] X. Redon and P. Feautrier. Scheduling reductions. In *ACM Int. Conf. on Supercomputing*, pages 117–125, Manchester, UK, July 1994.

[Rin97] M. Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. In *6^{th}ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'97)*, pages 112–123, Las Vegas, Nevada, USA, June 1997.

[RR99] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *7^{th}ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'99)*, Atlanta, Georgia, USA, May 1999.

[RS97a] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 1: Word Language Grammar. Springer-Verlag, 1997.

[RS97b] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 3: Beyond Words. Springer-Verlag, 1997.

[SCFS98] M. M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independant storage mapping for loops. In *ACM Symp. on Architecture Support for Programming Languages and Operating Systems*, 8, 1998.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, Chichester, UK, 1986.

[SKR90] B. Steffen, J. Knoop, and O. Rüthing. The value flow graph: A program representation for optimal program transformations. In *Proc. of the 3^{rd}European Symp. on Programming (ESOP'90)*, volume 432 of *LNCS*, pages 389–405, Copenhagen, Denmark, May 1990.

[SRH96] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *IEEE Trans. on Computers*, 167(1–2):131–170, October 1996.

[SRW96] S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *23rdACM Symp. on Principles of Programming Languages (PoPL'96)*, pages 16–31, St. Petersburg Beach, Florida, USA, January 1996.

[SSP99] H. Saito, N. Stavrakos, and C. Polychronopoulos. Multithreading runtime support for loop and functional parallelism. In *Int. Symp. on High Performance Computing (ISHPC'99)*, number 1615 in LNCS, pages 133–144, Kyoto, Japan, May 1999. Springer-Verlag.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *23rdACM Symp. on Principles of Programming Languages (PoPL'96)*, pages 32–41, St. Petersburg Beach, Florida, USA, January 1996.

[TD95] O. Temam and N. Drach. Software assistance for data caches. *Future Generation Computer Systems*, 1995. Special issue on high performance computer architectures.

[TFJ86] R. Triolet, P. Feautrier, and P. Jouvelot. Automatic parallelization of fortran programs in the presence of procedure calls. In *Proc. of the 1stEuropean Symp. on Programming (ESOP'86)*, number 213 in LNCS, pages 210–222. Springer-Verlag, March 1986.

[TP93] P. Tu and D. Padua. Automatic array privatization. In *6thWorkshop on Languages and Compilers for Parallel Computing*, number 768 in LNCS, pages 500–521, Portland, Oregon, USA, August 1993.

[TP95] 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.

[Tzo97] S. Tzolovski. Data dependences as abstract interpretations. In *International Static Analysis Symposium SAS'97*, Paris, France, 1997.

[Wol92] M. Wolfe. Beyond induction variables. In *ACM Symp. on Programming Language Design and Implementation (PLDI'92)*, pages 162–174, San Francisco, California, USA, June 1992.

[Won95] D. G. Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, University of Maryland, 1995.

[WP95] D. Wonnacott and W. Pugh. Nonlinear array dependence analysis. In *Proc. Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, 1995. Troy, New York, USA.

[WR93] D. K. Wilde and S. Rajopadhye. Allocating memory arrays for polyhedra. Technical Report 749, Institut de Recherche en Informatique et Systèmes Aléatoires, Université de Rennes, France, July 1993.

Index

Symbols

$<_{\text{LEX}}$, *70*, *see* lexicographic order, 75, 140, 197

$<_{\text{PAR}}$, *81*, *see* parallel execution order

$<_{\text{SEQ}}$, *70*, *see* sequential execution order

$<_{\text{TXT}}$, *70*, *see* textual order, 144

Σ_{CTRL}, *66*, *see* statement label

L_{CTRL}, *68*, *see* control word, 70, 129, 139

L_{DATA}, *71*, *see* data structure abstraction, 140

M_{DATA}, *71*, *see* data structure abstraction, 129, 140

$[\![\mathtt{i}, \alpha]\!]$, *128*, *see* induction variable, 130, 135

$[\![\mathtt{i}]\!](w)$, *128*, *see* induction variable

$\mathtt{D}_{\text{EXP}}$, *156*, *see* memory expansion

$\mathtt{E}_S$, *196*, *see* expansion vector

$\mathtt{E}_S\mathtt{[}p+1\mathtt{]}$, *197*, *see* expansion degree

$\mathbf{A}$, *80*, *see* access, 82, 134

$\mathbf{A}_e$, *63*, *see* access, 80

$\mathbf{E}$, *62*, *see* program execution, 70, 129, 156, 191, 222

$\mathbf{I}$, *80*, *see* instance, 82

$\mathbf{I}_e$, *62*, *see* instance, 68, 80

$\mathbf{R}$, *80*, *see* read, 140

$\mathbf{R}_e$, *63*, *see* read *and* access, 80

$\mathbf{W}$, *80*, *see* write, 140

$\mathbf{W}_e$, *63*, *see* write *and* access, 80

Γ, *92*, *see* stack alphabet *and* push-down automaton

γ_0, *92*, *see* initial stack word *and* push-down automaton

$\langle S, x\rangle$, *75*, *see* iteration vector *and* instance

$\langle S, x, \mathtt{ref}\rangle$, *75*, *see* iteration vector *and* access

$\equiv$, *209*, *see* constraint relation, 214

$\mathfrak{R}^*$, *173*, *see* static expansion, 175

$\mathfrak{R}$, *173*, *see* static expansion

$\mathfrak{W}^*$, *217*, *see* weakened static expansion

$\mathfrak{W}$, *217*, *see* weakened static expansion

δ, *77*, *see* dependence relation, 140

δ_e, *77*, *see* dependence relation, 140

δ^{EXP}, *81*, *see* dependence relation *and* memory expansion, 82, 210, 214

δ_e^{EXP}, *81*, *see* dependence relation *and* memory expansion

κ^*, *175*, *see* conflict relation *and* static expansion

κ, *76*, *see* conflict relation, 175, 191

κ_e, *76*, *see* conflict relation, 191

$\not\kappa$, *191*, *see* no-conflict relation, 193

$\not\kappa_e$, *191*, *see* no-conflict relation

$\bowtie$, *193*, *see* interference relation, 194, 210, 211

∞, *211*, *see* interference relation, 214

χ, *212*, *see* coloring relation

$\chi_\equiv$, *213*, *see* constraint coloring relation

σ, *78*, *see* reaching definition

σ^{ML}, *164*, *see* reaching definition of a memory location *and* memory expansion

σ_e^{ML}, *164*, *see* reaching definition of a memory location *and* memory expansion

σ_e, *77*, *see* reaching definition

ϕ, *156*, *see* memory expansion, 168, 174, 217, 219

$\textsc{Joins}_{\mathtt{A}}$, *220*, *see* join

$\textsc{Points}$, *219*, *see* program point

$\textsc{Ancestors}(u)$, *142*, *see* ancestor

$\textsc{Array}$, *160*, *see* memory expansion

$\textsc{CurIns}$, *156*, *see* run-time instance *and* memory expansion, 227, 240

$\textsc{Iter}$, *160*, *see* memory expansion *and* iteration vector

$\textsc{Stmt}$, *160*, *see* memory expansion *and* iteration vector

$\textsc{Undefined}$, *130*, *see* induction variable

θ, *82*, *see* schedule, 85

ε, *91*, *see* empty word

f_e^{EXP}, *81*, *see* storage mapping *and*

memory expansion, 173, 191, 209
f_e, *75*, *see* storage mapping, 173
f, *129*, *see* storage mapping
$\mathtt{A}_S[\mathrm{x}]$, *160*, *see* memory expansion
$\Phi\mathtt{D}_{\mathrm{EXP}}$, *157*, *see* memory expansion

A
α-selection, *108*
α-selection, 141, 231
access, *63*, 75
$\mathbf{A}$, *80*, 82, 134
$\mathbf{A}_e$, *63*, 80
$\mathbf{R}$, 140
$\mathbf{R}_e$, *63*, 80
$\mathbf{W}$, 140
$\mathbf{W}_e$, *63*, 80
$\langle S, x, \mathtt{ref}\rangle$, *75*
algebraic function, *116*
algebraic grammar, *92*
algebraic language, *92*
algebraic relation, *115*
algebraic transducer, *114*, *see* push-down transducer
aliased, *65*
analysis of conflicting accesses, *76*
ancestor, *142*, 144, 148
ANCESTORS(u), *142*

B
block, *63*

C
call tree, 70
causality constraint, *82*
coloring relation, *212*
χ, *212*
complete, *105*
configuration, *93*, 114
conflict, *76*
conflict equation
κ, 175
conflict relation, *76*, 139, 140, 191, 211
κ^*, *175*
κ, *76*, 191
κ_e, *76*, 191
constrained expansion, *209*
constraint coloring relation, *213*
$\chi_\equiv$, *213*
constraint relation, *209*
$\equiv$, *209*, 214
$\not\equiv$, 214
context-free grammar, *92*
context-free language, *92*
control automaton, *67*
compressed, *69*
control parallelism, 58
control tree, *70*, 123, 142
compressed, *70*
control word, *68*
L_{CTRL}, *68*, 70, 129, 139

D
data parallelism, 59
data structure abstraction, 139
L_{DATA}, *71*, 140
M_{DATA}, *71*, 129, 140
data-flow execution order, *200*
δ-synchronizable, *102*
δ-synchronous, *102*
dependence, *77*
dependence analysis, *77*
dependence relation, *77*
δ, *77*, 140
δ_e, *77*, 140
δ^{EXP}, *81*, 82, 210, 214
δ_e^{EXP}, *81*
deterministic algebraic languages, *93*
dominance frontier, *219*
dynamic arrays, *160*

E
edge name, 64, 71, 236
empty word, *91*
ε, *91*
execution front, 60
execution trace, 66
expansion correctness criterion, *192*, 193, 194
expansion degree, *197*
$\mathtt{E}_S[p+1]$, *197*
expansion vector, *196*
$\mathtt{E}_S$, *196*

F
finer, *81*, *see* storage mapping, 174
finite-state automaton
deterministic, *91*
finitely generated, *91*, 97, 98

formal language, *91*
free monoid, *91*
free partially commutative monoid, *72*, 118

I
induction variable, *127*
  $[\![\mathtt{i}, \alpha]\!]$, *128*, 130, 135
  $[\![\mathtt{i}]\!](w)$, *128*
  UNDEFINED, *130*
  undefined value, *130*
  value at an instance, *128*
initial stack word, *92*
  γ_0, *92*
input automaton, *100*
instance, *62*, 75
  $\mathbf{I}$, *80*, 82
  $\mathbf{I}_e$, *62*, 68, 80
  $\langle S, x\rangle$, *75*
integer linear programming, *87*
interference relation, *193*, 210, 211
  $\bowtie$, *193*, 194, 210, 211
  ∞, *211*, 214
iteration vector
  $\langle S, x, \mathtt{ref}\rangle$, *75*
  $\langle S, x\rangle$, *75*
  ITER, *160*
  STMT, *160*
iteration vectors, *74*

J
join, *219*
  $\text{JOINS}_{\mathtt{A}}$, *220*

L
left-synchronizable, *102*
left-synchronous, *102*, 148, 231
lexicographic order, *70*, 75, 88, 103, 140
  $<_{\text{LEX}}$, *70*, 75, 140, 197
loop variable, 64

M
maximal, *211*
maximal constrained expansion, 222
maximal static expansion, *174*
memory expansion, *81*
  $\mathtt{D}_{\text{EXP}}$, *156*
  δ^{EXP}, *81*, 82, 210, 214
  δ_e^{EXP}, *81*
  σ^{ML}, *164*
  σ_e^{ML}, *164*
  ϕ, *156*, 168, 174, 217, 219
  ARRAY, *160*
  CURINS, *156*, 227, 240
  ITER, *160*
  STMT, *160*
  f_e^{EXP}, *81*, 173, 191, 209
  $\mathtt{A}_S[\mathrm{x}]$, *160*
  @-structures, *157*, *166*
  Φ-structures, *157*
  $\Phi\mathtt{D}_{\text{EXP}}$, *157*
monoid, *90*

N
no-conflict relation, *191*
  $\not\kappa$, *191*, 193
  $\not\kappa_e$, *191*

O
one-counter automaton, *94, 95*
one-counter language, *95*
one-counter relation, *116*
one-counter transducer, *116*
online algebraic transducer, *116*
online algebraic transduction, *116*, 231
online rational transducer, *101*
online rational transduction, *101*, 231
output automaton, *100*

P
parallel execution order, *81*
  $<_{\text{PAR}}$, *81*
parallelization, *81*
partial expansion, *196, 197*
partial renaming, *196*
path, *91*, *99*
  label, *91*, *99*
privatization, 233
program execution, *62*
  $\mathbf{E}$, *62*, 70, 129, 156, 191, 222
program point, *219*
  POINTS, *219*
pseudo-left-synchronizable, *119*
pseudo-left-synchronous, *119*, 148
push-down automaton, *92*
  Γ, *92*
  γ_0, *92*
  deterministic, *93*
push-down transducer, *114*

push-down automaton interpretation, *115*
underlying rational transducer, *118*, 120, 122

Q

quasi-affine selection tree, *88*, *see* quast
quast, *88*, 160, 165
quasi-affine selection tree, *88*

R

rational function, *99*
rational language, *92*
rational relation, *97*, 128
rational set, *97*, 128
rational transducer, *98*
finite-state automaton interpretation, *99*, 107
reaching definition, *77*, 173
σ, *78*
σ_e, *77*
reaching definition analysis, *78*
reaching definition of a memory location, *164*, 217
σ^{ML}, *164*
σ_e^{ML}, *164*
read, *63*
$\mathbf{R}$, *80*, 140
$\mathbf{R}_e$, *63*, 80
realize, *91*, *99*, 135, 140
by empty stack, *93*, *115*
by final state, *93*, 94, *114*, 116
recognizable relation, *97*, 148
recognizable set, *97*
regular language, *91*, *see* rational language
right-synchronizable, *103*
right-synchronous, *103*
run-time instance, *61*
CurIns, *156*, 227, 240

S

SA, 156
schedule, 59, *82*, 85
θ, *82*, 85
schedule-independent, *188*, 200
semi-group, *90*
sequential execution order, 70
$<_{\text{SEQ}}$, *70*
sequential function, *100*, 231
sequential transducer, *100*
shape analysis, *65*
single-assignment, *156*
SSA, 156
stack alphabet, *92*
Γ, *92*
statement, *63*
statement label, *66*
Σ_{CTRL}, *66*
static expansion, *173*
$\mathfrak{R}^*$, *173*, 175
$\mathfrak{R}$, *173*
κ^*, *175*
static single-assignment, 156
storage mapping, *75*, 126, 128, 135
f_e^{EXP}, *81*, 173, 191, 209
f_e, *75*, 173
finer, *81*
sub-sequential function, *101*, 231
sub-sequential transducer, *100*
synchronizable, *102*
synchronization graph, *236*
synchronous, *102*

T

textual order, *70*, 144
$<_{\text{TXT}}$, *70*, 144
tiling, *84*
tile, *84*
top stack symbol, *93*
transduction, *98*
algebraic, 115
rational, 98
recognizable, 98
transmission rate, *110*
trim, *91*, *99*

U

unambiguous, *105*
underlying rational transducer, 148
use, *77*, 173

W

weakened static expansion, *217*
$\mathfrak{W}^*$, *217*
$\mathfrak{W}$, *217*
write, *63*
$\mathbf{W}$, *80*, 140
$\mathbf{W}_e$, *63*, 80

Résumé

Les microprocesseurs et les architectures parallèles d'aujourd'hui lancent de nouveaux défis aux techniques de compilation. En présence de parallélisme, les optimisations deviennent trop spécifiques et complexes pour être laissées au soin du programmeur. Les techniques de parallélisation automatique dépassent le cadre traditionnel des applications numériques et abordent de nouveaux modèles de programmes, tels que les nids de boucles non affines, les appels récursifs et les structures de données dynamiques. Des analyses précises sont au cœur de la détection du parallélisme, elles rassemblent des informations à la compilation sur les propriétés des programmes à l'exécution. Ces informations valident des transformations utiles pour l'extraction du parallélisme et la génération de code parallèle.

Cette thèse aborde principalement des analyses et des transformations avec une vision par instances, c'est-à-dire considérant les propriétés individuelles de chaque instance d'une instruction à l'exécution. Une nouvelle formalisation à l'aide de langages formels nous permet tout d'abord d'étudier une analyse de dépendances et de définitions visibles par instances pour programmes récursifs. L'application de cette analyse à l'expansion et la parallélisation de programmes récursifs dévoile des résultats encourageants. Les nids de boucles quelconques font l'objet de la deuxième partie de ce travail. Une nouvelle étude des techniques de parallélisation fondées sur l'expansion nous permet de proposer des solutions à des problèmes d'optimisation cruciaux.

Mots-clés : parallélisation automatique, programmes récursifs, nids de boucles non affines, analyse de dépendances, analyse de définitions visibles, expansion de la mémoire.

Abstract

Compilation for todays microprocessor and multi-processor architectures is facing new challenges. Dealing with parallel execution, optimizations become overly specific and complex to be left to the programmer. Traditionally devoted to numerical applications, automatic parallelization addresses new program models, including non-affine nests of loops, recursive calls and pointer-based data structures. Parallelism detection is based on precise analyses, gathering compile-time information about run-time program properties. This information enables transformations useful to parallelism extraction and parallel code generation.

This thesis focuses on aggressive analysis and transformation techniques from an instancewise point of view, that is from individual properties of each run-time instance of a program statement. Thanks to a novel formal language framework, we first investigate instancewise dependence and reaching definition analysis for recursive programs. This analysis is applied to memory expansion and parallelization of recursive programs, and promising results are exposed. The second part of this work addresses nests of loops with unrestricted conditionals, bounds and array subscripts. Parallelization via memory expansion is revisited in this context and solutions to challenging optimization problems are proposed.

Keywords: automatic parallelization, recursive programs, non-affine loop nests, dependence analysis, reaching definition analysis, memory expansion.