

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
Celestijnenlaan 200A, B-3001 Leuven, België

EFFICIENT AUTOMATIC VERIFICATION OF LOOP AND DATA-FLOW TRANSFORMATIONS BY FUNCTIONAL EQUIVALENCE CHECKING

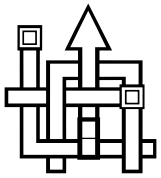
Promoters:
Prof. Dr. ir. Maurice BRUYNOOGHE
Prof. Dr. ir. Francky CATTHOOR

A dissertation submitted to the Department of Computer Science in partial fulfillment of the requirements for the degree of Doctor of Engineering Sciences

by

K. C. SHASHIDHAR

May 2008



In collaboration with

imec *VZW*

Interuniversitair Micro-Elektronica Centrum vzw
Kapeldreef 75
B-3001 Leuven, België



KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
Celestijnenlaan 200A, B-3001 Leuven, België

EFFICIENT AUTOMATIC VERIFICATION OF LOOP AND DATA-FLOW TRANSFORMATIONS BY FUNCTIONAL EQUIVALENCE CHECKING

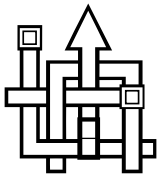
Jury:
Prof. Dr. ir. Ann Haegemans, voorzitter
Prof. Dr. ir. Maurice Bruynooghe, promoter
Prof. Dr. ir. Francky Catthoor, promoter
Prof. Dr. ir. Gerda Janssens
Prof. Dr. Bart Demoen
Prof. Dr. ir. Luc Claesen (U. Hasselt/ NCTU, Taiwan)
Prof. Dr. Denis Barthou (U. de Versailles)
Prof. Dr. ir. Jeroen Voeten (T.U. Eindhoven)

A dissertation submitted to the Department of Computer Science in partial fulfillment of the requirements for the degree of Doctor of Engineering Sciences

by

K. C. SHASHIDHAR

U.D.C. 681.3*D34 May 2008



In collaboration with

imec *VZW*

Interuniversitair Micro-Elektronica Centrum vzw
Kapeldreef 75
B-3001 Leuven, België

©Katholieke Universiteit Leuven – Faculteit Ingenieurswetenschappen
Arenbergkasteel, B-3001 Leuven – Heverlee (België)

Alle rechten voorbehouden. Niets uit deze uitgave mag vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotocopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the publisher.

D/2006/7515/11

ISBN 978-90-5682-677-2

To my parents,
Smt. U. M. Rajeevi & Sri. K. S. Chikkaputtaiah,
and my sister, Shubha.

Acknowledgments

I gratefully acknowledge that many people have advised, cared, counseled, encouraged, helped, taught and supported me before, during and after the course of my study that led to the completion of this dissertation. I thank them sincerely and wholeheartedly:

— first and foremost, for promoting and guiding my research, teaching, collaborating, co-authoring, counseling, inspiring and letting me test their infinite patience: Professor Francky Catthoor and Professor Maurice Bruynooghe. It has been a great privilege for me to be their student. It would have been impossible for me to complete this dissertation without their great support, which on many occasions went far beyond their call of duty as my promoters;

— for numerous discussions that helped evolve this work: Professors Gerda Janssens, Henk Corporaal and Luc Claesen;

— for the members of the jury for evaluating my doctoral work: Professors Ann Haegemans, Gerda Janssens, Bart Demoen, Luc Claesen, Denis Barthou and Jeroen Voeten;

— for carefully reading this dissertation and giving numerous suggestions and corrections: Sven Verdoolaege;

— for providing a project assistantship and facilitating my stay in Leuven for an additional period: Professor Maurice Bruynooghe;

— for their teaching and advise that led me to take up doctoral study: Professors M. Balakrishnan, S. Arun-Kumar, C. P. Ravikumar, T. N. Nagabhushan and Swami Manohar. It was from Professor C. P. Ravikumar that I came to know about the doctoral programme at IMEC;

— for their help and support in IMEC: Erik Brockmeyer, Koen Danckaert, Patrick David, Karine Discart, Michel Eyckmans, Eddy De Greef, Myriam Janowski, Andy Lambrechts, Rudy Lauwereins, Pol Marchal, Martin Palkovic, Praveen Raghavan, Karel Van Oudheusden, Annemie Stas, Arnout Vandecappelle, Frederik Vermeulen, Diederick Verkest, Johan Vounckx, Sven Wuytack, Ma Zhe, and many others;

— for many discussions relating to my work, help and support in CS Department at KUL: Peter Vanbroekhoven, Sven Verdoolaege, Fu Qiang and Professor Frank Piessens;

— for being my local guardians and taking care of my well-being in Leuven: Prabhat Avasare, Murali Jayapala and Harshada Samant;

— for the good banter, agreements and disagreements: Javed Absar, Prabhat Avasare, Saher Islam, Murali Jayapala, Karel Van Oudheusden, Harshada Samant, M. G. Santhosh and Siddharth Srivastava;

— for the good time while sharing the apartment on Celestijnenlaan: M. G. Santhosh;

— for the fun and friendship I had in Leuven and other places: Javed Absar, Prabhat Avasare, Bikramjit Basu, Francky Catthoor, Uday Chitraghar, Nitin Chandrachoodan, Dr. Chandrasekar, Arun Chandrasekhar, Stephane Delbeke, Samuel Xavier de Souza, Yukiko Furukawa, Prashanth S. Harish, Lucía Vergara Herrero, Saher Islam, Ilya Issenin, Murali Jayapala, Antony Joseph, Nimi Joseph, Chidamber Kulkarni, Anil Kottantharayil, Reena Lasrado, Patrick Mulder, Basavaraj Madivala, Vivek Nema, Karel Van Oudheusden, Nacho Gómez Pérez, Praveen Raghavan, Vijayaraghavan Raghavan, Bhasker Reddy, Harshada Samant, M. G. Santosh, Narendra Saini, Swapna Sharma, Siddharth Srivastava, Chetan Singh Solanki, Rajni Solanki, Vaidy Subramanian, Dessislav Valkanov, Vinodh Velusamy, Ma Zhe and many others;

— for the fun I had while working for Rose VZW: Sanchita Bandyopadhyay Ghosh, Uday Chitraghar, Subrata Bandhu Ghosh, Anil Kottantharayil, Reena Lasrado, Basavaraj Madivala, Anshu Mehta, Praveen Raghavan, Narendra Saini, Chetan Singh Solanki, and many others;

— for their help with the visa formalities when my parents visited me in Leuven: Pol Marchal and his parents;

— for their friendship over the years: Rohit Ananthakrishna, Vijay Arya, S. N. Sandesh Chakravarthy, P. B. Guruprasad, Rajiv S. Kumar, Narendra T. Manappa, Sanjay Nagaraj, Srinivasa G. Narasimhan, Niranjan Nayak, Vasudeva Nayak, Srivatsan Raghavan, Kiran B. R., Sudhir B. Sangappa, Mohammad Abu Sarwat, Bhupendra Singh, S. Srikanteshwara and all those that I am going to later regret for not remembering now;

— for their support and encouragement while completing this dissertation: Mr. N. H. Sathyaraja, Professor S. Ramesh, Dr. B. G. Prakash, and my colleagues at GM R&D;

— for some of the already mentioned reasons and all valid ones that still remain: *Amma*, *Anna*, Shubha, Amaresh, Vrushank, *Atte*, *Maava*, and my relatives;

— finally, for the wonderful things that lie beyond the realm of reasons and reasoning: Harshita Rao.

K. C. Shashidhar
Bengaluru, May 2008

Abstract

Thesis — *Automatic and efficient verification of loop and data-flow transformations commonly applied while optimizing digital signal processing and scientific computing programs is feasible by functional equivalence checking of the original and transformed programs.*

Application of transformations, in general, is known to enable efficient implementation of programs. For resource constrained embedded systems, however, transformations are essential to meet the stringent constraints on the power consumption and performance of the implementation. The choice of relevant transformations depends here on the target processor and memory architecture. Hence compilers are often not able to apply such transformations, leaving them to be applied either manually or using transformation tools. This necessitates verification of the correctness of the transformed program. At present, the verification is being done by simulation-based testing. But testing is very time-consuming, often inconclusive and calls for additional effort for debugging. To overcome these limitations, this dissertation presents a fully automatic and efficient functional equivalence checking method for formal verification of the transformed program against the original.

The presented equivalence checking method targets a class of programs and transformations that is common in the domain of digital signal processing and scientific computing applications. Most importantly, in these applications, program routines subject to transformations are typically array intensive, with piecewise affine expressions to index and bound the references to arrays, and have static control-flow. The method is based on a model that represents the data and operator dependencies between the elements of the output and the input variables relying on the single assignment form of the program. It is able to check equivalence of models of the original and the transformed programs that are related through a combination of global loop and data-flow transformations. Reasoning with value-based dependencies, it establishes equivalences between sets of values in corresponding points in the data-flow of the two programs. When the transformed program fails to pass the check, it generates feedback on the possible locations of errors in the program. A prototype implementation of the method demonstrates its efficiency on real-life program pairs.

Nederlandse Synopsis

Thesis — *Ontwerpers van digitale signaalverwerkings en/of wetenschappelijke toepassingen maken intensief gebruik van lus- en datastroomtransformaties om de performantie van hun computerprogramma's op te drijven. De controle op de correctheid van dergelijke transformaties is zeer tijdsrovend. In deze thesis stellen we daarom een methode voor om automatische en op een efficiënte wijze de functionele equivalentie tussen het originele en het getransformeerde programma na te gaan.*

Lus- and datastroomtransformaties worden veelvuldig gebruikt door ontwerpers van ingebedde systemen. Ze verminderen immers de benodigde geheugenruimte en laten toe om dezelfde performantie met een lagere processorsnelheid te halen.

In een typisch ontwerp, kiezen de designers op basis van de originele, niet geoptimaliseerde code een aantal mogelijke transformaties uit om hierop uit te voeren. De keuze van welke transformaties ze gebruiken, hangt in sterke mate af van de beoogde processor- en geheugenarchitectuur. Deze afhankelijkheid maakt het erg moeilijk voor hedendaagse compilers om de juiste transformaties automatisch uit te kiezen en uit te voeren. Dergelijke transformaties moeten daarom nog steeds manueel toepast worden, al dan niet met computerondersteuning. Dit houdt in dat het getransformeerde programma geverifiëerd moet worden. Hiervoor vertrouwen ontwerpers meestal op simulaties. Deze tijdsrovende simulatie-gebaseerde verficatie laat echter niet toe met zekerheid te garanderen dat het getransformeerde programma volledig foutvrij is. In deze verhandeling willen we net dit laatste probleem oplossen.

Deze verhandeling presenteert daarom een methode om de functionele equivalentie tussen twee programma's na te gaan. De methode richt zich tot programma's en transformaties in de context van digitale signaalverwerkings- en wetenschappelijke toepassingen. De procedures in deze klasse van programma's zijn erg "array-intensief", maken gebruik van stapsgewijze affiene expressies als index of bovengrens van referenties naar arrays, en hebben een statische controlestroom. Tenslotte benut de voorgestelde methode de enkelvoudige toekenningsvorm in deze programma's.

In onze methode stellen we het originele en getransformeerde program-

ma voor op basis van de data- en operatorafhankelijkheden tussen de elementen van de invoer- en uitvoersvariabelen. Vervolgens gaan we de functionele equivalentie na tussen de modellen van de originele code enerzijds en van de getransformeerde code anderzijds. Door gebruik te maken van waarde-gebaseerde afhankelijkheden kan de methode equivalentie nagaan tussen twee verzamelingen van waarden die overeenkomen met de punten uit de datastroom van de twee programma's in kwestie. Indien het getransformeerde programma niet equivalent blijkt te zijn met het originele programma, dan genereert onze methode een lijst van de locaties in het getransformeerde programma waar de potentiële fout zou kunnen zitten. De efficiëntie van de methode wordt geïllustreerd door middel van een prototype implementatie en de uitvoering ervan op verschillende realistische en veel-voorkomende programma's.

Contents

Acknowledgments **I**

Abstract **V**

Nederlandse Synopsis **VII**

Contents **IX**

1 A Transformation Verification Problem **1**
1.1 Introduction 1
1.2 Importance of Program Transformations 2
1.2.1 General Program Optimization 2
1.2.2 Programmable Embedded Systems Design 5
1.3 Need for Verification of Transformations 7
1.4 Verification by Equivalence Checking 7
1.5 Problem Context and Focus 9
1.5.1 Array-Intensive Sequential Programs 9
1.5.2 Global Loop and Data-Flow Transformations 10
1.5.3 Data Transfer and Storage Exploration 11
1.6 Solution Overview and Contributions 18
1.6.1 Verification Technique in Brief 18
1.6.2 Summary of Contributions 20
1.7 Dissertation Outline 21

2 Current Solutions and Limitations **25**
2.1 Introduction 25
2.2 Methods Not Focused on Loops and Arrays 27
2.2.1 Methods from Hardware Synthesis 27
2.2.2 Methods from Software Analysis 28
2.3 Methods Focused on Loops and Arrays 29

2.3.1 Translation Validation Approach 30
2.3.2 Fractal Symbolic Analysis 30
2.3.3 Equivalence Checking for SAREs 31
2.4 Prior Work at IMEC 32
2.4.1 Claesen's SFG-Tracing Method 32
2.4.2 Angelo's Theorem Prover based Method 33
2.4.3 Samsom's Loop Verification Method 34
2.4.4 Čupák's System-Level Verification Methodology . . 34
2.5 Summary 34

3 Class of $\mathscr{D}$ Programs 37
3.1 Introduction 37
3.2 Properties of $\mathscr{D}$ Programs 38
3.2.1 Static Control-Flow 38
3.2.2 Affine Indices and Bounds 39
3.2.3 Uniform recurrences 40
3.2.4 Single-Assignment Form 40
3.2.5 Valid Schedule 41
3.2.6 Other Properties 42
3.3 Justification for the $\mathscr{D}$ Class 42
3.4 Summary 44

4 Representation of $\mathscr{D}$ Programs 45
4.1 Introduction 45
4.2 Representation of Data Dependencies 48
4.3 Array Data Dependence Graphs (ADDGs) 53
4.4 Data Dependence Paths 55
4.5 Recurrences in Paths 59
4.6 Data Dependence Slices 64
4.7 Summary 68

5 Global Loop and Data-Flow Transformations 69
5.1 Introduction 69
5.2 Loop Transformations (LTs) 70
5.2.1 Slice Preserving LTs 71
5.2.2 Slice Repartitioning LTs 73
5.3 Data-Flow Transformations 76
5.3.1 Data-Reuse Transformations 80
5.3.2 Expression Propagations 81
5.3.3 Algebraic Transformations 82

5.4 Operations for ADDG Normalization 87
5.4.1 Internal Array Node Elimination 89
5.4.2 Flattening of an Associative Chain 91
5.5 Summary 94

6 Statement-Level Equivalence Checking Methods 97
6.1 Introduction 97
6.2 Verification of Only Loop Transformations 98
6.2.1 Constrained Expressions (CEs) 98
6.2.2 Sufficient Condition for Equivalence of CEs 102
6.2.3 Checking Equivalence of CEs 103
6.2.4 Limitations of Samsom's Method 105
6.3 Verification of Loop and Data-Reuse Transformations . . . 106
6.3.1 Preconditions as Tuples of Dependency Mappings . 107
6.3.2 Statement Matching and Elimination of Copies . . 110
6.3.3 Sufficient Condition for Equivalence 113
6.3.4 Equivalence Checking Method 115
6.3.5 Prototype Implementation and Experience 117
6.4 Limitations of Statement-Level Checking 117
6.5 Summary 118

7 Operator-Level Equivalence Checking Methods 119
7.1 Introduction 119
7.2 Verification of LTs and Expression Propagations 120
7.2.1 Sufficient Condition for Equivalence 124
7.2.2 Synchronized Traversal of Two ADDGs 130
7.2.3 Handling Recurrences in ADDGs 143
7.3 Verification of Loop and Data-Flow Transformations 152
7.3.1 Sufficient Condition for Equivalence 155
7.3.2 General Equivalence Checking Method 159
7.4 Limitations of our Operator-Level Checking 167
7.5 Summary 168

8 Features of the General Method 169
8.1 Introduction 169
8.2 Errors and their Diagnosis 169
8.2.1 Types of Detected Errors 170
8.2.2 Limits to Error Localization 172
8.3 Optimizations to Speed-up the Method 173
8.3.1 Tabling Mechanism 173

8.3.2 Reconvergent Paths . . . 173
8.3.3 Focused Checking . . . 174
8.4 Performance Analysis . . . 175
8.4.1 Complexity Analysis . . . 175
8.4.2 Experimental Analysis . . . 175
8.5 Summary . . . 176

9 Method in Practice 177
9.1 Introduction . . . 177
9.2 Pre-processing the Source Code . . . 179
9.2.1 Selective Function-Inlining . . . 180
9.2.2 If-Conversion . . . 180
9.2.3 DSA-Conversion . . . 182
9.2.4 DEF-USE Checking . . . 185
9.3 Case Studies . . . 185
9.3.1 Implementation Characteristics . . . 185
9.3.2 Application Characteristics . . . 186
9.3.3 Verification Characteristics . . . 186
9.4 Summary . . . 186

10 Conclusions and Future Work 189
10.1 Summary and Contributions . . . 189
10.2 Directions for Future Research . . . 191

A Definitions of the Geometric Operations 195

References 197

List of Publications 209

Curriculum Vitæ 213

Chapter 1

A Transformation Verification Problem

1.1 Introduction

This dissertation presents an approach to solve a verification problem that is commonly faced in practice while designing cost-constrained programmable embedded systems. The problem arises because there exists a strong need, of increasing importance, for application of program transformations on an initial implementation of an algorithm.

Application of transformations, though common in general programming practice, takes a central place in the design activity in application domains like, for example, mobile and scientific computing (Section 1.2). In practice, system design activity is often separated into two main concerns, viz., synthesis and verification, leading to two threads of activities that are equally important. Therefore, whenever the need for application of transformations arises during synthesis, the need for verification of the applied transformations arises automatically (Section 1.3). This need presents a problem, *a transformation verification problem*, that this dissertation addresses. Many different approaches seem to lead to a possible solution to this problem. Among them, the equivalence checking approach seems to be the most promising, and moreover, it has been an extremely successful approach to address verification problems associated with hardware synthesis at the lower ab-

straction levels (Section 1.4). Tempting as the approach is, it cannot, in theory, be employed to solve the present problem in full generality. The reason being that it cannot be fully automated, but would require user interaction. An approach that is amenable for full automation, therefore, calls for a clear problem focus in terms of both the class of programs and transformations that can be handled (Section 1.5). But given the problem context, the focus should be such that a solution to it should still be relevant in practice. This dissertation presents a solution to such a focused, yet very relevant, subset of the complete transformation verification problem and contributes toward improving the system designer's productivity (Section 1.6).

1.2 Importance of Program Transformations

A program transformation refers to the act of transforming a program from one version to another version. Program transformations are employed in various software engineering processes like synthesis and optimization, maintenance and debugging, etc. In this dissertation, our concern on program transformations is with respect to their increasing importance in program optimization, in general (Section 1.2.1) and programmable embedded systems design (Section 1.2.2), in particular.

1.2.1 General Program Optimization

Ever since we began to program in high-level languages and use compilers to translate source code into machine code, the concern of efficiency of the latter has constantly stayed with us. This concern has been addressed, from almost the very first compilers, by including a process during compilation that strives to improve the efficiency of the program being compiled. This process is implemented in a stage called the *code optimizer* in a typical compiler. The code optimizer analyzes the program for possible improvements and applies suitable optimizing transformations on the program that can improve its efficiency (Aho et al. 1986). The improvements that can possibly be achieved by an optimizer depend on three factors: (1) the nature of the programming language in which the source code has been written, (2) the nature of the architecture of the machine for which the code has to be generated; and (3) the

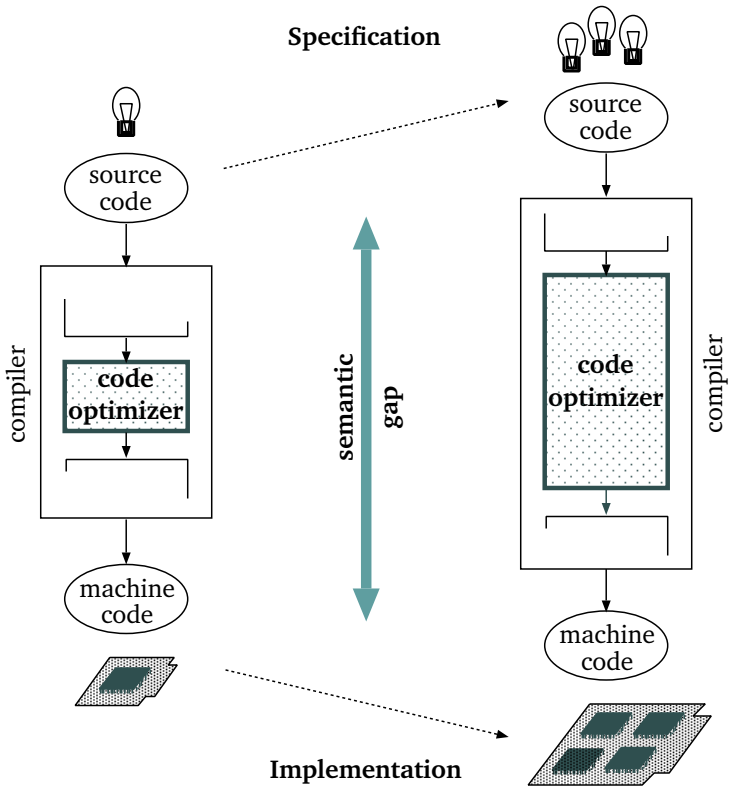


Figure 1.1: The trends in computing imply a greater need for program transformations to map more complex specifications onto more complex machines.

power of the analyses and the transformations that are at the optimizer's disposal. The evolution of computing has had a strong influence on all three of these factors.

On the one hand, with the increase in the complexity of the applications to be programmed, the trend has steadily been toward programming at higher abstraction levels. This has helped programs to be closer to the application related problem domain and has hence significantly freed the burden on the programmer from concern for issues not related to the problem and its conception itself. On the other hand, the vast advances in the semiconductor technologies has made it possible to have not only more resources on the target architecture, but to have designs that are

increasingly sophisticated in their complexity. For example, multi-core processors with highly pipelined architectures and multi-level caches are now quite common.

The above two trends have led to a divergence between the software and the hardware in terms of the semantics of the primitives at the two levels. This has resulted in the so-called *semantic gap* between the two that is ever widening, as shown in Figure 1.1. The burden of bridging this semantic gap falls on the compiler, in general, and its code optimizer, in particular. Therefore, the software and hardware design practice has a direct bearing on the third factor, namely that of the power of analyses and transformations of the code optimizer (Adve et al. 1997). The greater the expressive power of a programming language, the harder it gets to analyze the program. This implies less available information to apply suitable transformations. Also, the greater the possibilities in the dynamic configuration of a hardware, the harder it gets to reach the optimal points in the design space via transformations.

This has meant that the importance of program transformations, though long recognized for high-performance applications (Bacon et al. 1994), is becoming essential even for general purpose computing (Muchnick 1997). For instance, lately multi-core processors are common even for general purpose systems. Exploiting them requires transformations that expose parallelism in the original program. Many of these transformations are best applied in a pre-compilation phase at the software architecture level. But it is clear that for the envisioned future, given the fundamental incompleteness of transformational design systems in general (Voeten 2001), the automated steering techniques will still have to be complemented with manual guidance.

Furthermore, the future of general purpose computing itself is heading toward mobile, wearable computing, shifting the optimization needs to a newer level, where as we discuss in the following section, application of program transformations are believed to be *sine qua non* for implementing a system that meets the functional and non-functional requirements of the specification.

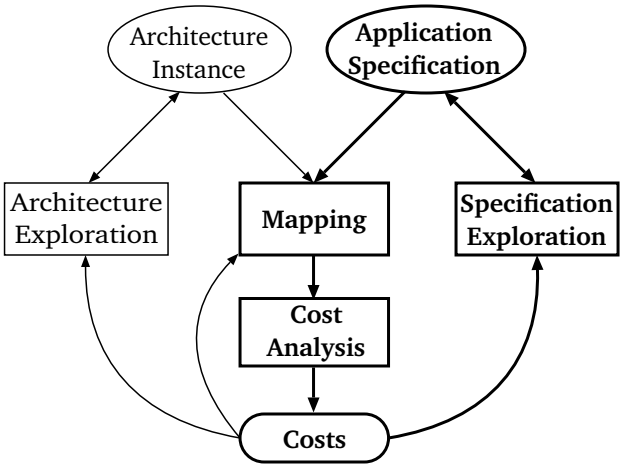


Figure 1.2: Cost-aware design of programmable embedded systems: The Y-chart approach (Kienhuis et al. 2002).

1.2.2 Programmable Embedded Systems Design

The design of programmable embedded systems for the consumer electronics market, particularly for multimedia and telecom signal processing applications, is a complex task. The software and hardware design trends that we discussed in the previous section apply here too. But the demands on the optimality of design of these systems in terms of performance, area, power and cost are extremely high. Traditional, business as usual, system design methodologies borrowed from other domains of computing are not able to cope with the design space and time-to-market challenges as presented in this domain. This has motivated research and development of frameworks for systematic design of embedded systems. The frameworks call for design exploration and optimization at different levels of abstractions, to arrive at a mapping of the software onto the custom made platform, which is as close to the optimal implementation as possible.

Most of the design frameworks have a commonality that is nicely captured by the so-called Y-chart (Kienhuis et al. 2002). This is shown in the Figure 1.2. At any given abstraction level, high-level cost estimates are used to explore the design space of the architecture, the application, and the mapping between them. We are concerned with the iterative de-

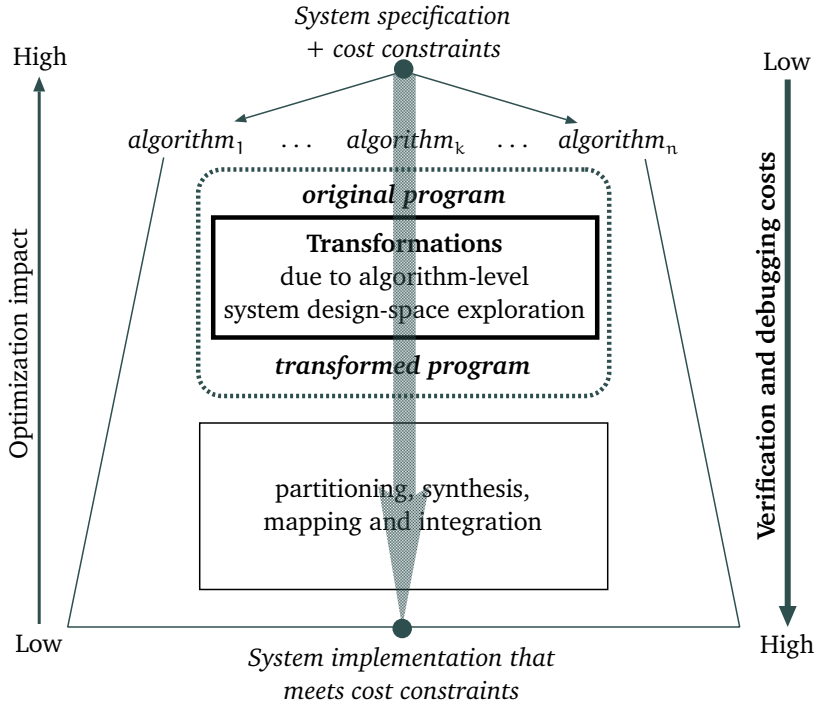


Figure 1.3: Program transformations in embedded systems design.

sign cycle that involves the exploration of the functional specification, the portion of the Figure 1.2 shown in bold. An important design rule is that optimizations applied at higher abstraction levels offer greater gains. Therefore, the initial source code, called the *executable specification*, is the starting point for a systematic exploration which subjects it to source-to-source transformations (Catthoor et al. 1998b; Catthoor et al. 2002; Leupers 2002; Brandolese et al. 2002; Wolf and Kandemir 2003). The exploration successively refines the mapping of the specification with respect to the platform architecture, that is also being orthogonally explored. Figure 1.3 shows the schematic of such a design exploration flow that is driven by program transformations.

1.3 Need for Verification of Transformations

As far as the general program optimization is concerned, the increased need for program transformations has resulted in highly complex compilers that can no longer be trusted to produce correct code (Boyle et al. 1999). In fact many compiler manuals come with the warning that users enable optimizations provided they do so at their own risk. It has therefore become increasingly necessary to verify the correctness of the analyses the compilers use, and also their implementations.

In the context of embedded systems design, the need is even more obvious due to the so-called *designer-in-the-loop* design space exploration. Here, the designer's understanding of the characteristics of the application and the hardware architecture that is drawn from previous design experience is crucial in order to make a good choice from the large space of valid designs. The systematic nature of the design methodologies aside, the problem is in the lack of tools to aid the designer in applying the refinements and transformations. It is often much easier for the designer to apply the transformations manually. Therefore, similar to the need for verification in the context of high-level hardware synthesis, the need for verification of transformations is very strong in system-level design. For instance, an exercise recording the effort spent in man-months applying a design methodology in practice showed close to 50% of the total time spent in verification related activities (Catthoor and Vandecappelle 2000). It is important to note that a postponement of the verification task to later stages in the design activity incurs a huge penalty. Indeed, any errors introduced by the program transformations proliferate as the design passes through refinements at each stage of the design flow and the verification and debugging tasks become costlier.

1.4 Verification by Equivalence Checking

Given the increasing need for a posteriori verification of program transformations, the question arises as to how to address the need. The obvious and the most commonly prevalent a posteriori approach in practice is to adopt simulation-based testing. The original and the transformed programs are both given the same input values and it is checked whether they both output identical values at the end of their execution. But

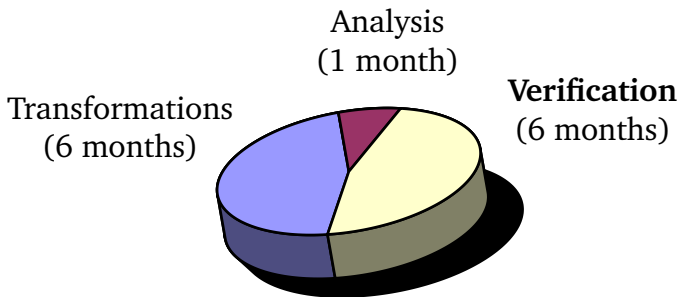


Figure 1.4: Record of effort spent in man-months applying DTSE methodology on MPEG-4 application.

then, matching outputs ensure that they are functionally identical only for the input values they were executed upon. Therefore, the process is repeated many times with different inputs until sufficient confidence is gained in the functional equivalence of the transformed program and the original program. Verification by simulation-based testing approach, though easier to deploy in practice, is typically inconclusive. It does not guarantee that the transformed program is functionally equivalent to the original for *all* inputs, unless it is tested for all possible input values, which is very time-consuming and often infeasible. Moreover, in the event that testing shows that the transformed program is in error, it is hard to determine the source of the error, unless the program is well instrumented. Therefore, verification by simulation-based testing approach has been largely unsatisfactory in practice and there is a strong need for an alternative verification approach.

As mentioned in the previous section, program transformations rely on analyses of the programs. The transformations that are applied at compile-time commonly rely on *static* program analyses that reason, *formally*, on various aspects of the possible run-time behaviors of the program. Here the input to the analyzer is just the text of the program being analyzed. It does not reason about the behavior of the program for a *particular* input, but for the entire space of inputs as a *whole*. This has the advantage that the information gathered from such a reasoning is valid for any possible run of the program. This advantage of the static analyses approach has made it attractive not only for program transformation purposes, but also for verification purposes. In the lit-

erature, verification by this approach is also called *formal verification* to distinguish it from the simulation-based approach. For the verification problem at hand, it is desirable to ensure the correctness of the transformation by formally and automatically verifying the functional equivalence of the original and the transformed programs. In other words, by *equivalence checking*.

Equivalence checking has had enormous success in the hardware synthesis domain. In fact, it is by far the most successful of formal verification methods in practice. In hardware synthesis, transformations are applied on design objects like finite state machines (FSMs) and Boolean functions. In theory, the analyses required to show equivalence of such objects are simpler and fully automatable. But in software synthesis, or in general, in system synthesis, the design objects of concern are programs that are far more expressive than FSMs or Boolean functions. Unfortunately, in theory, the analyses required to show equivalence of general programs cannot be automated. However, it is possible to have a fully automatable equivalence checking solution when either the programs belong to certain classes and/or the transformations applied on them belong to certain classes. Therefore, any hope for verification of transformation by equivalence checking has to inevitably rely on the existence of such classes in the problem at hand. As far as our transformation verification problem is concerned, it is indeed the case.

1.5 Problem Context and Focus

In this section, we discuss the problem context at hand, by providing an outline of the nature of the programs of our interest (Section 1.5.1), the transformations of our interest (Section 1.5.2) and an example design methodology that motivates a solution to the problem (Section 1.5.3).

1.5.1 Array-Intensive Sequential Programs

Signal processing and scientific applications are data dominated. The programs that do the most work in these applications are mainly the numerical routines. It is the optimization of these routines that has the highest impact on the overall efficiency of the system. Therefore, often the optimization effort is particularly focused on optimizing these

routines (Allen and Kennedy 2001).

There are certain properties of the programs that implement these routines that make them amenable for certain static analyses. These programs are typically implemented in a structured, imperative programming language. They are also sequential, that is, they have a single thread of control. Given that they implement numerical routines, arrays serve as the pre-dominant data-structures in the programs. Also, the data-dependent control-flow in these programs is minimal. Analysis methods for optimization of these programs also rely on the fact that, typically expressions in the subscripts of arrays and loop bounds of these programs are affine.

Our interest is in the verification of source-to-source transformations applied on the programs that belong to the above described general class, that is referred to as *array-intensive programs*.

1.5.2 Global Loop and Data-Flow Transformations

Source code transformations that reduce the accesses to the instruction and data memory hierarchy have become increasingly important to apply. This is due to the fact that reduction of accesses play a crucial role in significantly lessening the influence of the gap between the operation speeds of processors and memories.

Broadly, there are two kinds of such transformations viz., *global loop transformations* and *global data-flow transformations*. Global loop transformations are applied to reorder and restructure the `for`-loops in the complete program in order to minimize the data transfers between different layers of the hierarchy by improving the temporal and spatial locality of the accessed data. On the other hand, global data-flow transformations are applied either to remove repeated computation or to break bottlenecks caused by data-flow dependencies in the program. They comprise of *expression propagations* that introduce or eliminate temporary variables that hold intermediate values and *global algebraic transformations* that take advantage of algebraic properties of the operators in transforming the data-flow.

We are concerned with the verification of these two categories of transformations. The need for verification support for these transformations is rather high because they invariably involve error prone manipulation

of the index expressions of the array variables, especially when applied manually.

1.5.3 Data Transfer and Storage Exploration

The goal of the Data Transfer and Storage Exploration (DTSE) methodology developed at IMEC, Belgium, is to determine an optimal execution order for the background data transfers together with an optimal background memory architecture for storing the array data of the given application. The complete DTSE methodology is described in detail by Catthoor et al. (1998b) for customized architectures and by Catthoor et al. (2002) for programmable architectures. A global overview of the DTSE steps is as shown in Figure 1.5.

The steps in the methodology are applied successively starting from an executable system specification with accesses to multi-dimensional array signals. The output is a transformed source code specification, combined with a full or at least partial netlist of memories. The latter serves as the input for customizing (or configuring) the memories during final platform architecture design and integration. The transformed source code is input for the software compilation stage in the case of instruction-set processors.

The flow is based on the idea of orthogonalization (Catthoor and Brockmeyer 2000), where in each step a problem is solved at a certain level of abstraction. The consequences of the decisions are propagated to the next steps and as such decreases the search space of each of the next steps. The order of the steps ensures that the most important decisions and the decisions that do not put too many restrictions on the other steps are taken earlier.

The methodology, as mentioned, advocates a systematic exploration of the design-space for a close to optimal implementation that is for a large part based on program transformations. As a consequence of this, in order to ensure correctness of the design that is evolving, automatic transformation verification support has been recognized as very important in order to deploy the methodology in practice. The Figure 1.4 presented earlier is but one example that illustrates its importance. In what follows, we provide a brief summary of its main steps and note the transformations for which there is a strong need for verification sup-

port and where our method is able to help. A reader who finds that the problem that we address is sufficiently motivated and is clear about the general context in which it arises can proceed to Section 1.6.

Platform independent steps

The first steps are platform independent. They optimize the data flow, the regularity and locality of data accesses in general, and make the data reuse possibilities explicit.

1. **Pre-processing**

   This step consists of a pruning and a scenario creation sub-step and precedes the actual DTSE optimizations. It is intended to isolate the data-dominant code which is relevant for DTSE, and to present this code in a way which is optimally suited for transformations (Catthoor et al. 1998b). All the freedom is exposed explicitly, and the complexity of the exploration is reduced by hiding constructs that are not relevant for DTSE. A remark here is that, verification of transformations employs static code analyses, just as transformations employ the analyses to identify opportunities for optimization. Therefore, if the code is pre-processed to make it amenable for code analysis, it not only aids application of transformations, but also, or even more importantly, aids the verification task. In fact, the verification method that we develop derives its main strengths from the pre-processing steps, especially the ones that have been automated.

   In the pruning sub-step the specification is first partitioned into a three-layered hierarchy. All the dynamic control-flow in the application is grouped in the top layer, the memory accesses schema of the algorithmic kernels are grouped in the middle layer and all the arithmetic and logic functions are grouped in the bottom layer. The DTSE optimizations then focus exclusively on the middle layer as shown in Figure 1.6. Additionally, during pruning, the pointer accesses in the middle layer are converted to array accesses (Séméria and Micheli 1998; van Engelen and Gallivan 2001; Franke and O'Boyle 2003), other constructs that cannot be modeled by the geometrical model are encapsulated and moved to the bottom layer (Palkovic et al. 2004), functions are selectively inlined (Absar et al. 2005) and the code is rewritten into

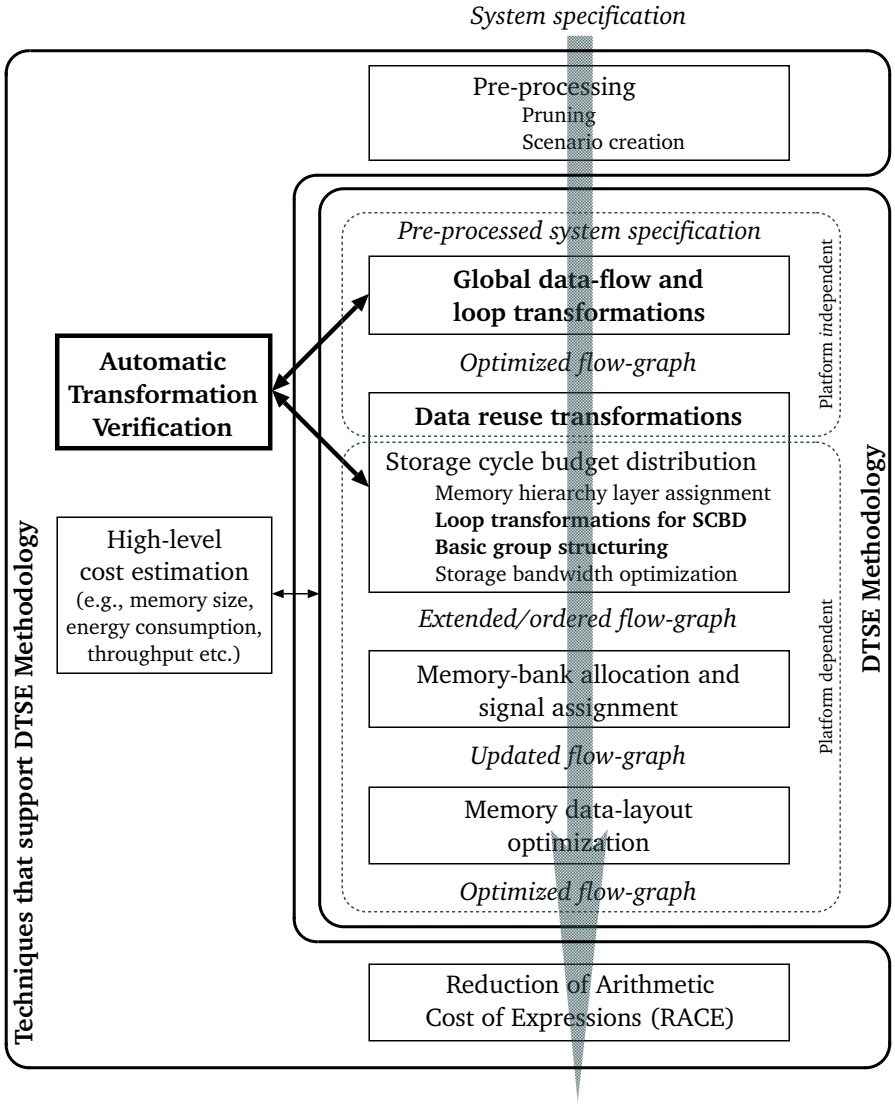


Figure 1.5: Global overview of the steps in the DTSE methodology and automatic transformation verification's place in the methodology.

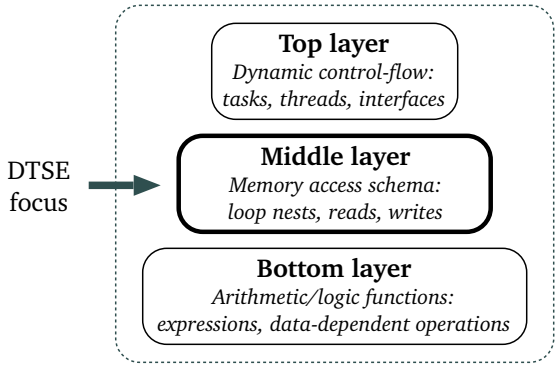


Figure 1.6: Focus of Data Transfer and Storage Exploration (DTSE) methodology.

dynamic single-assignment form (DSA), wherein every memory element is written only once during execution (Feautrier 1988; Vanbroekhoven et al. 2005b). Although DSA is not a strict requirement for DTSE transformations, it does increase the freedom potentially allowing better transformations to be performed. The scenario creation sub-step deals with the data-dependent control-flow in the code that stands in the way of using many of the static analyses of the subsequent steps. It results in separation of versions of codes that have static control-flow for each of the common execution scenarios as identified via profiling (Palkovic et al. 2005).

2. **Global data-flow transformations**

   The set of system-level data-flow transformations that have the most crucial effect on the system exploration decisions has been classified, and a systematic methodology has been developed for applying them (Catthoor et al. 1996; Catthoor et al. 1998a). Two main categories exist. The first one directly optimizes the important DTSE cost factors by removing redundant accesses and reducing intermediate buffer sizes. The second category serves as an enabling transformation for the subsequent steps because it removes the data-flow bottlenecks wherever required, especially for the global loop transformations step. At present, these transformations are predominantly applied manually and the need for

verification of the transformed version of the code is strong here.

3. **Global loop transformations**

   The transformations in this step of the script aim at improving the data access regularity and locality for multi-dimensional array signals and at removing the system-level buffers introduced due to mismatches in production and consumption ordering (van Swaaij 1992; Danckaert 2001). They are applied globally across the full code, not only individual loop nests, but also across function scopes because of the selective inlining applied in the preprocessing step. This step, to a large extent, has been implemented in a recent prototype tool (Verdoolaege 2005). However, in some cases, as we motivated earlier, the required loop transformations are obvious for the designer and manual application of the transformations are then preferred over the use of a loop transformation tool. This, along with the earlier motivating arguments, implies the need for automatic a posteriori verification of transformations.

4. **Data reuse exploration**

   In this step the data locality introduced by the previous global loop transformation step is exploited. Data reuse possibilities are made explicit by analyzing virtual multi-copy hierarchies (including bypasses) for the trade-off of power and memory size cost. Heavily reused data is copied to smaller power-efficient on-chip memories, while costly accesses to external memory are reduced (Wuytack 1998; Van Achteren 2004; Issenin et al. 2004). This step has been automated partially, but at present in many cases it is being applied manually and thereby requiring verification support.

Platform dependent steps

The steps that follow are platform dependent. Here physical properties of the target background memory architecture are taken into account to map and schedule the data transfers in a cost-efficient way.

1. **Storage cycle bandwidth distribution (SCBD)**

   This step mainly determines the bandwidth/latency requirements and the balancing of the available cycle budget over the different memory accesses. It has four main sub-steps.

During memory hierarchy layer assignment (Brockmeyer et al. 2003), the data reuse copy hierarchies and the corresponding transfers are partitioned over several hierarchical memory layers and the class of each layer is determined (e.g., ROM, SRAM, DRAM,...).

Additional loop transformations are performed to meet the real-time constraints, such as merging of loops without dependencies, software pipelining and partial loop unrolling (Vandecappelle et al. 2003; Shashidhar et al. 2001). These normally do not influence the access order of data elements, so also the data reuse behavior remains the same. Unlike during the global loop transformation, verification support is needed here as these loop transformations are at present applied manually.

The initial data types (arrays or sets) are grouped/partitioned in basic groups in a sub-step called *basic-group structuring*. These transformations need verification support as well and the method we present is applicable as long as the transformations here are applied on statically allocated arrays.

Storage bandwidth optimization (SBO) performs a partial ordering of the flow graph at the basic group level. It tries to minimize the required memory bandwidth for a given cycle budget. This step produces a conflict graph that expresses which basic groups are accessed simultaneously and therefore have to be assigned to different memories or different ports of a multi-port memory (Wuytack 1998; Omnès 2001).

2. **Memory/bank allocation and signal assignment (MAA)**

   The goal in the memory/bank allocation and signal-to-memory or bank assignment step (MAA) is to allocate memory units and ports (including their types) from a memory library and to assign the data to the best suited memory units, given the cycle budget and other timing constraints (Balasa et al. 1997).

3. **Memory data layout optimization**

   In the memory allocation and signal-to-memory assignment step, signals were assigned to physical memories or to banks within predefined memories. However, the signals are still represented by multi-dimensional arrays, while the memory itself knows only addresses. In other words, the physical address for every signal element still has to be determined. This transformation is the data

layout decision. This involves several sub-steps and focuses both on the cache(s) and the main memory. For hardware-controlled caches advanced memory layout organization techniques have been developed, which allow removal of most of the conflict misses due to the limited cache associativity (Kulkarni 2001).

Techniques that Support the Core DTSE Methodology

The steps in the DTSE methodology call upon the supporting techniques discussed in the following.

- **High-level memory size estimation**

  The memory data layout optimization (see above) is the last step in the DTSE methodology and determines the overall needed memory size of the application. However, in the earlier DTSE steps the final execution order of the memory accesses is not yet fixed. This stage provides lower and upper bounds for the needed memory size for a partially defined loop organization and order (Kjeldsberg 2001).

- **Reduction of arithmetic cost of expressions (RACE)**

  This stage is also not part of the DTSE methodology itself, but is vital to deal with the addressing and control-flow overhead that is introduced by the DTSE steps. The methodology to deal with this overhead is incorporated into another system design stage developed at IMEC, namely the RACE project, previously known as the ADOPT (ADdress OPTimization) project (Miranda et al. 1998).

- **Formal verification techniques for system-level transformations**

  This stage is the main motivation for our research. Most of the steps in the DTSE methodology have not been automated, let alone in mature, verified tools. While applying DTSE methodology in design projects simulation based testing has been used until now, but it is extremely time consuming and unsatisfactory. The goal of this stage is to provide a fully automatic technique to verify the transformations that can also help diagnose erroneously applied transformations.

  Verification support for the most important of the DTSE transformations, namely, the loop transformations has been presented

by Samsom (1995). A technique to make this approach feasible for large problem instances, called incremental dimension handling, has been presented by Čupák (1998) and he has shown how to integrate the technique into the standard verification flow for behavioral-level hardware synthesis. Our research on the problem has led to the development of a single unified method that can verify the global loop and data-flow transformations and also provide useful error diagnostics for debugging (Shashidhar et al. 2005d). We will discuss our contributions in detail in the following section.

1.6 Solution Overview and Contributions

In this section, we present our solution to the stated transformation verification problem (Section 1.6.1), followed by our contributions (Section 1.6.2).

1.6.1 Verification Technique in Brief

For the class of programs and transformations that lie in our focus as discussed in Section 1.5, we have developed a transformation verification technique that is based on an equivalence checking method that takes the original and the transformed programs as input and checks whether the two programs are functionally input-output equivalent. In the event that the checker fails to prove the equivalence, it generates error diagnostics that localize the error in the transformed program and hence help in debugging. The scheme is as outlined in Figure 1.7. The technique does not distinguish between the transformations as long as they are only from the categories of global loop and data-flow transformations. The transformed program can be under any combination of the transformations. The verification is *oblivious* of any information about the particular instances of the above transformations that were applied and the order of their application. Therefore, the designer does not need to provide any additional information apart from the original and the transformed programs.

The equivalence checking method that our technique relies on is based on reasoning on the data dependencies in the two programs. It uses a program representation called an *a*rray data dependency graph (ADDG)

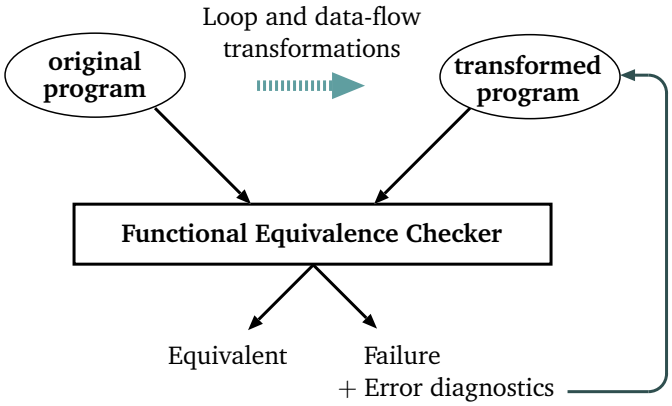


Figure 1.7: Outline of our verification technique.

that captures all the data dependencies in a program at the level of operators in its statements. The information about data dependencies is represented in terms of polyhedral domains and mappings in closed form. Based on a synchronized traversal on the ADDGs of the original and the transformed programs, the method checks that for each pair of corresponding paths in the computation between the observable variables, both programs have:

1. identical sequences of operators; and
2. identical data dependencies.

This is a sufficient condition for equivalence and the ADDGs can be suitably normalized under the considered transformations to check this condition. The traversal starts from each pair of the corresponding outputs and proceeds to the inputs by accounting for identical operators for corresponding paths and updating the data dependency relationship from the output. To address algebraic transformations, when operators with known algebraic properties (e.g., associativity, commutativity, ...) are reached, the checking normalizes the ADDGs and establishes corresponding paths to proceed with the traversal. Whenever the traversal reaches inputs in both programs, the data dependency relationship between the output and the input arrays for the corresponding paths are checked for identicality. In the event that the checking fails, the method

has enough book-keeping information available to reasonably deduce the location of errors in the text of the transformed program.

The technique has been implemented in a prototype tool and integrated in a more general framework which contains other prototype tools for the pre-processing steps shown in Figure 1.5. It has been used to verify original and transformed program pairs taken from design practice. The verification has been possible, as desired, in a push-the-button style and typically requires only a few seconds. It has also demonstrated its ability to generate useful error diagnostics. In two separate instances, the errors reported by the tool were traced to bugs in separate prototype code transformation tools.

1.6.2 Summary of Contributions

We present the contributions of our work in chronological order.

Our work began with an instance of an actual **case-study of a turbo-decoder**, a component of a telecom application, that involved application of program transformations (Shashidhar et al. 2001; Vandecappelle et al. 2003). This exercise provided us with insights into the nature of the verification problem faced by the designers in industrial practice.

The search for a general solution to the transformation verification problem has begun with the initial extension of a prior method that could handle only loop transformations to also handle **data-reuse transformations** in a combined way (Shashidhar et al. 2002b; Shashidhar et al. 2003b).

We have then developed a general program representation that allowed us to generalize data-reuse transformations to **handle expression propagations** combined with loop transformations (Shashidhar et al. 2003a).

Following this, we have proposed demand driven normalizations and extended the method to also **handle algebraic transformations** (Shashidhar et al. 2005a).

We have subsequently **generalized the method** and shown how the method can **handle recurrences** in the data-flow (Shashidhar et al. 2005d).

In validating our work with programs from real applications, we have

contributed in defining and composing the chain of code pre-processing tools in cooperation with other members of the DTSE PhD team.

1.7 Dissertation Outline

This dissertation has nine chapters to follow, not all of which are of the same length and detail. Each chapter begins with an introduction that outlines the material that it covers and ends with a summary of the discussion. An outline of the content of each of the chapters is as below.

Chapter 2 starts with an overview of formal verification and broadly distinguishes the related work in literature into a priori and a posteriori approaches. A discussion of the pros and cons of the two approaches motivates the suitability of equivalence checking for addressing the transformation verification problem. The closely related techniques in the literature are discussed by classifying them into partially automatic and fully automatic techniques. Following this, we recall prior work at IMEC that has influenced our work.

Chapter 3 characterizes the programs that are handled by our method, called the class of $\mathscr{D}$ programs. The important properties of the class, viz., static control-flow, affine indices and bounds, uniform recurrences, single-assignment form and existence of a valid schedule are discussed in successive sections. This is followed by a discussion on the justification for the choice of this class.

Chapter 4 discusses a representation for the $\mathscr{D}$ programs. Representation of data dependencies in a statement in the program are discussed first, followed by a whole program representation called an array data dependence graph (ADDG) that glues them together. The notions of data dependence paths, recurrences and slices that our method calls upon are then discussed in successive sections.

Chapter 5 classifies the categories of transformations that our method can verify. The discussion of each transformation is followed by noting its effect on the ADDG representation and the invariant

property that is preserved under the transformation. Two operations for local normalization of an ADDG are then discussed. They are used on a demand-driven basis by our method.

Chapter 6 discusses equivalence checking methods that are restricted to the statement-level scope. The method for verification of loop transformations developed by Samsom is recalled first, followed by our extension of the method that can handle both loop and data-reuse transformations. Prior to presentation of each method, the choice of representation, its ramifications and sufficient condition for equivalence are detailed. A discussion of the limitations of the statement-level equivalence checking is provided next.

Chapter 7 presents the equivalence checking methods that function at the operator-level scope. We first discuss a method that can handle loop and expression propagation transformations and later extend the method to also handle algebraic transformations. The sufficient condition in terms of the data dependence slices is discussed, followed by a central scheme of synchronized traversal of the ADDGs of the original and the transformed programs. The chapter concludes with a discussion of limitations of our general method.

Chapter 8 discusses the implementation aspects of our method and the features that come with it automatically. We first discuss the computational complexity of the method. We then illustrate some of the features of our representation that provide opportunities for optimization of our method. This is followed by a discussion of the ability of the method in providing diagnostics when it fails to prove equivalence. Examples are provided to illustrate its ability and the limits that exist at present in its error localization.

Chapter 9 presents a demonstration of our prototype implementation of the method on some real-life applications. In order to deal with the limitations of the program class that can be handled by our method, the implementation relies on a chain of automatic code pre-processing tools. We first discuss this tool chain, followed by a discussion of the application case studies that provide the characteristics of the applications considered and the characteristics of the use of our verification method on them.

Chapter 10 concludes the dissertation with a summary of our work, the contributions thereof, and possible directions for future research on the transformation verification problem.

Chapter 2

Current Solutions and Limitations

2.1 Introduction

In this chapter, we discuss the solutions that are on offer in the literature for the transformation verification problem and their current limitations. In order to focus the discussion to our problem context, we very broadly classify the correctness questions that are typically posed as in Figure 2.1. On the left-hand top corner is a *source program* and on the right-hand bottom corner is the *target code* that performs well on the intended hardware architecture. Conceptually, the journey taken by the source program is divided into a horizontal segment, to represent source code level *transformations*, and a vertical segment, to represent the translations from the source language to the target language program. In order to ensure that the *transformed source program* in the right-hand top corner is correct with respect to the source program, either of the following questions can be posed:

- Is the transformation tool correct?, or
- Is the transformed source program a correct transformation of the source program?

Addressing the first question here asks for an *a priori* solution. As motivated in the previous chapter, it is not suitable in our context because of-

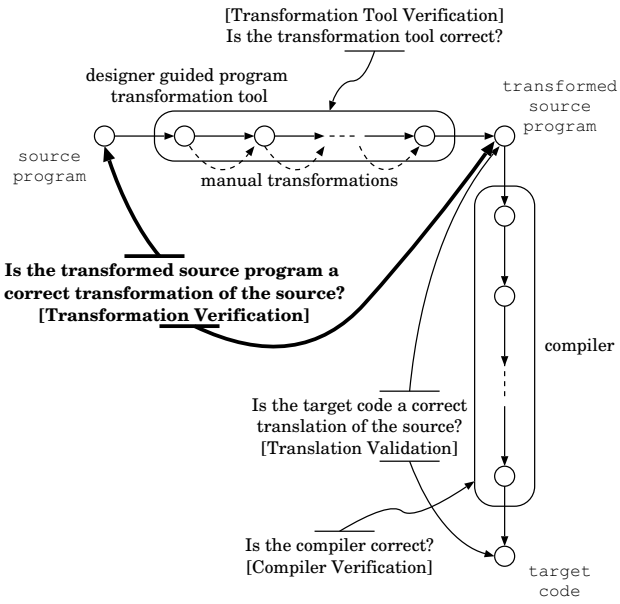


Figure 2.1: Contrasting with related work

ten a stable, flexible code transformation tool that can satisfy the needs of a typical designer does not exist. Therefore, our discussion discounts the rich literature on *a priori* transformation verification methods that either propose a *correct-by-construction* approach or a full-fledged formal verification of the code transformation tool. In our context, it is the second question that is required to be addressed with an *a posteriori* verification solution. This dissertation presents a solution that shows how the question can be answered for the class of programs and transformations that arise in the application domain of our interest.

Once the transformed source program is obtained, the compiler takes over the task of translating it to the target code. In order to ensure that the target code is correct with respect to the (transformed) source program, again, either of the following questions can be posed:

- Is the compiler correct?, or
- Is the target code a correct translation of the source program?

It is obvious that, except for the distinction that we are making between transformations and translations for the sake of convenience, the ques-

tions are similar to those already discussed. The first question has been examined as the compiler verification problem, for example (Müller-Olm 1997; Goos and Zimmermann 1999), and again, we discount them from our discussion. The second question that calls for an *a posteriori* solution is closely related to the problem we address to be relevant for our discussion.

The chapter is organized as follows. In Section 2.2, we discuss methods that address this question, but are not focused on programs with loops and arrays. In Section 2.3, we discuss methods that are closely related to our work. In Section 2.4, we recall the prior work of our colleagues at IMEC that addressed the initial concerns of formal correctness in our context. Finally, we conclude in the last section by enumerating the distinguishing requirements that are on the solution that we propose in this dissertation.

2.2 Methods Not Focused on Loops and Arrays

In this section, we discuss a posteriori verification methods that do not exactly match our problem context, but are nevertheless interesting to note. The discussion is split into methods that come from the hardware synthesis domain and those that come from the software analysis domain.

2.2.1 Methods from Hardware Synthesis

In the context of verification, the term *functional equivalence checking* is most commonly used in the hardware synthesis domain. From the very beginning of the digital circuit design practice, a common problem faced by designers has been to verify whether an optimized circuit is functionally the same as the original circuit. The infeasibility of testing and the forbidding cost of hardware design cycle immediately motivated the development of equivalence checkers for digital circuits. The designer could supply these checkers with the reference (original) circuit and the optimized (transformed) circuit, and push a button to verify their functional equivalence.

With the evolution in the applications, the circuits have gained an enor-

mous complexity, and the design practice has shifted from circuits to higher level representations of circuits in order to deal with the complexity of the designs. This has in turn meant that equivalence checkers also scale in their ability, not only to handle larger problem sizes, but also directly reason at the higher level representations. Also, as the design has moved to higher level abstractions, the need for guaranteeing certain safety and liveness properties in the design has become important, thereby requiring, not only equivalence checkers, but also, *property checkers*. But using property checkers requires skill and their use is limited to the expert designers, unlike equivalence checkers which do not require special skills and can be used by non-expert engineers too. The cycle of evolution in hardware design style and the pressure on verification to respond to the challenge has fueled an intense research and development activity on formal verification methods and tools for hardware synthesis (Kern and Greenstreet 1999; Kurshan et al. 2002; Prasad et al. 2005).

A great variety of methods exist in formal hardware verification in which the underlying engine is essentially an equivalence checker. The distinction between checkers come from the level of abstraction in the data-type and the circuit-type that is handled. The data-type abstraction ranges from bit-level to word-level and beyond, whereas the circuit-type abstraction ranges from combinational-level to sequential-level and beyond. The methods have reached maturity at the lower levels and the current challenge is at the word-level checking for behavioral specifications written in hardware description languages.

In the transformation verification problem that we are facing, the data-type level is that of the array data-structure and the quest is for an equivalence checking solution for algorithm-level specifications with arrays as data-types. Clearly, as it follows from the above discussion, this is beyond the space of current activity on equivalence checking in the hardware synthesis domain.

2.2.2 Methods from Software Analysis

The question whether the data values at two different points in the control-flow of a program are equal or in some relation for all possible executions of the program is a fundamental one for software engineering. The analyses that answer this question serve numerous applica-

tions, viz., program optimization, verification, refactoring, comprehension, maintenance, etc., in software engineering. Naturally, there are a plethora of techniques that can essentially be viewed as performing equivalence checking. But for what concerns us, only those methods that explicitly deal with two program texts and check their functional equivalence are interesting.

In program optimization, a class of methods, called algorithm recognition methods, propose replacement of parts of a program with more efficient ones from a library of optimized programs (Metzger and Wen 2000). The proposal is to look through the input program and if any part of the program is found to be functionally equivalent to one of the programs in the library, the part is replaced with the program from the library. Therefore, algorithm recognition methods are equivalence checking methods for programs. In the context of program integration, source-code level equivalence checking methods (Yang et al. 1989) have been proposed based on program representation graphs and program slicing. Automatic validation of code-improving transformations on low-level program representations or assembly codes is another area where many proposals have been made for equivalence checking (Samet 1978; Subramanian and Cook 1996; Currie et al. 2000; van Engelen et al. 2004).

The above methods do not deal with detection of equivalences of sets of values in array variables for programs dominated with loops. A solution often proposed is to completely unroll the loops, but given that the loops are nested and the bounds are quite large in real programs, this is clearly infeasible in our target domain. This is especially so in embedded multimedia applications.

2.3 Methods Focused on Loops and Arrays

In this section, we discuss three methods that explicitly address handling loops and array variables in the program without resorting to unrolling of the loops. The first two methods are concerned with the equivalence checking of the two versions of programs, the transformation agent in between them being a compiler. The third method proposes a method for checking equivalence of SAREs (systems of affine recurrence equa-

tions) in the context of algorithm recognition.

2.3.1 Translation Validation Approach

Translation validation is an approach introduced by Pnueli et al. (1998a) for verifying the translation of a compiler by checking the semantic equivalence of the source and the target codes. The approach has since been actively pursued with the focus shifting from checking the correctness of the code generator in the compiler (Pnueli et al. 1998b) to checking the correctness of the code optimizer in the compiler (Necula 2000; Zuck et al. 2003). For the latter, the technique originally relied upon hints provided by the compiler in order to verify the optimizing transformations, but since then the heuristics used have matured relying less and less on the hints.

Much of the recent work following this approach focuses on verifying loop transformations. However, the approach makes use of rules in order to check the legality of the transformations (Goldberg et al. 2004). This has meant introduction of new rules in order to increase the class of loop transformations that can be handled (Hu et al. 2005). Also, the transformations are checked not in a unified manner but in multiple phases (Barrett et al. 2005).

The main drawback of this approach is that the use of rules implies that the checker essentially infers a valid transformation-path from one program to the other. Complete automation of verification then inherently either depends on the completeness of the system of rules or on the adequacy of the available hints. In the method we present, explicit rules are not used for individual transformation instances. Instead, we have identified only a few categories of transformations for which all the target transformation instances belong. In particular, we use rules in the style of translation validation only for transformations for which the need for them is inevitable, namely, for algebraic transformations.

2.3.2 Fractal Symbolic Analysis

Mateev et al. (2003) have proposed a technique called fractal symbolic analysis (FSA) to address the transformation verification problem from the viewpoint of the code optimizer inside a compiler. Their idea is to reduce the difference between the two programs by incrementally ap-

plying simplification rules until the two programs become close enough to allow a proof by symbolic analysis. Each simplification rule preserves the meaning of the program. These rules are similar to the ones proposed by the translation validation approach.

The programmer can also assist the analyzer by providing some problem specific program invariants. The power of FSA depends on the simplification rules that are available. However, the more rules, the larger the search space and it is yet unclear whether the heuristics to measure the difference and to select the most promising simplification rule are sufficient in practice. In comparison, our method, while addressing a more limited (but in practice important) class of transformations, does neither require search nor guidance from the programmer.

The fundamental distinction of our work when compared to FSA and the closely related translation validation approach, is that we separate the two different concerns: (1) the correctness of the control-flow (i.e., the schedule); and (2) the correctness of the data-flow, in the transformed program. We check the two independently of each other. This has helped to make our method oblivious of instances of the applied transformations and has allowed us to make the checking fully automatic without any need for intervention from the designer.

2.3.3 Equivalence Checking for SAREs

The work that is closest to our approach is the equivalence checking method for Systems of Affine Recurrence Equations (SAREs) proposed by Barthou et al. (2001). SAREs are a representation of the computation and data dependencies for programs that have certain properties. The method was developed for the purpose of algorithm recognition for program optimization, but it is applicable for verification of global loop and data-flow transformations applied on programs from which SAREs can be extracted.

SAREs characterize the same class of programs as the one handled by our method. The equivalence checking method they propose is similar to our method and was developed independently, in parallel to our own work. The differences between the two methods are in the way the encodings of the data dependencies are handled during the checking (Barthou et al. 2002). They leave handling of algebraic transformations

for future work, but normalization techniques that we propose for our method, in principle, could also be plugged into their basic method.

The difference in the motivation for the development of the two methods has also meant difference in focus as concerning the problem context. While their approach has since been generalized toward recognition of program templates (Alias and Barthou 2005), ours has been focused on addressing the context of verification of program transformations. Therefore, our solution relies on a pre-processing stage before accepting a program in a conventional programming language like C. Other key distinctions from the context are that our solution provides hooks for useful error diagnostics when a transformation is in error and some heuristics that can potentially reduce the checking time while handling large programs.

2.4 Prior Work at IMEC

Research on formal verification methodologies has previously been very active in IMEC with many interesting results. It was motivated by the correctness problems posed by application-specific hardware design and synthesis using *silicon compiler* environments. A significant body of work resulted from this research that addressed the various aspects of guaranteeing a correct implementation of digital VLSI system into silicon. In this section, we discuss the specific contributions of this prior work that have influenced the development of our solution to the transformation verification problem.

2.4.1 Claesen's SFG-Tracing Method

SFG-Tracing is a formal design verification methodology developed at IMEC (Claesen et al. 1991; Claesen et al. 1992). It allows verification of a lower level implementation against its higher level reference specification and its applicability has been demonstrated on several design case studies (Genoe et al. 1991; Genoe et al. 1994). The verification is achieved by means of a systematic tracing of the high level specification that can be represented as a signal flow graph (SFG) that establishes correctness of its refinement into a lower level implementation at behavioral register transfer (bRT), structural register transfer (sRT)

or switch levels. The language of choice for the high level specification was the applicative programming language for digital signal processing algorithms called Silage (Hilfinger 1985).

The method mainly focused on verification of translations from higher to lower abstraction levels. But it can also be used to verify source-to-source transformations applied on Silage programs. However, the main drawback of the method has been the need for inductive reasoning when faced with loop transformations. As a result, it is not very well suited for fully automatic verification of loop transformations. Therefore, the method has only been automated for non-loop transformations. The method we present bears resemblance to SFG-tracing in that it establishes correspondences by traversing the graph structure extracted from the specification. Our method takes advantage of array dependence analysis and availability of solvers for the theory of interest in order to obtain a fully automatic checking of equivalence conditions for programs involving loop constructs. It is feasible to combine our method and the SFG-tracing method to obtain a complete flow for formal equivalence checking. This has indeed been shown for a preliminary loop transformation verification method (Čupák et al. 2003).

2.4.2 Angelo's Theorem Prover based Method

A hybrid approach between a priori and a posteriori verification has been developed by Angelo (1994) in her dissertation. She proposed a *system for correct transformational design* that was based on the HOL theorem prover (Gordon 1985). The system was demonstrated for Silage programs and included a transformation editor, called SynGuide (Samsom et al. 1993), that lets the designer to select the code and apply transformations from a library. The library contains the pre-verified set of transformations.

The system presents a hybrid approach in the sense that, on the one hand, if the designer applies the transformations via SynGuide, they are correctly applied by construction. On the other hand, if the designer manually applies a transformation, the system can be supplied with the original and the transformed programs *and* the transformation that was applied, and the system semi-automatically verifies the transformation. The verification is rigorous, based on the formal semantics of Silage programs and HOL proof procedures. The system is, however, not suited for

the design context that we address where the designer wants flexibility in applying the transformations and at the same time desires a fully automatic verification of the applied transformations.

2.4.3 Samsom's Loop Verification Method

For our problem context and focus, Samsom (1995) has developed a fully automatic verification method for loop transformations in his dissertation. His approach is to use data dependence analysis and show the preservation of the dependencies in the original in the transformed program (Samsom et al. 1995). The program representation that he used for his method allows only a statement-level equivalence checking between the programs and hence cannot handle data-flow transformations. We provide a summary of his method and discuss its limitations in Section 6.2.

2.4.4 Čupák's System-Level Verification Methodology

In his dissertation, Čupák (1998) has showed the feasibility of system-level verification by combining Samsom's method for loop transformation verification and SFG-tracing for verification of high-level hardware synthesis (Čupák et al. 2003). He has also presented a heuristic called *incremental dimension handling* (Čupák et al. 1998) that addresses a specific problem related to Samsom's method that came from the choice of encoding of the data dependencies in the program representation. This problem has prevented the method from being able to handle larger programs.

2.5 Summary

In this chapter we have discussed several proposals for functional equivalence checking found in the literature. These proposals differ in the classes of programs and transformations they are able to handle. The transformation verification problem that is before us demands that its solution should:

1. allow full categories of global loop and data-flow transformations,

2. be fully automatic,
3. not rely on any user hints and be totally oblivious of the instances of the applied transformations,
4. not unroll the loops, but handle them in closed form,
5. be efficient in time and space consumption, and
6. sufficiently localize the errors when it suspects an incorrectly transformed program.

From the discussion of the proposals in the state-of-the-art, it follows that none of them meets all the requirements that are enumerated above. This has motivated our work and, as we will show in the chapters to follow, the method and the prototype implementation that we have developed successfully addresses all these requirements.

Chapter 3

Class of $\mathscr{D}$ Programs

3.1 Introduction

This chapter delineates a class of programs for which the method we develop is applicable. This class is characterized by a set of properties that hold for programs that it encompasses. Even though the individual properties of the class are well-known in the rather vast literature related to array data-flow analysis, there does not seem to be a single, generally accepted term for this class. Hence, for want of a better term, we choose to call the allowed programs as belonging to the *class of $\mathscr{D}$ programs*.

The $\mathscr{D}$ programs permit implementation of algorithms for numerical methods and commonly occur as kernels of signal processing and scientific computing applications. They are *transformational*, as opposed to *reactive*, in nature and are implemented in a sequential and structured ALGOL-like imperative programming language. To be *directly* applicable, our method relies on the properties of the $\mathscr{D}$ programs. We discuss each of these properties in Section 3.2. The properties are semantics-based, and hence, independent of the syntax of a particular programming language. Due to this reason, in many cases, even though a program does not seem to explicitly comply with a certain property, the underlying semantics does. In such cases, it is often possible to employ certain program analyses and transform a given program into an equivalent program for which the property holds explicitly. After we have explained

a property, whenever applicable, we provide a follow-up discussion on any relevant program analyses and code pre-processing transformations that can be leveraged to obtain a program for which the property is explicit.

The class of $\mathscr{D}$ programs, with all its enumerated properties, in theory, covers only a rather small set of general programs. However, it finds an important place in practice with regard to signal processing and scientific computing applications. In Section 3.3, we argue that limiting our method to this class, to a large extent, is justified, given the nature of the programs and the transformations in the application domain of our interest and the context where the verification problem we address arises.

3.2 Properties of $\mathscr{D}$ Programs

In this section, we discuss the properties that characterize the class of $\mathscr{D}$ programs. The discussion is necessarily brief, serving only to prepare the reader for the subsequent chapters where the properties have been made more precise.

3.2.1 Static Control-Flow

The control-flow of a program, that is, the order in which its instructions are executed, may either be dependent on the input data or not. Respectively, the program is said to have *data-dependent* or *data-independent* control-flow. The foremost property of $\mathscr{D}$ programs is that they have data-independent control-flow, or *static* control-flow, that is, the entire control-flow of the execution of the program can be exactly determined at compile-time. This property precludes the $\mathscr{D}$ programs from having data-dependent conditional branches, meaning that, data-dependent `goto` statements, `if`-statements, and `while`-loops are not allowed. This leaves statement sequence operators, data-*in*dependent `if`-statements and `for`-loops as the only available constructs to specify the control-flow of $\mathscr{D}$ programs.

Data-dependent conditional branches, when present in a program, require addition of *control dependencies* into the set of dependencies in

the program. Any legal program transformation that is applied to it has to satisfy this set of dependencies. Since the control dependencies increase the size of this set, they often stand in the way of application of aggressive optimizations to the program. Hence, it is common to reduce the influence of control dependencies in the program through some pre-processing transformations *before* subjecting it to analyses meant to determine opportunities to apply optimizing transformations. The `goto` statements, whether data-dependent or not, are easily dismissed since we have already reduced our focus to structured programs. In the case of data-dependent `if`-conditions, a trick that is commonly adopted is to convert the control dependencies into data dependencies by using *if-conversion* (Allen et al. 1983). In the case of `while`-loops, however, there is no such easy way out. The designer, through analyses or profiling, has to determine worst-case execution scenario and convert a `while`-loop into a `for`-loop with worst-case bounds. The data-dependent condition of the `while`-loop then appears as a global `if`-condition inside the body of the `for`-loop which is further subjected to if-conversion. Once these pre-processing transformations are applied we are left with a program with a control-flow that can be completely determined at compile-time.

3.2.2 Affine Indices and Bounds

A property of $\mathscr{D}$ programs that is as important as static control-flow is that of affine expressions for indices and bounds in the programs. It means that all arithmetic expressions in the subscripts of array variables and in the bounds of the `for`-loops must be affine functions in the iterator variables of the enclosing `for`-loops and symbolic constants. The functions need not be strictly affine, but piece-wise (or quasi-) affine, that is, they can also have the `mod`, `div`, `max`, `min`, `floor` and `ceil` operators.

In the context of array data-flow analysis, the class of programs with the properties of static control-flow and the affine expressions for indices and bounds was introduced by Feautrier (1991) in order to compute exact value-based data dependence information. In array data-flow analysis literature, the assumption of the affine property is, in general, common, since it allows representing the dependence relationships and the ordering in the schedules between assignment statements as systems of affine inequalities in integers and makes it possible to use well-understood dependence tests to solve those systems. This also holds for

the equivalence checking method we develop as this property is critical for the decidability or computability of many of the operations that we need to perform on sets of integers and mappings described by constraints expressed as affine functions.

3.2.3 Uniform recurrences

The data-flow in a program may have *recurrences*, that is, cycles in its flow. When confronted with a program with recurrences, there is an operation on an integer mapping that our method calls upon, namely, computation of the transitive closure of the mapping. Unfortunately, even for an affine mapping, the transitive closure is not exactly computable in the general case. However, if the given mapping for the recurrence is of the form $M(\vec{x}) = \vec{x} + \vec{c}$, where $\vec{x}$ is a vector of integer variables and $\vec{c}$ is a vector of constant integers, it meets a sufficient condition for the exact computation of the transitive closure (Pugh 1991; Kelly et al. 1996b). The recurrences of this form are also called *uniform recurrences* (Karp et al. 1967). Only uniform recurrences are allowed in the data-flow of a $\mathscr{D}$ program. We believe that this is not a major limitation for realistic multimedia application programs. Further, when confronted they can also be hidden in the bottom layer as discussed in Section 1.5.3 (see Figure 1.6), still enabling the verification of the rest of the program.

3.2.4 Single-Assignment Form

A program is said to be in *single-assignment form* when any memory location that is written in the program is written only once, but is read as many times as required. We stipulate a $\mathscr{D}$ program to be in single-assignment form.

We first give a background on the nature of dependencies and then discuss the available solution to show that it is not as strong a requirement as it appears to be. We start by recalling the terminology. Data dependencies in a program are usually classified under *true* dependencies and *false* dependencies. The true dependencies are due to the data-flow and hence are also called *flow* dependencies. They refer to the read-after-write (RaW) constraints that are inherent to the algorithm. The false dependencies are present because of two types of constraints due to reuse of memory locations to store values during computation. As the first

type of false dependence, there are write-after-read (WaR) constraints, which are also called *anti*-dependencies and as the second type, there are write-after-write (WaW) constraints, which are also called *output* dependencies. When a program is in single-assignment form, there is no reuse of any memory location by the program and hence, no false dependencies at all. By stipulating the $\mathscr{D}$ programs to be in single-assignment form, the burden on the analyses required for equivalence checking is significantly reduced.

As mentioned, the false dependencies are just artifacts due to reuse of memory. Their presence only inhibits application of transformations since the constraints due to false dependencies are also to be respected. Therefore, it is desirable to eliminate the false dependencies while keeping only the true dependencies before analyzing the program. This is well understood and optimizing compilers use the so-called *static single-assignment* (SSA) form (Cytron et al. 1991) to facilitate optimizations. But when array variables are present, SSA form does not ensure that false dependencies are completely eliminated. For a program to be completely free from false dependencies, every individual memory location is assigned a value at most once during the execution of the program. A *dynamic single-assignment* (DSA) form provides such a program form which is completely free from false dependencies. When we refer to the single-assignment form we exclusively refer to the DSA form. When a program is not in DSA form, but has static control-flow with affine indices and bounds, there are methods for converting it into an equivalent program in single-assignment form completely automatically (Vanbroekhoven et al. 2005b; Vanbroekhoven et al. 2003; Feautrier 1988).

3.2.5 Valid Schedule

We require that the control-flow of a $\mathscr{D}$ program is such that the schedule that it entails for the data-flow of the program is a valid one. In other words, we require that in a $\mathscr{D}$ program each value that is read in an assignment statement has either been already written or is an input variable.

The method we develop separates the two concerns of correctness of the control-flow and the correctness of the data-flow. For a $\mathscr{D}$ program checking the correctness of the control-flow reduces to checking whether the define-use (called DEF-USE) order is legal, which can be

done upfront separately. This is a regular check in most array data-flow analysis methods (Wolfe 1996; Allen and Kennedy 2001) and is crucial in determining valid loop transformations. Given a program that has static control-flow with affine indices and bounds, the validity of its schedule can easily be checked.

3.2.6 Other Properties

No pointer references

There are no pointer references in a $\mathscr{D}$ program. Our analysis assumes all data references are explicit and can be directly gathered by parsing the program. When a program has dynamic memory allocation and deallocation, in some cases sophisticated pointer-analysis can identify the data-flow information that can enable our method. When the program has only statically allocated data, and still has pointer references, so-called pointer-to-array conversion methods can be used. The method proposed by van Engelen and Gallivan (2001) is an example for such a method.

No side effects

All functions and operators in a $\mathscr{D}$ program are free from side-effects. That is, only an assignment statement can write to the memory at the location addressed by the element of the defined variable that is assigned. This is a common assumption made by many program analysis methods which are based on gathering information related to the program data-flow. Also, presence of side-effects is in general very undesirable in programs that are subject to optimizations/transformations. In our experience, we have not come across any implementation of a signal processing kernel relying on side-effects. However, when required, a suitable side-effect analysis method may be used to eliminate side-effects.

3.3 Justification for the $\mathscr{D}$ Class

As it is true with many of the interesting program analysis problems, equivalence checking problem between general purpose programs too is undecidable. In the restricted case of terminating programs, it is not

difficult to see that the problem is decidable – one just need run the programs for all possible inputs and compare the outputs for equivalence of each execution instance. However, the solution is still infeasible given the size of the domain of inputs in practice. This reasoning holds also for the class of $\mathscr{D}$ programs. Therefore, one needs a compromise – on the one hand, a language that is expressive enough to have practical relevance, and on the other hand, ability to find sufficient conditions that are applicable in practice. This implies that it is inevitable that we restrict ourselves to a sub-problem by focusing only on programs belonging to a certain class. The restriction is then required to entail either a decision procedure or a sufficient condition. However, if a solution to an equivalence problem is to be of interest in practice at all, the class of programs should still be representative of programs in the target application domain and be relevant for the transformations of interest. Fortunately, the class of $\mathscr{D}$ programs is indeed large enough to be representative of programs in the signal processing and scientific computing application domains and hence, relevant for high impact transformations applied in those domains.

Furthermore, from a design methodology perspective, there are two reasons that justify this class. Firstly, if the programs do not belong to this class, they can either not be analyzed at all or can only be analyzed poorly for applying transformations. Therefore, the agent applying the transformations also relies on the nice properties of the class that make them analyzable. For example, all optimizing compilers for high-performance computing that follow dependence-based approaches (Wolfe 1996; Allen and Kennedy 2001) require that the program has static control-flow with affine indices and bounds. Secondly, for all the required properties except the affine indices and bounds, and uniform recurrences, as we discussed, there are methods or workarounds to ensure that even if we have to deal with a program not in the class, we can transform it to an equivalent program that belongs to the class. Such pre-processing transformations pave the road for representation of the program in a model that we discuss in Chapter 4.

3.4 Summary

In this chapter we have discussed the programs that we will be limiting ourselves to, namely, the class of $\mathscr{D}$ programs. The distinguishing properties of this class are that they have static control-flow with affine indices and bounds, single-assignment form, a valid schedule and have only uniform recurrences. Even though the outlined program class represents a subset of programs expressible in an imperative language, it is not unduly restrictive for the application domains of our interest. Moreover, there are methods that help in converting programs into this class. In the following chapter, we discuss the representation that we adopt for a $\mathscr{D}$ program.

Chapter 4

Representation of $\mathscr{D}$ Programs

4.1 Introduction

This chapter presents the program representation used by our equivalence checking method. A representation serves as a data structure which stores the relevant information about the program and allows it to be queried in an efficient manner by a program analyzer. Choice of a certain suitable representation depends mainly on the analyses that it is meant to serve, and naturally, the literature is rich with numerous proposals for well-known classes of programs. In fact, even a simple compiler makes use of a plethora of representations during the different stages of compilation. Each representation has its scope, it either represents the whole program or only a part of it. Even within its scope, it is often restricted to one or more aspects of its program code that is relevant to the concerned analyses. Therefore, development of an analysis method often entails a set of choices that gives a representation that is best suited for its purpose.

We are given the class of $\mathscr{D}$ programs, therefore our representation is necessarily restricted to it. In this class, for the scope of the whole program, our concern is with the data dependencies that exist between the array variables in the program and the computation that intervenes between them. The most important aspect of the representation is that it

```
void foo (int in1[], int in2[], int out[][])
{
  int i, tmp[2*N];

  for (k = 0; k <= 2*N-2; k++)
      tmp[k] = f(in1[k], in2[3*k]); // statement X

  for (i = 0; i <= N-1; i++)
      for (j = 0; j <= N-1; j++)
          out[i][j] = tmp[i+j];      // statement Y
}
```

Figure 4.1: A simple program.

is able to capture *sets* of data dependencies at once, without enumeration, in a compact form (Section 4.2). It also captures the computation in the data-flow in terms of the operators across which the data dependencies hold. Given that $\mathscr{D}$ programs have a static control-flow with a valid schedule, these two pieces of information are all that is there of interest to our method. Therefore, the representation essentially presents the complete information about data dependencies in the program and the intervening computation in terms of a graph called an array data dependence graph (Section 4.3). The representation is for the whole program, and therefore, it is a union of data-flows that defines each of the output variables. The method we develop identifies corresponding data-flows in the two representations, and in doing so, keeps focus on a window of the data-flow that is relevant at any given instance. This brings in the notions of paths and slices in the representation that we discuss in Sections 4.4 and 4.6, respectively.

Example 1

Consider a simple program from the $\mathscr{D}$ class as shown in Figure 4.1. We discuss the information that our representation captures from the program. Let us assume that `N` is a positive integer.

We start by noting the observables in the program, that is, it has two input variables, viz., `in1[]` and `in2[]`, and one output variable, `out[][]`. The actual identifiers of the iterator variables are of no consequence for us, therefore the only remaining variable of interest in the program is, `tmp[]`, which is a temporary variable. The data dependencies within ev-

ery assignment statement is noted first and then any data dependencies between the statements is inferred. There are two assignment statements in the program, viz., X and Y.

The execution of X is controlled by the `for`-loop that encloses it, with an iterator `k` that ranges from 0 to $2 * \mathtt{N} - 2$ in unit steps. Therefore, X is executed $2 * \mathtt{N} - 1$ times, once for each instance of `k`. The right-hand side of X contains a simple expression with just one operator $\mathtt{f}/2 : (\mathtt{int}, \mathtt{int}) \rightarrow \mathtt{int}$. When `k` is the value of the iterator, $\mathtt{f}/2$ accepts `in1[k]` and `in2[3 * k]` as the input values and computes a new value, which is assigned to `tmp[k]` by X. This implies a data dependence from `tmp[]` to `in1[]` and `in2[]`. The dependency is characterized by two element-wise mappings between the indices to their elements, one from `tmp[]` to `in1[]` and another from `tmp[]` to `in2[]`, which are obtained from the index expressions of the respective variables. In this case, they are from $(\mathtt{k})$ to $(\mathtt{k})$, for the former and from $(\mathtt{k})$ to $(3 * \mathtt{k})$, for the latter, for all values that iterator `k` takes. Also, the set of values of the indices to the elements of `tmp[]` that is assigned and the set of values of the indices to the elements of `in1[]` and `in2[]` that are read by X are given by the domain and image of the mappings, respectively. Similarly to X, the data dependence in Y is between `out[][]` and `tmp[]` and, with `i` and `j` being the iterators, their element-wise mapping is from $(\mathtt{i}, \mathtt{j})$ to $(\mathtt{i}+\mathtt{j})$, where both `i` and `j` take unit steps from 0 to $\mathtt{N} - 1$.

The statements X and Y have a variable `tmp[]` that is common between them. Moreover, the set of elements of `tmp[]` that is written by X, overlaps with the set of elements of `tmp[]` that are read by Y. This overlap, which happens to be complete in the present case, links the individual data dependencies in the two statements and helps not only infer the existence of the data dependence from `out[][]` to `in1[]` and `in2[]`, but also helps in computing the element-wise mappings between them when needed. In the representation that is presented in this chapter, all the above discussed pieces of information, including the information about the overlaps between the writes and their subsequent reads, are captured in a graph structure.

Some remarks are necessary before we begin. In our discussion, we only need to deal with assignment statements. Therefore, we refer to them as just statements and the fact that we are referring to only assignment statements is implicit. Also, we use the words arrays and variables interchangeably to mean the same. In order to reduce the clutter, in all the source codes that we use in our examples, we drop the program interface and variable declarations, and only show the body of the program code. Any symbolic constants we use as `for`-loop bounds are assumed to be non-negative integers. In the text, whenever we have occasion to

```
for (k_1 = ...)
 ...
 for (k_r = l_r(k⃗_{r-1}); k_r ≤ u_r(k⃗_{r-1}); k_r = k_r + s_r(k⃗_{r-1}))
  if (c_r(k⃗_r))
   ...
   for (k_d = ...)
    ...
S:   v[f_{i_1}(k⃗)]...[f_{i_n}(k⃗)] = exp(...,u[f_{j_1}(k⃗)]...[f_{j_m}(k⃗)],...);
```

Figure 4.2: Template code for an assignment statement in a $\mathscr{D}$ program.

refer to an array by its identifier, we suffix it with '`[]`' for easier reading, irrespective of the array's dimensionality. Also, we assume that all references to an array respect the declared bounds in each of its dimensions.

4.2 Representation of Data Dependencies

In a single-assignment form program all the variables, other than the iterators, can be replaced with array variables. This implies that values computed and assigned in a statement depend on the instantiation of the subscripts of the variables in it. A statement might be outside the scope of any `for`-loops, in which case the arrays in it will have constant values in the subscripts. On the other hand, if it is enclosed inside a nest of `for`-loops, the values of the subscripts are obtained when the index expressions of the variables are evaluated. This information on what values the subscripts take during execution of the statement can be described in closed form as a domain of integer points in a multi-dimensional geometrical space. Such descriptions which record a variety of information related to the statements and dependencies among them are together referred to as the geometrical or polyhedral representation. This representation is commonly used for dependence analysis by optimizing, especially parallelizing, compilers (Collard 2003; Allen and Kennedy 2001; Wolfe 1996; Bacon et al. 1994; Banerjee 1988). Again, depending on the kind of analysis, the representation includes various kinds of objects about the nature of data and control dependencies in the program. But in what follows, we only define the geometric objects that are relevant for our representation.

Let us consider a statement S that appears in a body of program code inside a nest of d `for`-loops with $\vec{k} = (k_1, \ldots, k_r, \ldots, k_d)$ as its vector of iterator variables. The general template for such an assignment statement in a $\mathscr{D}$ program is as shown in Figure 4.2. Each of the statements of a program in the class of allowed programs falls into this template, and hence, the objects that we define can be extracted for each of them from the code.

In the template code shown, let l_r, u_r and s_r shown in the code be affine functions defining the lower and upper bounds, and the stride (or step length), respectively, for each iterator variable k_r in the nest of `for`-loops. Also, let c_r be a conditional expression on affine functions, $\vec{k}_r$ be the partial vector $(k_1, \ldots, k_r)$, and $l_1(\vec{k}_0)$ and $u_1(\vec{k}_0)$ be integer constants. We have shown all the `for`-loops to be varying from the lower bound to the upper bound in the increasing order, but we do not lose generality due to this. For our purposes, bounds only serve to represent the range of the iterator variable of the `for`-loop and the order information is of no consequence as far as the objects of interest to us are concerned. Therefore, the roles of the two bounds can simply be reversed in the case of a `for`-loop with decreasing iterator.

The data values that are referenced through the variables in the statement *S* are all in terms of affine functions on the iterator variables. Therefore, it is first required that we determine the domain, called the *iteration domain* of the statement, which provides the values which instantiate the iterator variables. We can define the iteration domain of *S* as follows.

Definition 4.2.1 (Iteration domain, D)
Integer domain in which each point $[k_1, \ldots, k_d]$ *represents exactly one execution of the statement S.*

$$\mathbf{D} := \{[k_1, \ldots, k_d] \mid \bigwedge_{r=1}^{d} ((\exists\, \alpha_r \in \mathbb{Z} \mid\ k_r = \alpha_r s_r(\vec{k}_{r-1}) + l_r(\vec{k}_{r-1})) \\ \wedge\ l_r(\vec{k}_{r-1}) \leq k_r \leq u_r(\vec{k}_{r-1}) \\ \wedge\ c_r(\vec{k}_r) \wedge k_r \in \mathbb{Z}\ )\}.$$

Once the iteration domain for a given statement is determined, we can determine the domains of references for its individual arrays. Suppose

that the statement reads an m dimensional array `u[]` among other arrays, applies the computation defined by `exp(...)` on them and assigns the computed values to an n dimensional array `v[]`. Let $f_{_}$ represent affine functions defined by the index expressions within the subscripts of the array variables. For a given instance of values of the iterators, these functions are evaluated to provide the subscripts for each array, and hence, uniquely reference a particular element of the array. The collection of these references taken together defines its domain of reference, *definition domain* if the array is on the left-hand side and *operand domain* if the array is on the right-hand side of the statement.

Definition 4.2.2 (Definition domain, ${}_S\mathbf{W}_v$)
Integer domain in which each point $[i_1, \ldots, i_n]$ *represents exactly one element,* `v[`i_1`]...[`i_n`]`*, of the array* `v` *defined by the statement* S *with iteration domain* $\mathbf{D}$.

$$ {}_S\mathbf{W}_v := \{[i_1, \ldots, i_n] \mid (\exists\, \vec{k} \in \mathbf{D} \mid \bigwedge_{r=1}^{n} i_r = f_{i_r}(\vec{k}))\}. $$

Due to the single-assignment form of our programs, each point of the definition domain also represents exactly one write to its element in the array. Moreover, each reference due to a point in a definition domain is unique not only for the execution of a given statement from which the definition domain is derived, but for the *entire* program. Therefore, the set of definition domains for a given array, drawn from all the statements in a given program, are all mutually disjoint.

Definition 4.2.3 (Operand domain, ${}_S\mathbf{R}_u$)
Integer domain in which each point $[j_1, \ldots, j_m]$ *represents exactly one element,* `u[`j_1`]...[`j_m`]`*, of an operand array* `u` *in statement* S *with iteration domain* $\mathbf{D}$.

$$ {}_S\mathbf{R}_u := \{[j_1, \ldots, j_m] \mid (\exists\, \vec{k} \in \mathbf{D} \mid \bigwedge_{r=1}^{m} j_r = f_{j_r}(\vec{k}))\}. $$

In contrast to the definition domain, the points in the operand domain may represent multiple reads to its element in the array. This means, references to an operand array during different instances of the execution of the same statement need not be unique and the operand domain

gives the set of all the elements of an array that are referenced by the statement. Nevertheless, each element of an array that is written in the left-hand side depends on a single element of each of the arrays on the right-hand side. This dependence, which is in reverse direction to the flow of data values, is a mapping from the definition domain to the operand domain and it is called the *dependency mapping*. It is defined separately for each of the arrays in the right-hand side.

Definition 4.2.4 (Dependency mapping, ${}_S\mathbf{M}_{v,u}$)
A mapping associated with a statement, between a defined array v *and an operand array* u*. Each instance* $[i_1,\ldots,i_n] \rightarrow [j_1,\ldots,j_m]$ *in the mapping indicates that element* $u[j_1]\ldots[j_m]$ *is read when the element* $v[i_1]\ldots[i_n]$ *is written by the statement* S *with iteration domain* $\mathbf{D}$*.*

$$ {}_S\mathbf{M}_{v,u} := \{[i_1,\ldots,i_n] \rightarrow [j_1,\ldots,j_m] \mid (\exists \vec{k} \in \mathbf{D} \mid (\bigwedge_{r=1}^{n} i_r = f_{i_r}(\vec{k})) \wedge (\bigwedge_{r=1}^{m} j_r = f_{j_r}(\vec{k})))\}. $$

Note that, for a given program, the identifier for a statement in it, gives access to the definition domain, the operand domains and the dependency mappings of the statement. Given all the operand and definition domains of all the statements in a program, we can partition the set of arrays into sets of *output* arrays, *internal/temporary* arrays and *input* arrays. This partitioning follows from observing the union of all the definition domains and the union of all the operand domains of a given array for the whole program. An array, `v[]`, is an output array if some if its elements are not read by any of the statements in the program. This holds when the union of all the operand domains of `v[]` is a strict subset of the union of all the definition domains of `v[]`. That is, when $\bigcup \mathbf{R}_v \subset \bigcup \mathbf{W}_v$. An array, `v[]`, is an input array if none of its elements are written in the program, i.e., $\bigcup \mathbf{W}_v = \emptyset$. Finally, `v[]` is an internal/temporary array when all of its elements are written and also read, i.e., $\bigcup \mathbf{W}_v = \bigcup \mathbf{R}_v$.

All the domains and the mapping that we have defined are described using logical formulas, called *Presburger formulas* (Presburger 1929), that are built from affine constraints over integer variables, symbolic constants, logical connectives $\neg$, $\wedge$ and $\vee$, and the quantifiers $\forall$ and $\exists$. We will need to perform many operations involving these formulas and they imply certain computability and complexity issues. We will

```
for (k = 0; k <= 2*N-2; k++)
    if (k < N){
       tmp[k] = f(in1[k], in2[3*k]);     // P
       for (p = 0; p <= k; p++)
           out[p][k-p] = tmp[k];         // Q
    } else {
       tmp[k] = f(in1[k], in2[3*k]);     // R
       for (p = k-N+1; p <= N-1; p++)
           out[p][k-p] = tmp[k];         // S
    }
```

Figure 4.3: Program in Figure 4.1 after some transformations.

postpone discussion on such implications that arise due to the expressive power of these formulas until we have explained our method.

As far as our notations for the domains and mappings are concerned, we liberally decorate them with additional subscripts/superscripts as and when the occasion calls for disambiguation between objects from different programs, statements or arrays in different argument positions within the right-hand side of a given statement.

Example 2

Consider the program in Figure 4.3. It has been obtained after applying some transformations on the simple program in Figure 4.1. We present the domains and mappings extracted from all the four statements in the program viz., P, Q, R and S.

Statement P:

$\mathbf{D}_P := \{[k] \mid 0 \leq k \leq 2 * N - 2 \wedge k < N \wedge k \in \mathbb{Z}\}$

${}_P\mathbf{W}_{tmp} := \{[i_1] \mid (\exists\, k \in \mathbf{D}_P \mid i_1 = k)\}$

${}_P\mathbf{R}_{in1} := \{[j_1] \mid (\exists\, k \in \mathbf{D}_P \mid j_1 = k)\}$

${}_P\mathbf{M}_{tmp,in1} := \{[i_1] \rightarrow [j_1] \mid (\exists\, k \in \mathbf{D}_P \mid i_1 = k \wedge j_1 = k)\}$

${}_P\mathbf{R}_{in2} := \{[j_1] \mid (\exists\, k \in \mathbf{D}_P \mid j_1 = 3 * k)\}$

${}_P\mathbf{M}_{tmp,in2} := \{[i_1] \rightarrow [j_1] \mid (\exists\, k \in \mathbf{D}_P \mid i_1 = k \wedge j_1 = 3 * k)\}$

Statement Q:

$\mathbf{D}_Q := \{[k,p] \mid 0 \leq k \leq 2 * N - 2 \wedge k < N \wedge 0 \leq p \leq k \wedge (k,p) \in \mathbb{Z}^2\}$

${}_Q\mathbf{W}_{out} := \{[i_1,i_2] \mid (\exists\, (k,p) \in \mathbf{D}_Q \mid i_1 = p \wedge i_2 = k - p)\}$

${}_Q\mathbf{R}_{tmp} := \{[i_1] \mid (\exists\, (k,p) \in \mathbf{D}_Q \mid i_1 = k)\}$

$_{Q}\mathbf{M}_{out,tmp} := \{[i_1, i_2] \rightarrow [j_1] \mid (\exists\, (k, p) \in \mathbf{D}_Q \mid i_1 = p \wedge i_2 = k - p \wedge j_1 = k)\}$

Statement R:

$\mathbf{D}_R := \{[k] \mid 0 \leq k \leq 2 * N - 2 \wedge k \geq N \wedge k \in \mathbb{Z}\}$

$_{R}\mathbf{W}_{tmp} := \{[i_1] \mid (\exists\, k \in \mathbf{D}_R \mid i_1 = k)\}$

$_{R}\mathbf{R}_{in1} := \{[j_1] \mid (\exists\, k \in \mathbf{D}_R \mid j_1 = k)\}$

$_{R}\mathbf{M}_{tmp,in1} := \{[i_1] \rightarrow [j_1] \mid (\exists\, k \in \mathbf{D}_R \mid i_1 = k \wedge j_1 = k)\}$

$_{R}\mathbf{R}_{in2} := \{[j_1] \mid (\exists\, k \in \mathbf{D}_R \mid j_1 = 3 * k)\}$

$_{R}\mathbf{M}_{tmp,in2} := \{[i_1] \rightarrow [j_1] \mid (\exists\, k \in \mathbf{D}_R \mid i_1 = k \wedge j_1 = 3 * k)\}$

Statement S:

$\mathbf{D}_S := \{[k, p] \mid 0 \leq k \leq 2*N-2 \wedge k \geq N \wedge k-N+1 \leq p \leq N-1 \wedge (k, p) \in \mathbb{Z}^2\}$

$_{S}\mathbf{W}_{out} := \{[i_1, i_2] \mid (\exists\, (k, p) \in \mathbf{D}_S \mid i_1 = p \wedge i_2 = k - p)\}$

$_{S}\mathbf{R}_{tmp} := \{[i_1] \mid (\exists\, (k, p) \in \mathbf{D}_S \mid i_1 = k)\}$

$_{S}\mathbf{M}_{out,tmp} := \{[i_1, i_2] \rightarrow [j_1] \mid (\exists\, (k, p) \in \mathbf{D}_S \mid i_1 = p \wedge i_2 = k - p \wedge j_1 = k)\}$

4.3 Array Data Dependence Graphs (ADDGs)

In the previous section, we have discussed how data dependencies that exist between the arrays within a given statement are represented as mappings. This information by itself says little about either the computation or the relationship between statements in the program. But we want our representation to capture information at a finer-grain level about the computation inside each statement and also show dependencies between statements in the program. This is facilitated by recognizing the other data dependencies that exist in the program. They exist, firstly, due to the operators in the computation in the right-hand side of the statement, and secondly, due to the reads of values from an array by a statement that are written earlier by other statements in the program or possibly by itself. This requires that we link the operators and arrays across statements and a graph structure is naturally well-suited for this purpose. We present such a structure in this section.

We start by looking at an individual statement in a program first. In the simple case, this statement may be representing just a copy of values from one array to another. In this case, the array on the left-hand side depends directly on the array on the right-hand side. This dependence is represented trivially by the mapping that we mentioned in the previous section. In the general case, the statement may be representing a com-

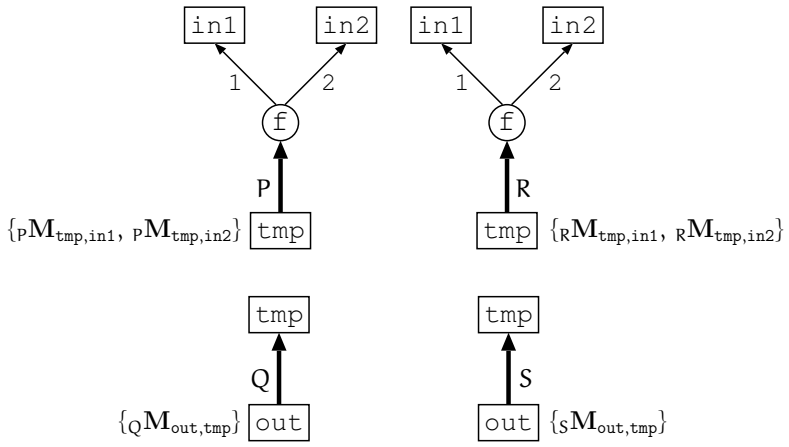


Figure 4.4: The ADDGs of the statements in the program in Figure 4.3.

putation that takes values from its operand arrays and generates new values that are copied to another array. In this case, the computation is revealed in terms of the operators in the expression on the right-hand side. The array on the left-hand side depends on the operator that fires last depending on the precedence order among the operators in the expression. This last firing operator in turn depends on its arguments which are either other operators or operand arrays.

Now we look at the relationship between statements in the program. A data dependence exists between two statements S and T when S stores a set of values in an array and T subsequently reads a set of values from the same array and there is an overlap between the two sets. That is, if `v[]` is the common array and its definition domain in S is ${}_{S}\mathbf{W}_{v}$ and its operand domain in T is ${}_{T}\mathbf{R}_{v}$, then a data dependence exists from T to S provided that ${}_{S}\mathbf{W}_{v} \cap {}_{T}\mathbf{R}_{v} \neq \emptyset$.

The set of all the dependencies that we discussed can be represented in an array data dependence graph (ADDG) defined as follows.

Definition 4.3.1 (Array Data Dependence Graph, ADDG)
The ADDG of a $\mathscr{D}$ program is a directed graph $G(V, E)$*, where the node set* V *is the union of the arrays used in the program (array nodes) and the operator occurrences (operator nodes) of the statements and the edge set* E *represents the dependencies. An edge with operator node as source is*

labeled by the operand position of its destination; an edge with an array as source is labeled with the statement identifier of the assignment. Array nodes of defined arrays are annotated with the dependency mappings of the statements.

Example 3

Consider the program in Figure 4.3. The ADDGs obtained by parsing the right-hand sides of the four statements in the program are as shown in Figure 4.4. The ADDG of the whole program combines the individual ADDGs by uniting the identical array nodes as shown in Figure 4.5. Note that the directed edges in bold represent assignments to the arrays at their source.

Standard data dependence graphs used in high-performance compilers represent dependencies at the statement level (Allen and Kennedy 2001; Wolfe 1996), whereas in an ADDG, the data dependencies are at the level of variables. Therefore, an ADDG provides more fine grain level information about dependencies than standard data dependence graphs. Also, a data dependence, denoted by a directed edge in an ADDG, refers not just to a single value, but to a *set of values*. Indeed, in the discussion that is to follow, it will help to view each edge as a data-bus, with the dependency mapping defining the connectivity relation of the lines in the data-bus from the elements of the defined array (source) to the elements of the operand array (destination), passing through zero or more operation units.

4.4 Data Dependence Paths

Once we have extracted the ADDG from a given program, we can reason about the data dependencies and infer transitive relationships between arrays at the level of the whole program. In this section, we restrict ourselves to transitive relationships between any two given arrays in the program, and discuss how such relationships can be established.

Consider the ADDG of any single statement. Any path from the assigned array to any of the statement's operand arrays contains only operator nodes. Suppose array `u[]` is assigned in a given statement. In its ADDG consider a path from node `u` to an operand node `v` representing array

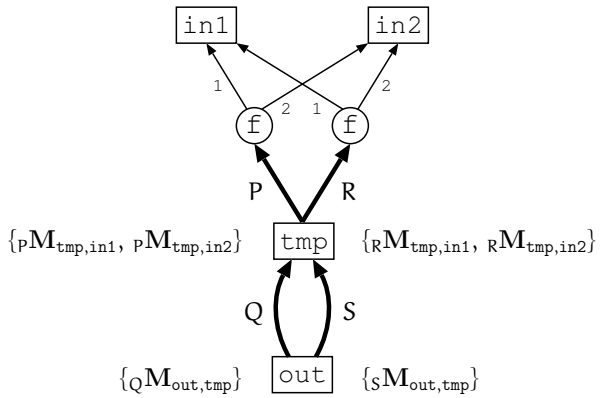


Figure 4.5: The ADDG of the program in Figure 4.3.

v[] that appears as the k-th argument for the right-hand side expression. It possibly has operators $o_1, \ldots, o_n$ on the path. From the Definition 4.2.4, for this path, the element-wise dependence from u[] to v[] is given by the dependency mapping $\mathbf{M}_{u,v,k}$.

Now consider the ADDG of the whole program. Suppose there is a path from an array node in the ADDG to another array node and there are other array nodes intervening on the path along with operator nodes. Since we have the dependency mappings between every two adjacent arrays on the path extracted from the individual statements, we can compute the element-wise relationship between the arrays at the ends of the path. This relationship is called the *transitive dependency mapping* from one array to the other. It is defined as follows.

Definition 4.4.1 (Transitive dependency mapping, ${}_p\mathbf{M}_{v_0 \rightsquigarrow v_n}$)
Let p be a path in an ADDG starting in array node v_0, ending in an array node v_n and passing through array nodes $v_1, \ldots, v_{n-1}$ ($n \geq 0$). Suppose that $s_0, \ldots, s_{n-1}$ are the statements that assign the arrays on the path. Then, the transitive dependency mapping from v_0 to v_n on path p is given by,

$$ {}_p\mathbf{M}_{v_0 \rightsquigarrow v_n} := \begin{cases} \text{RestrictDomain}(I, {}_{s_0}\mathbf{W}_{v_0}) & n = 0 \\ {}_{s_0}\mathbf{M}_{v_0,v_1} & n = 1 \\ {}_{s_0}\mathbf{M}_{v_0,v_1} \bowtie {}_{s_1}\mathbf{M}_{v_1,v_2} \bowtie \ldots \bowtie {}_{s_{n-1}}\mathbf{M}_{v_{n-1},v_n} & \text{otherwise.} \end{cases} $$

where, I represents the identity mapping.

The $\bowtie$ symbol used above stands for the join operator on relations. Its definition is as provided in Appendix A.

Note that the transitive dependency mapping computed on a certain path from a source array to a destination array need not be defined for the full definition domain of the source array at the first statement on the path. This can happen because subsets of the elements of an intermediate array on the path might be written in different statements, leading to *branching* of the dependencies of the source into other paths along the way. In fact, for a given path, the transitive dependency mapping can even be empty, meaning that, the source array is not data dependent at all on the destination array via the given path. But we are interested in a path only if it implies data dependencies. We distinguish such a path as a *data dependence path*. It is defined as follows.

Definition 4.4.2 (Data dependence path)
A path, p*, between two array nodes,* v_o *and* v_n*, is a data dependence path iff* ${}_\mathrm{p}\mathbf{M}_{v_\mathrm{o} \rightsquigarrow v_\mathrm{n}} \neq \emptyset$.

In the reasoning that we develop further on, it will be convenient if we generalize the notion of a path between two array nodes described above, to a path between two arbitrary nodes of an ADDG. In this context, given a (transitive) dependency mapping leading to the beginning of such a path, the computation of the transitive dependency mapping upto the last array node in the path is an operation that will be called upon by the algorithms that we develop later on. The steps in the operation are given by Algorithm 1.

A data dependence path from an output array to an input array is of special interest for us. It defines an element-wise transitive dependency mapping from the elements of the output array to the elements of the input array, or simply, an *output-to-input mapping*, for a path in the ADDG of the program. The set of output-to-input mappings, for all the paths taken together, characterizes the complete data-flow information in the program. So, for the purposes of our equivalence checking method, the ADDG adequately serves as the internal representation with all the relevant information from the program.

Algorithm 1: Computation of transitive dependency mapping.

ComputeMapping$(G, {}_{q}\mathbf{M}_{x\rightsquigarrow y}, b, e)$
Input: An ADDG G, two nodes b and e representing the first and the last nodes of a path p of interest in it, and a (transitive) dependency mapping ${}_{q}\mathbf{M}_{x\rightsquigarrow y}$ leading to the path p. Note that the path p that is referred here will be known from the calling context and is kept implicit here.
Output: The transitive dependency mapping m that updates ${}_{q}\mathbf{M}_{x\rightsquigarrow y}$ upto the last array node on the path p.

```
begin
    m ⟵ _qM_{x⇝y};
    v ⟵ y;
    while b ≠ e do
        if succ(b) is an internal array node then
            l ⟵ label of the first edge on the path from v to
            succ(b);
            m ⟵ m ⋈ _lM_{v,succ(b)};
            v ⟵ succ(b);
        b ⟵ succ(b);
    if v = e then
        return m;
    else
        l ⟵ label of the first edge on the path from v to e;
        m ⟵ RestrictRange(m, _lW_v);
    return m
end
```

Example 4

Consider the ADDG in Figure 4.5. Using the definitions from Example 2 in page 53, we show the computation of the transitive dependency mapping from `out[]` to `in1[]` for two of the four paths from `out[]` to `in1[]`.

For the leftmost path from `out[]` to `in1[]`, that is,

$$p_1 : \texttt{out} \xrightarrow{Q} \texttt{tmp} \xrightarrow{P} \texttt{f} \xrightarrow{1} \texttt{in1},$$

the transitive dependency mapping is given by,

$$\begin{aligned}
{}_{p_1}\mathbf{M}_{\mathtt{out}\rightsquigarrow\mathtt{in1}} &:= {}_{Q}\mathbf{M}_{\mathtt{out},\mathtt{tmp}} \bowtie {}_{P}\mathbf{M}_{\mathtt{tmp},\mathtt{in1}} \\
&:= \{[i_1, i_2] \rightarrow [j_1] \mid (\exists\, (k, p) \in \mathbf{D}_Q \mid i_1 = p \\
&\qquad\qquad \wedge i_2 = k - p \wedge j_1 = k)\} \\
&\quad \bowtie \{[i_1] \rightarrow [j_1] \mid (\exists\, k \in \mathbf{D}_P \mid i_1 = k \wedge j_1 = k)\} \\
&:= \{[i_1, i_2] \rightarrow [j_1] \mid (\exists\, (k, p) \in \mathbf{D}_Q \mid i_1 = p \\
&\qquad\qquad \wedge i_2 = k - p \wedge j_1 = k)\}.
\end{aligned}$$

For the next to leftmost path from `out[]` to `in1[]`, that is,

$$p_2 :\ \mathtt{out} \xrightarrow{Q} \mathtt{tmp} \xrightarrow{R} \mathtt{f} \xrightarrow{1} \mathtt{in1},$$

the transitive dependency mapping is given by,

$$\begin{aligned}
{}_{p_2}\mathbf{M}_{\mathtt{out}\rightsquigarrow\mathtt{in1}} &:= {}_{Q}\mathbf{M}_{\mathtt{out},\mathtt{tmp}} \bowtie {}_{R}\mathbf{M}_{\mathtt{tmp},\mathtt{in1}} \\
&:= \{[i_1, i_2] \rightarrow [j_1] \mid (\exists\, (k, p) \in \mathbf{D}_Q \mid i_1 = p \\
&\qquad\qquad \wedge i_2 = k - p \wedge j_1 = k)\} \\
&\quad \bowtie \{[i_1] \rightarrow [j_1] \mid (\exists\, k \in \mathbf{D}_R \mid i_1 = k \wedge j_1 = k)\} \\
&:= \emptyset.
\end{aligned}$$

Due to the non-empty transitive dependency mapping, the path p_1 defines a data dependence path, whereas the path p_2 does not. In p_2, the elements of `tmp[]` that elements of `out[]` depend on along statement Q are entirely defined by statement P. Therefore, at `tmp[]`, the data dependencies due to Q entirely branch into P, resulting in their discontinuation along R. This renders the path p_2, as far as reasoning with data dependencies is concerned, uninteresting. For the same reason as for p_2, another path in the ADDG that has Q and R as adjacent edges leading to `in2[]` is not a data dependence path either.

Note that since p_1 is a data dependence path from an output array to an input array, the mapping ${}_{p_1}\mathbf{M}_{\mathtt{out}\rightsquigarrow\mathtt{in1}}$ also defines an output-to-input mapping.

4.5 Recurrences in Paths

Recall that the definition of the ADDG only states that it is a directed graph. Therefore, it is possible that there may be cycles in a given ADDG. However, due to the single-assignment property of a $\mathscr{D}$ program, a cycle does by no means imply a *data dependence cycle*. Instead, if present at

```
tmp[0] = in[0];                 // X
for (k = 1; k <= N; k++)
    tmp[k] = f(tmp[k-1]);       // Y
out[0] = tmp[N];                // Z
```

(a) A simple program with a recurrence

```
for (k = 0; k < N; k++)
    tmp[k] = in[N];             // X
for (k = N; k < 2*N; k++)
    tmp[k] = f(tmp[k-N]);       // Y
for (k = 0; k < N; k++)
    out[k] = tmp[k+N];          // Z
```

(b) A program without a recurrence

Figure 4.6: Example programs to illustrate recurrence.

all, it may only imply a data dependence path that is like a *coil*. Such a path indicates that there exists a set of statements that define the arrays in a coiled data dependence path and these arrays depend on themselves for values assigned by the same set of statements earlier in the execution. The statements involved are then said to define a *recurrence* in the dependencies in the data-flow.

Example 5

Consider the program in Figure 4.6(a). Assume that $\texttt{N} > 0$. The program takes an input value `in[0]` and applies the function `f`/1 on it `N` times and copies the computed value to `out[0]`. Its ADDG representation is as shown in Figure 4.7. The ADDG contains the cycle $\texttt{tmp} \xrightarrow{Y} \texttt{f} \xrightarrow{1} \texttt{tmp}$. However, from the program it is clear that the value at `tmp[N]` depends on the value at `tmp[N-1]`, which in turn depends on the value at `tmp[N-2]` and so on, until `tmp[1]`, all of which are assigned by statement Y itself. The element at the end, `tmp[0]`, is assigned in statement X that takes it out of the coil.

Now, consider the program in Figure 4.6(b). It has the same ADDG as shown in Figure 4.7, except that its annotated dependency mappings will be different from those for the program in Figure 4.6(a). But even though there is a cycle in the ADDG, the program does not have a recurrence in

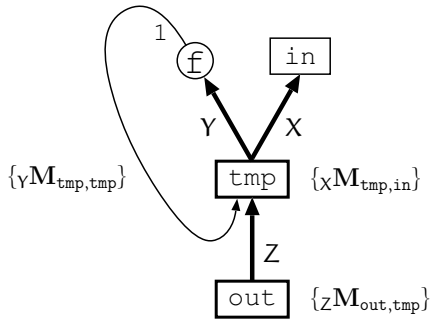


Figure 4.7: The ADDG of the program in Figure 4.6(a).

its data-flow. This is due to the fact that none of the elements of array `tmp[]` that are read by the statement Y are defined by itself.

Also recall that a $\mathscr{D}$ program allows only terminating `for`-loops. Therefore, all data dependence paths in the program are finite. Hence, all coils in the program are also of finite length. But the coiled path, depending on how many elements of the same array are defined by the recurrence, might still be very long. Therefore, when we are required to compute the transitive dependency mapping between arrays that are separated by a coiled path, it is obvious that we need a better way than resorting to exhaustively stepping through the recurrence, which would be analogous to completely uncoiling the recurrence. Fortunately, for $\mathscr{D}$ programs, since they have only uniform recurrences (Section 3.2.3), it is always possible to compute the transitive dependency mapping over a recurrence, or simply, *across-recurrence mapping*, directly without enumeration. The steps in such a computation are given by Algorithm 2. The domains and mappings used by the algorithm in computing the across-recurrence mapping are also illustrated in Figure 4.8.

Example 6

Consider the program in Figure 4.6(a). We show how the transitive dependency mapping from array `out[]` to array `in[]` that are separated by a recurrence as discussed in Example 5 is computed. Note that the bound `N` in the program is a symbolic positive integer constant.

The array `out[]` is assigned by statement Z and the successor array `tmp[]` anchors the trivial cycle $\mathtt{c} = (\mathtt{tmp}, \mathtt{tmp})$ assigned by statements (Y, Y). Therefore, we first compute the across-recurrence mapping from `tmp[]`

Algorithm 2: Computation of an across-recurrence mapping.

AcrossRecurrenceMapping(M)
Input: A dependency mapping M.
Output: The across-recurrence mapping M' for M.
begin
  // compute the positive transitive closure of M ;
  $m \longleftarrow M^+$;
  $d \longleftarrow \text{domain}(m)$; // domain of the computed closure ;
  $r \longleftarrow \text{range}(m)$; // range of the computed closure ;
  $d' \longleftarrow (d - r)$; // domain of the end-to-end mapping ;
  $r' \longleftarrow (r - d)$; // range of the end-to-end mapping ;
  // restrict the closure to tuples in the end-to-end mapping ;
  $\mathbf{M}' \longleftarrow \{x \rightarrow y \mid x \rightarrow y \in m \wedge x \in d' \wedge y \in r'\}$;
  return $\mathbf{M}'$
end

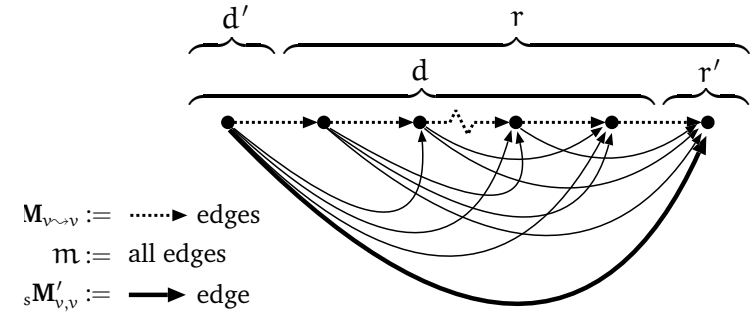


Figure 4.8: The domains and mappings used in Algorithm 2.

to tmp, ${}_{C}\mathbf{M}'_{\texttt{tmp},\texttt{tmp}}$ and use it to compute the transitive dependency mapping over the recurrence from out[] to tmp, ${}_{X}\mathbf{M}^{R}_{\texttt{out},\texttt{tmp}}$. Since there are no intermediate arrays in the cycle, the transitive dependency mapping from tmp[] to itself is given by the only dependency mapping of the statement Y. That is,

$$\begin{aligned} {}_{C}\mathbf{M}_{\texttt{tmp}\rightsquigarrow\texttt{tmp}} &:= {}_{Y}\mathbf{M}_{\texttt{tmp},\texttt{tmp}} \\ &:= \{[k] \rightarrow [k-1] \mid 1 \leq k \leq N \wedge k \in \mathbb{Z}\}. \end{aligned}$$

The positive transitive closure of the above mapping is given by,

$$\begin{aligned} m &:= ({}_{C}\mathbf{M}_{\texttt{tmp}\rightsquigarrow\texttt{tmp}})^{+} \\ &:= \{[k] \rightarrow [\alpha] \mid 0 \leq \alpha < k \leq N \wedge (\alpha, k) \in \mathbb{Z}^2\}. \end{aligned}$$

The domain and range of m are given by,

$$\begin{aligned} d &:= \text{domain}(m) \\ &:= \{[k] \mid 0 < k \leq N \wedge k \in \mathbb{Z}\} \text{ and} \\ r &:= \text{range}(m) \\ &:= \{[\alpha] \mid 0 \leq \alpha < N \wedge \alpha \in \mathbb{Z}\}. \end{aligned}$$

The domain and range of the end-to-end mapping are given by,

$$\begin{aligned} d' &:= (d - r) \\ &:= \{[N] \mid N \in \mathbb{Z}^{+}\} \text{ and} \\ r' &:= (r - d) \\ &:= \{[0]\}. \end{aligned}$$

If we restrict the closure to tuples in the end-to-end mapping, we get the following across-recurrence mapping,

$$\begin{aligned} {}_{Y}\mathbf{M}'_{\texttt{tmp},\texttt{tmp}} &:= (m \backslash d')/r' \\ &:= \{[N] \rightarrow [0] \mid N \in \mathbb{Z}^{+}\}. \end{aligned}$$

Now the transitive dependency mapping over the recurrence from out[] to tmp[] is obtained by,

$$\begin{aligned} {}_{(Z,(Y,1)^{+})}\mathbf{M}^{R}_{\texttt{out},\texttt{tmp}} &:= {}_{Z}\mathbf{M}_{\texttt{out},\texttt{tmp}} \bowtie {}_{Y}\mathbf{M}'_{\texttt{tmp},\texttt{tmp}} \\ &:= \{[0] \rightarrow [N] \mid N \in \mathbb{Z}^{+}\} \bowtie \{[N] \rightarrow [0] \mid N \in \mathbb{Z}^{+}\} \\ &:= \{[0] \rightarrow [0]\}. \end{aligned}$$

From the discussion in Example 5 of program in Figure 4.6(a) it is clear that out[0] transitively depends on tmp[0] and that is what we have obtained by computing the across-recurrence mapping from out[] to tmp[]

along the path $(Z,(Y,1)^+)$. Now the transitive dependency mapping from array `out[]` to array `in[]` can be computed as follows:

$$\begin{aligned} {}_{(Z,(Y,1)^+,X)}\mathbf{M}_{\mathtt{out}\rightsquigarrow\mathtt{in}} &:= {}_{(Z,(Y,1)^+)}\mathbf{M}^{R}_{\mathtt{out},\mathtt{tmp}} \bowtie {}_{X}\mathbf{M}_{\mathtt{tmp},\mathtt{in}} \\ &:= \{[0] \rightarrow [0]\} \bowtie \{[0] \rightarrow [0]\} \\ &:= \{[0] \rightarrow [0]\}. \end{aligned}$$

4.6 Data Dependence Slices

Data dependence paths provide the information regarding the transitive data dependencies from an array to another array in the program. But an array usually depends not just on one array, but on a set of arrays. In this section we extend the inference of data dependence relationships from a given array to the full set of all arrays it depends upon.

As discussed in the previous section, a path in an ADDG fails to be a data dependence path when there is a discontinuation of dependencies at an intermediate array on the path. As was pointed out, this was due to writes to the elements of the array at multiple statements. In terms of the ADDG, the existence of more than one outgoing edge from an array node resulted in some paths not being data dependence paths. From this, it follows that, even within a connected component, an ADDG contains a superposition of *slices* of the program it represents. Program slice is a term that is used to refer to a reduced program that is guaranteed to faithfully represent the original program within the domain of a specified subset of the original program's behavior (Weiser 1981). Similar to what we saw for program representations, different notions of program slicing cater to different applications resulting in a variety of proposals in the literature (Tip 1995). Here we define a notion of program slice as it concerns $\mathscr{D}$ programs for our equivalence checking method in terms of the ADDG of the programs.

We start by recalling that, due to a possible branching of dependencies, a data dependence path originating at a certain source array may guarantee existence of dependencies from only a non-empty subset, or *subdomain*, of its elements to the elements of the destination array. That is, if path p is a data dependence path, in terms of Definition 4.4.1, we

have,

$$\emptyset \subset \text{domain}({}_{p}\mathbf{M}_{v_0 \rightsquigarrow v_n}) \subseteq {}_{s_0}\mathbf{W}_{v_0}.$$

However, the data dependence path defines the *maximal* subset of the definition domain of the source array that is data dependent on the destination array. Therefore, associated with a data dependence path, we always have the largest domain of elements of the source array for which the dependencies hold along the path. We call this the definition domain for the data dependence path or simply, *path definition domain*.

Definition 4.6.1 (Path definition domain, ${}_{p}\mathbf{W}_{v_0 \rightsquigarrow v_n}$)
For a data dependence path p *from* v_0 *to* v_n*, its path definition domain,*

$${}_{p}\mathbf{W}_{v_0 \rightsquigarrow v_n} := \text{domain}({}_{p}\mathbf{M}_{v_0 \rightsquigarrow v_n}).$$

A path definition domain is an extension of the notion of definition domain of a statement to a data dependence path that originates at it. Moreover, a path definition domain is contained in the statement definition domain. In other words, the data dependence path *restricts* the definition domain of the source array to a domain for which it holds. When required, we can further restrict it to a sub-domain of interest to obtain a data dependence path that is restricted to the sub-domain. Also, trivially, the smallest domain that it can be restricted to is for a single element in the path definition domain.

Example 7

Consider the ADDG shown in Figure 4.5 and the path p_1 defined in Example 4. Using the definitions in Example 2, the path definition domain of p_1,

$$\begin{aligned} {}_{p_1}\mathbf{W}_{\mathtt{out} \rightsquigarrow \mathtt{in1}} &:= \text{domain}({}_{p_1}\mathbf{M}_{\mathtt{out} \rightsquigarrow \mathtt{in1}}) \\ &:= \{[i_1, i_2] \mid (\exists\, (k, p) \in \mathbf{D}_Q \mid i_1 = p \wedge i_2 = k - p)\}. \end{aligned}$$

It can be noted that, in this example, the domain ${}_{p_1}\mathbf{W}_{\mathtt{out} \rightsquigarrow \mathtt{in1}}$ coincides with ${}_{Q}\mathbf{W}_{\mathtt{out}}$. That is, the path definition domain is identical to the statement definition domain implying that there is no restriction of the definition domain of the source array due to the data dependence path.

For a given array, we now broaden our data-flow window, from flow due to a single path to flow due to a program slice. A remark here is

that, unlike for paths, for a discussion of slices, it is convenient, and also more meaningful as far as our use of it is concerned, to take the output arrays as source arrays and the input arrays as the destination arrays of the slice.

Given an ADDG and an output array, a slice is a reduced ADDG that contains *all* the nodes and edges in the original ADDG that represent the data dependencies contributing *together* toward the computation of a set of values that are assigned to the output array. An important implication of this is that a slice involves only data dependence paths. Therefore, it is natural that we extend the notion of a data dependence path for an array to a *data dependence slice* for an output array. It is defined as follows.

Definition 4.6.2 (Data Dependence Slice)
Given an ADDG G *of a program and an output array node* v *in* G*, a subgraph* g *of* G *is a data dependence slice iff* g *consists of the edges and nodes in a set* P *of data dependence paths from* v *to input array nodes such that* P *is a maximal set having the property*

$$\bigcap_{\forall \mathrm{p} \in \mathrm{P}} ({}_{\mathrm{p}}\mathbf{W}_{v \rightsquigarrow v_{\mathrm{p}}}) \neq \emptyset,$$

where v_{p} *is the input array node at which a path* p *terminates.*

Simply put, a data dependence slice just contains only that part of an ADDG that defines a complete data-flow with respect to a set of elements of the output array. By requiring the set of paths to be maximal and all paths to only terminate at input array nodes, the definition ensures that there are no missing parts in a data dependence slice. It also ensures that it contains nothing else by requiring that all paths be data dependence paths. Note that this implies that *each* array node in a slice has only one outgoing edge (except in the case when a recurrence is present). Moreover, it implies that the paths define the same set of elements, or sub-domain, of the output array. Therefore, similar to a data dependence path, a data dependence slice is always associated with a largest domain of elements of an output array for which the dependence holds along the slice. This we call as the definition domain for the data dependence slice or simply, *slice definition domain*.

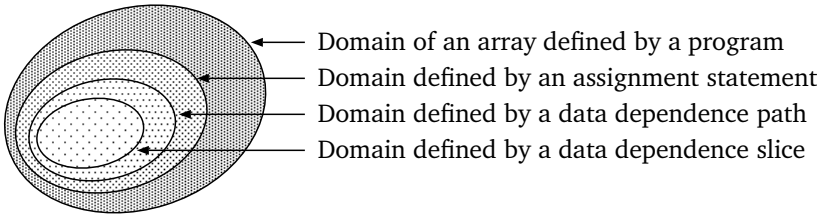


Figure 4.9: The containment relationship of definition domains of an array.

Definition 4.6.3 (Slice definition domain, ${}_{g}\mathbf{W}_{v \rightsquigarrow I_g}$)
For a data dependence slice g *from output array* v *to set* I_g *of input array nodes, its slice definition domain,*

$$ {}_{g}\mathbf{W}_{v \rightsquigarrow I_g} := \bigcap_{p \in g} ({}_{p}\mathbf{W}_{v \rightsquigarrow v_p}), $$

where v_p *is the input array node at which a path* p *terminates*

It is helpful to observe that, as shown in Figure 4.9, the domain of a defined array, that is, either an internal array or an output array, is successively restricted as we move our data-flow focus from the whole program, to a statement, to a path and finally, to a slice.

Example 8

Consider the program in Figure 4.3. Its ADDG shown in Figure 4.5 can be decomposed into two data dependence slices as shown in Figure 4.10.

As can be seen in the ADDG in Figure 4.5, there are two outgoing edges from each of the arrays `out[]` and `tmp[]`. This provides two opportunities for the data-flow in the program to be sliced. But in Example 4, we observed that all the data dependencies along Q entirely branch into P and therefore, paths with Q and R as adjacent edges are not data dependence paths. For a similar reason, paths with S and P as adjacent edges are not data dependence paths either. Therefore, the data-flow of the program is not sliced at array `tmp[]`. This leaves only the opportunity due to definition of the output array `out[]` at two statements viz., Q and S, which slices the data-flow into two, resulting in slice g_1 and slice g_2.

Using the definitions from Example 2, it follows that the slice definition domains of $g_1 := {}_{Q}\mathbf{W}_{out}$ and $g_2 := {}_{S}\mathbf{W}_{out}$.

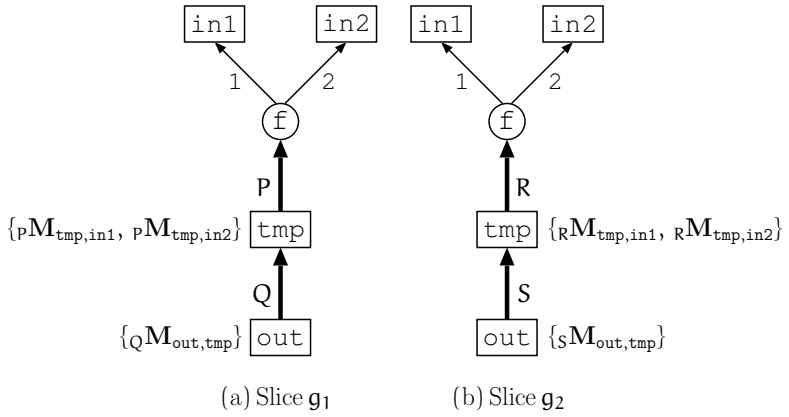


Figure 4.10: The data dependence slices in the ADDG in Figure 4.5.

4.7 Summary

In this chapter we have presented a program representation, called an array data dependence graph (ADDG), that serves our equivalence checking method. The most important aspect of this representation is that it is able to represent sets of data dependencies in a compact form. The notions of data dependence paths and slices were presented in order to reason about the data-flow in the program based on its ADDG representation. The representation, as we shall see in the following chapter, serves as a normal form for representation of the data dependencies under the class of transformations we wish to handle.

Chapter 5

Global Loop and Data-Flow Transformations

5.1 Introduction

This chapter discusses the class of transformations which our method can verify. The method, just as it focuses on the class of $\mathscr{D}$ programs, also focuses on a class of program transformations. The class has two categories of transformations, namely, loop transformations and data-flow transformations, that can be applied on a global or whole-program scope. It is required that the transformed program be obtained starting from the original program by applying a set of instances of transformations from this class. In other words, even though two programs that are input to our method belong to the class of $\mathscr{D}$ programs, it is able to prove their equivalence only when one can be obtained by applying transformations from this class on the other. However, the method is completely oblivious of the transformation process as it allows application of instances of the transformations from the two categories in the class in an unconstrained manner. That is, the designer can apply any number of these transformations, *in any order*, and need not provide our method with any information about even which transformations were applied.

Our discussion of transformations that is based on the two categories of loop and data-flow transformations is in contrast to the related work. For example, in the translation validation work (see Section 2.3.1) a dis-

tinction between *structure* preserving and modifying transformations is made. This distinction is based on whether there exists a clear mapping between *control points* in the original and the transformed programs, respectively. We focus exclusively on the data-flow, and it implies that the classification we follow and theirs are incomparable.

We first discuss loop transformations in Section 5.2, followed by data-flow transformations in Section 5.3. In discussing the transformations, we are not concerned with either the reasons for applying them or the gains that they offer the designer, as the motivation for their application has already been discussed in Sections 1.5.2 and 1.5.3. Instead our focus here will be on what *effect* they have on a $\mathscr{D}$ program and how it is reflected in the ADDG representation of the program. As we will discuss, given that we are only interested in the computation in the slices and the output-input mappings for its paths, correctly applied transformations affect the ADDG, if at all, only locally. In such cases, we can call upon some operations in order to normalize the ADDG within a given scope. We discuss them in Section 5.4.

5.2 Loop Transformations (LTs)

Loop transformations are one of the most widely studied of program transformations, and naturally, they come in a great variety. But from our program representation point of view, they can be distinguished based on the effect they have on the data dependence slices in the original program. Let us recall that slices group together sets of elements of the output variables that are assigned collectively by the data-flow. The loop transformations reorder the execution of assignment statements, that is, the control-flow of the program, while it leaves the data-flow as it was in the original program. Given that the schedule of statements is valid for $\mathscr{D}$ programs, this means that the set of elements of the variables read and written, and the dependency between them remains unaltered by the transformation. Then what remains, as far as slices in the original program are concerned, is the grouping of the elements of the output arrays. Loop transformations may either leave the grouping unchanged or regroup the sets of outputs by partitioning them differently. We call the former, *slice preserving* and the latter, *slice repartitioning* loop transformations. We discuss them separately.

5.2.1 Slice Preserving LTs

Under the slice preserving loop transformations, both original and the transformed programs have the same grouping of output elements, that is, identical slice definition domains (see Definition 4.6.3). Therefore, these transformations may (1) transform the iteration domains of some or all of the statements; and/or (2) given that two statements have identical iteration domains, either share or duplicate the iteration domain between them. In the first case, we note that for the data-flow to remain as before, a transformation of the iteration domain may have to be accounted for by an inverse transformation on the subscripts of the arrays that are referenced. In the second case, as far as the data-flow is concerned, if the resulting schedule is valid, the sharing or duplication of iteration domains is of no consequence. This observation implies, as we will discuss, that distinguishing slice preserving loop transformations is incomparable to the other common classifications. Nevertheless, it helps to recall the well-known transformations in the literature that belong to this category and illustrate their effect on an example program.

We start by taking a very brief look at the loop transformation theory and implementation. Loop transformations have traditionally been automated using matrix representations for the constraint system that follows for each assignment statement from iteration domain bounds, conditionals on iterators and the index expressions of array references in the statement. Then a required transformation or a set of transformations can also be represented by a matrix, called the *transformation matrix*, that is used to transform a given constraint system into another, resulting in the transformed program.

Loop transformations are often distinguished based on the nature of the transformation matrix. When the matrix is a so-called *unimodular* matrix (its determinant is ± 1), the transformation/s are called unimodular transformations for perfectly nested loops. They include loop reversal, interchange (or permutation, in general), skewing and bumping (or normalization) transformations, or a combination derived from them. All these transformations are slice preserving as they may only transform the iteration domains. For skewing and bumping transformations, the index expressions of any input or output array will have to be transformed to account for the change in the iteration domain within the statement. Note that, however, this is not necessary for internal arrays

```
for (k = 0; k <= 2*N-2; k++)
    tmp[k] = f(in1[k], in2[3*k]);        // X

for (i = 0; i <= N-1; i++)
    for (j = 0; j <= N-1; j++)
        out[i][j] = tmp[i+j];            // Y
```

Figure 5.1: The simple program from Figure 4.1.

as it also suffices if the reads are also adjusted depending on the writes, and thus ensure the original data-flow.

There are other well-known loop transformations that cannot be cast as unimodular transformations as they transform the volume of the iteration domains. Loop distribution, fusion, tiling (or blocking) and strip-mining transformations belong to this category. The transformation of the volume of the iteration domain does not effect the slice definition domains and hence, these transformations too are slice preserving.

Invariant property

The invariant property for a slice preserving loop transformation is that the ADDGs of the original and the transformed programs have identical output-input mappings for corresponding data dependence paths.

Example 9

Consider the program in Figure 5.1. Its ADDG is as shown Figure 5.2 on the left. It has only one data dependence slice, which is as shown to its right.

Figure 5.3 shows versions of the program obtained after applying unimodular transformations. Since they are slice preserving, the slice in the ADDG of the transformed program has the same slice definition domain as the slice in the ADDG of the original program. Therefore, they all have the same slice as shown in Figure 5.2.

Now consider the program in Figure 5.4(a), and its two transformed versions obtained after applying non-unimodular transformations like loop distribution and tiling. We have discussed earlier that these two transformations are also slice preserving transformations. Therefore, as we would expect, the slices in the ADDGs of versions (b) and (c) have the same definition domains as version (a). The slices are as shown in Figure 5.5.

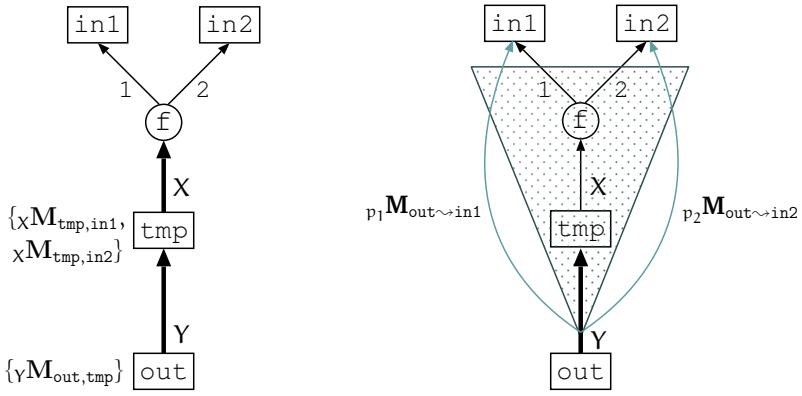


Figure 5.2: To the left, the ADDG of the program in Figure 5.1 and to the right, its only data dependence slice.

5.2.2 Slice Repartitioning LTs

Under the slice repartitioning loop transformations, for some of the output arrays, the slice definition domains in the original program may be either divided or merged or repartitioned in the transformed program. Transformations like loop fission, merging, folding, peeling, splitting and unrolling belong to this category.

Invariant property

The invariant property for a slice repartitioning loop transformation is that the ADDGs of the original and the transformed programs have identical *union* of output-input mappings for corresponding *sets* of data dependence paths.

Example 10

Consider again the program in Figure 5.1 and its ADDG and its only slice as shown Figure 5.2.

Figure 5.6 shows versions of the program obtained after applying slice repartitioning loop transformations. The ADDGs of the transformed versions are as shown in Figure 5.7 and the slices in them are as shown in Figure 5.8. The slices in the transformed ADDGs show a repartitioning of the definition domains of the output array `out[]`.

As another instance of slice repartitioning, consider the program in Figure 4.3, whose ADDG has two slices as shown in Figure 5.5. A loop merging transformation of this program as shown in Figure 5.9 combines the

```
for (k = 0; k <= 2*N-2; k++)
    tmp[k] = f(in1[k], in2[3*k]);        // X
for (j = 0; j <= N-1; j++)
    for (i = 0; i <= N-1; i++)
        out[i][j] = tmp[i+j];            // Y
```

(a) Loop interchange transformation

```
for (k = 2*N-2; k >= 0; k--)
    tmp[k] = f(in1[k], in2[3*k]);        // X
for (i = 0; i <= N-1; i++)
    for (j = 0; j <= N-1; j++)
        out[i][j] = tmp[i+j];            // Y
```

(b) Loop reversal transformation

```
for (k = 0; k <= 2*N-2; k++)
    tmp[k] = f(in1[k], in2[3*k]);        // X
for (i = M; i <= M+N-1; i++)
    for (j = 0; j <= N-1; j++)
        out[i-M][j] = tmp[i+j-M];        // Y
```

(c) Loop bumping transformation

```
for (k = 0; k <= 2*N-2; k++)
    tmp[k] = f(in1[k], in2[3*k]);        // X
for (i = 0; i <= N-1; i++)
    for (j = i; j <= i+N-1; j++)
        out[i][j-i] = tmp[j];            // Y
```

(d) Loop skewing transformation

Figure 5.3: Unimodular transformations on the program in Figure 5.1.

```
for (k = 0; k <= 2*N-2; k++)
    if (k < N){
       tmp[k] = f(in1[k], in2[3*k]);     // P
       for (p = 0; p <= k; p++)
           out[p][k-p] = tmp[k];         // Q
    } else {
       tmp[k] = f(in1[k], in2[3*k]);     // R
       for (p = k-N+1; p <= N-1; p++)
           out[p][k-p] = tmp[k];         // S
    }
```

(a) The program from Figure 4.3

```
for (k = 0; k <= 2*N-2; k++)
    if (k < N){
       tmp[k] = f(in1[k], in2[3*k]);     // P
       for (p = 0; p <= k; p++)
           out[p][k-p] = tmp[k];         // Q
    }
for (k = 0; k <= 2*N-2; k++)
    if (k >= N){
       tmp[k] = f(in1[k], in2[3*k]);     // R
       for (p = k-N+1; p <= N-1; p++)
           out[p][k-p] = tmp[k];         // S
    }
```

(b) Loop distribution transformation on program in (a)

```
for (k = 0; k <= 2*N-2; k++)
    if (k < N){
       tmp[k] = f(in1[k], in2[3*k]);     // P
       for (r = 0; r <= k; r+=8)
           for (p = r; p <= min(r+7, k); p++)
               out[p][k-p] = tmp[k];     // Q
    } else {
       tmp[k] = f(in1[k], in2[3*k]);     // R
       for (p = k-N+1; p <= N-1; p++)
           out[p][k-p] = tmp[k];         // S
    }
```

(c) Loop tiling transformation on program in (a)

Figure 5.4: Examples for non-unimodular, but slice preserving, loop transformations.

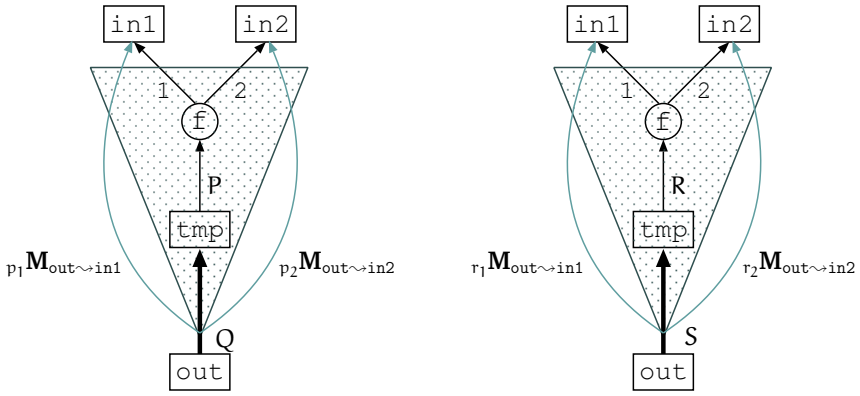


Figure 5.5: The two slices in the ADDG of the program in Figure 5.4(a), and also for programs (b) and (c) in the same figure.

two definition domains into one, thus having a single slice in its ADDG which is identical to the slice shown in Figure 5.2.

5.3 Data-Flow Transformations

Under the loop transformations, as we have seen in the previous section, both the original and the transformed programs have identical sets of assignment statements except for a possible difference in the index expressions of their arrays. In contrast, the data-flow transformations may either introduce new statements or eliminate some existing ones. Moreover, they may also transform the right-hand side expression of the statements. We distinguish between the transformations that exploit the algebraic properties of the operators in the data-flow, called the *algebraic transformations*, and the ones that do not, but introduce or eliminate intermediate copies in the data-flow, called the *expression propagations*. Owing to the influence of the context in which our verification problem was identified, i.e., the DTSE methodology, and also to reflect the evolution of our method, we separate out a specific but important case of the latter transformations where only copies are introduced in the data-flow, called the *data-reuse transformations*.

```
for (k = 0; k <= 2*N-2; k+=2){
    tmp[k] = f(in1[k], in2[3*k]);        // X1
    tmp[k+1] = f(in1[k+1], in2[3*k+3]); // X2
}
for (i = 0; i <= N-1; i++)
    for (j = 0; j <= N-1; j++)
        out[i][j] = tmp[i+j];            // Y
```

(a) Loop unrolling transformation

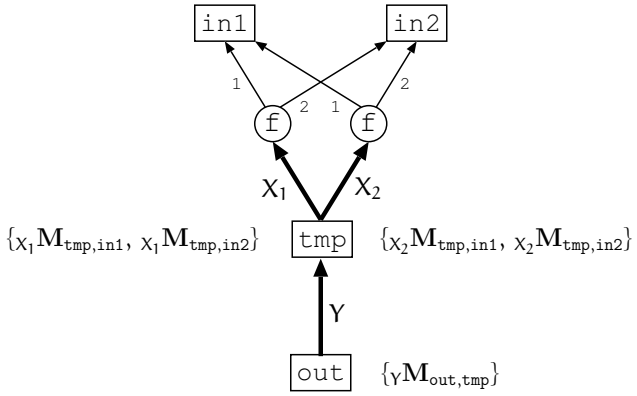
```
for (k = 0; k <= 2*N-2; k++)
    tmp[k] = f(in1[k], in2[3*k]);        // X
for (i = 0; i <= N-1; i++){
    for (j = 0; j <= M; j++)
        out[i][j] = tmp[i+j];            // Y1
    for (j = M+1; j <= N-1; j++)
        out[i][j] = tmp[i+j];            // Y2
}
```

(b) Loop splitting transformation

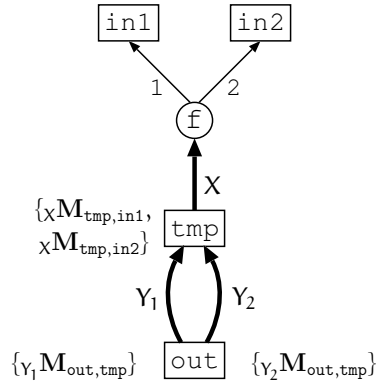
```
tmp[0] = f(in1[0], in2[0]);              // X0
for (k = 1; k <= 2*N-2; k++)
    tmp[k] = f(in1[k], in2[3*k]);        // X
for (i = 0; i <= N-1; i++)
    for (j = 0; j <= N-1; j++)
        out[i][j] = tmp[i+j];            // Y
```

(c) Loop peeling transformation

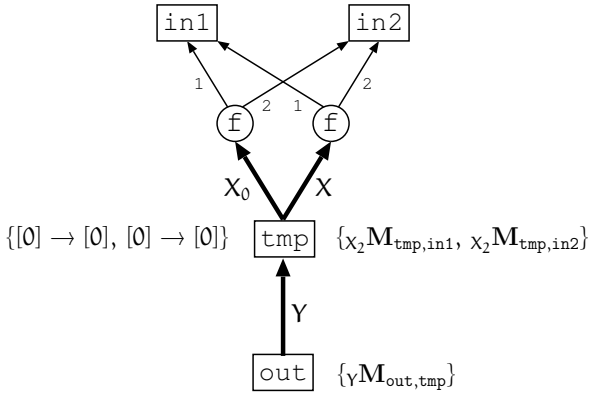
Figure 5.6: Slice repartitioning transformations on the program in Figure 5.1.



(a) The ADDG after loop unrolling

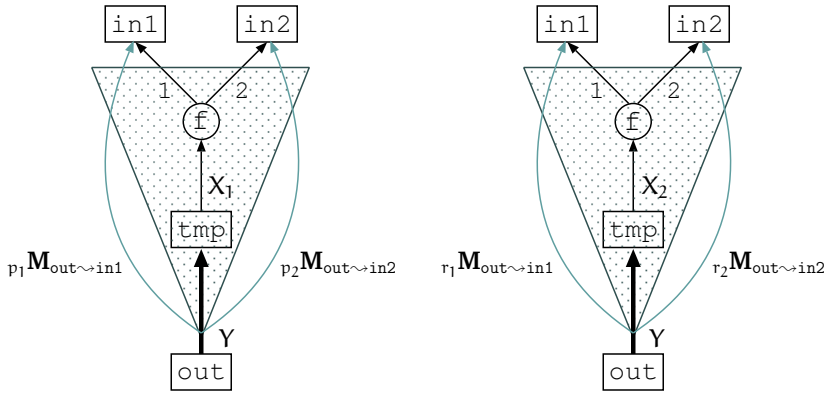


(b) The ADDG after loop splitting

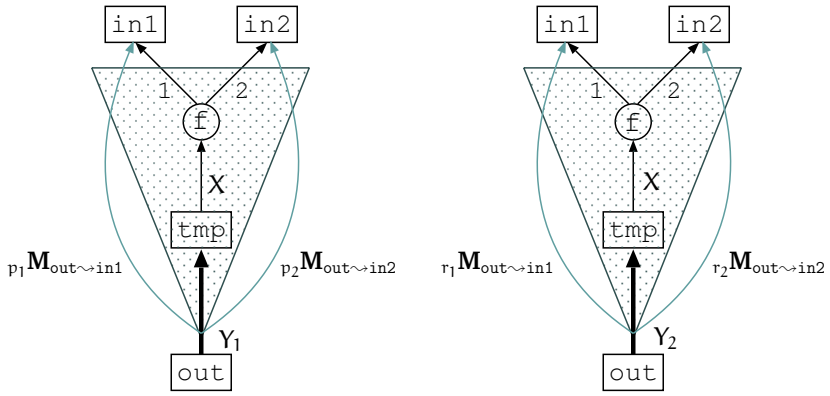


(c) The ADDG after loop peeling

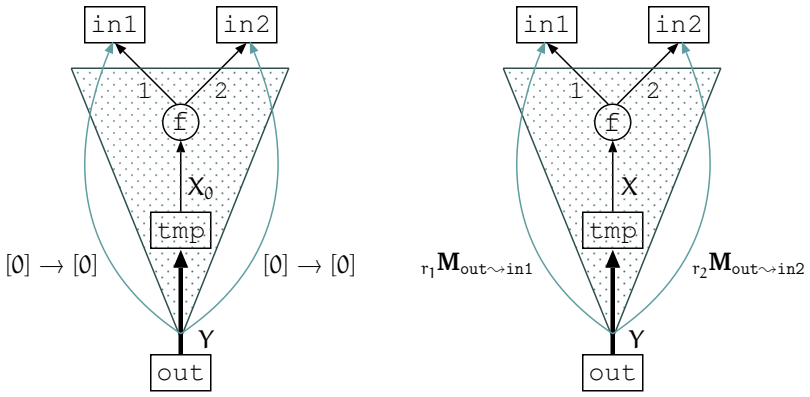
Figure 5.7: The ADDGs of the transformed programs in Figure 5.6.



(a) Slices in the ADDG after loop unrolling



(b) Slices in the ADDG after loop splitting



(c) Slices in the ADDG after loop peeling

Figure 5.8: The slices in the ADDGs in Figure 5.7.

```
for (k = 0; k <= 2*N-2; k++){
  tmp[k] = f(in1[k], in2[3*k]);              // X

  for (p = max(0,k-N+1); p <= min(k,N-1); p++)
     out[p][k-p] = tmp[k];                   // Y
}
```

Figure 5.9: Slice repartitioning by loop merging transformation on program in Figure 5.4(a).

5.3.1 Data-Reuse Transformations

The data-reuse transformation involves the introduction of an intermediate array, called the *buffer*, to hold the data elements that are accessed multiple times from the memory. This principle is shown in Figure 5.10. The effect of the transformation on the ADDG of the original program directly follows from this. An extra internal array node is inserted on the path immediately following the array node from which a copy is made. Since we are dealing with $\mathscr{D}$ programs, the single-assignment form implies that any element of the newly introduced array is also assigned only once. Also, introduction of buffers may require that new loops be added in order to facilitate the copy operation. But this does not effect the slices of the ADDG in that sense that they do not entail any split or merge of the dependence paths.

Invariant property

The invariant property for a data-reuse transformation is the same as for a slice preserving loop transformation, that is, the ADDGs of the original and the transformed programs have identical output-input mappings for corresponding data dependence paths.

Example 11

Consider the original and transformed programs in Figure 5.11. The multiple reads to the same element of the input array, `in1[]`, is avoided by the data-reuse transformation by introducing a copy to an intermediate scalar, `buf`. In order that the transformed program have single-assignment form we have converted `buf` into an array.

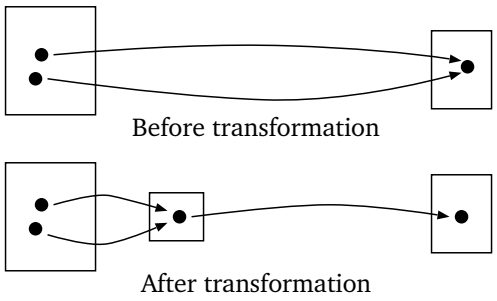


Figure 5.10: The principle of data reuse transformation

The ADDGs of the two versions are shown in Figures 5.12(a) and (b), along with their slices. As can be seen, the only difference in the transformed ADDG is the insertion of an internal array node.

5.3.2 Expression Propagations

Expression propagations fully generalize the data-reuse transformations. They involve both introduction and elimination of intermediate arrays for partial computations in the program function. For example, a statement with a summation of three terms on the right-hand side can be converted into two statements with summation of two terms each, by the introduction of an intermediate array. Expression propagations help realize such classical compiler optimizations as common sub-expression elimination, invariant code motion, dead code elimination, etc., for array-intensive programs. The effect of expression propagation on the ADDG of the program function is insertion or elimination of array nodes on the paths of the ADDG. This implies that slices in the original program are preserved in the transformed program.

Invariant property

The invariant property for expression propagation is the same as for a slice preserving loop transformation, that is, the ADDGs of the original and the transformed programs have identical output-input mappings for corresponding data dependence paths.

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        out[i][j] = f(in1[i], in2[i][j]);   // Y
```

(a) Original version

```
for (i = 0; i < N; i++){
    buf[i] = in1[i];                        // X
    for (j = 0; j < N; j++)
        out[i][j] = f(buf[i], in2[i][j]);   // Y
}
```

(b) Transformed version

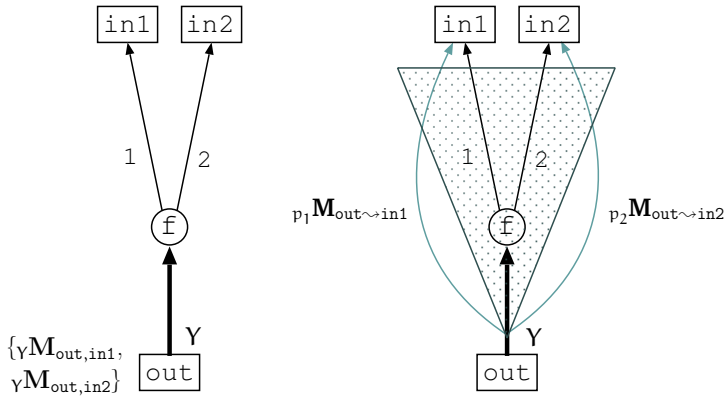
Figure 5.11: A simple example to illustrate the data reuse transformation.

Example 12

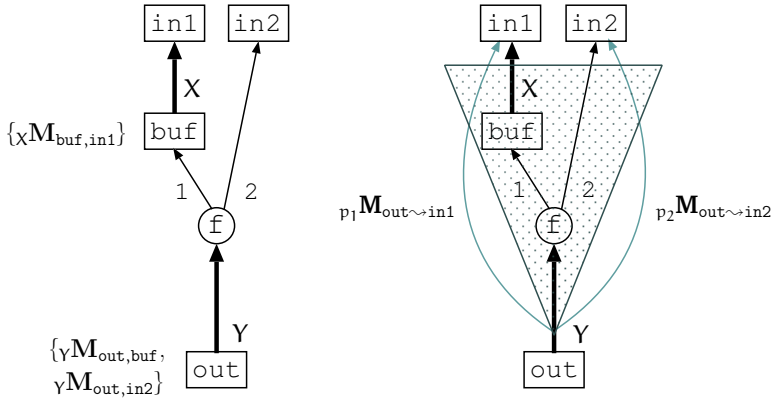
Consider the original and transformed programs in Figure 5.13. The ADDGs of the two versions are shown in Figures 5.14(a) and (b), along with their slices. In this example, the only difference in the transformed ADDG is the insertion of internal array nodes.

5.3.3 Algebraic Transformations

Algebraic data-flow transformations take advantage of the properties of the operators or user-defined functions and modify the data-flow such that the semantics of the original function are preserved (modulo overflow). The algebraic transformations are not restricted to the expression in a statement, but can have a global scope. The designers, based on the knowledge of the overall computation, are often able to apply algorithmic or big-step transformations. Typically, most of such global transformations just rely on the associativity and/or commutativity properties of the operators like addition and multiplication on a fixed-point datatype like integer. Hence in what follows, we restrict our discussion to only these transformations. Other algebraic properties related to identity, inverse, distributivity and evaluation of constants are less common in practice and can be handled in a way similar to what we present.



(a) The ADDG of the original version and its only slice



(b) The ADDG of the transformed version and its only slice

Figure 5.12: The ADDGs of the program versions in Figure 5.11.

```
for (i=0; i<M; i++)
    for (j=1; j<N; j++)
        out[i][j] = f(in1[i+2][j-1])
                              + g(in2[i+1]);   // Z
```

(a) Original version

```
for (i=0; i<M; i++){
    tmp[i] = g(in2[i+1]);                      // X
    for (j=1; j<N; j++){
        buf[i][j] = f(in1[i+2][j-1]);          // Y
        out[i][j] = buf[i][j] + tmp[i];        // Z
    }
}
```

(b) Transformed version

Figure 5.13: A simple example to illustrate the expression propagation transformation.

The effect of such algebraic transformations on an ADDG is shown in Figure 5.15, where it is assumed that the operator $\oplus$ is associative, $\otimes$ is commutative and $\circledast$ is both commutative and associative.

Invariant property

The invariant property for algebraic transformations is that there exists a *normal form* for the ADDGs of the original and the transformed programs such that corresponding data dependence paths having identical output-input mappings can be identified (or matched) between them.

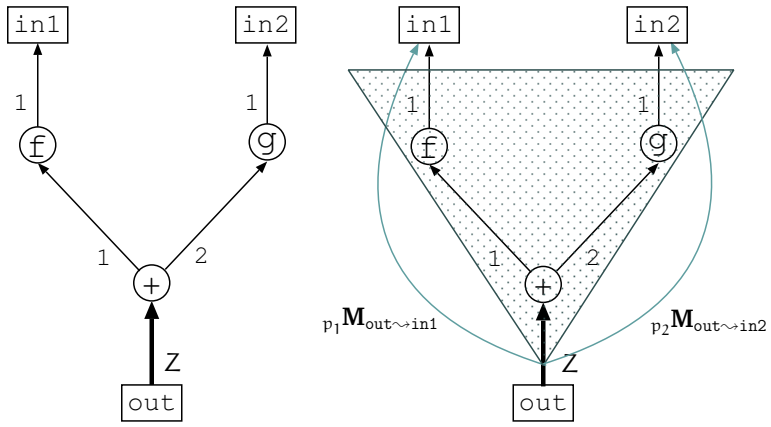
Example 13

Consider the original and transformed programs in Figure 5.16. The original program computes

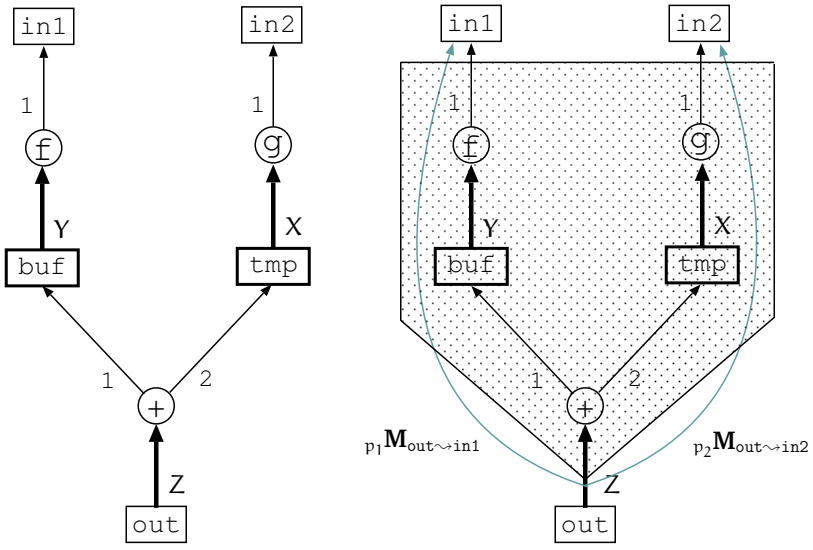
$\forall$ k $\in$ [0...N−1] : out[k] = in2[2*k] + in2[k] + in1[2*k] + in1[k],

while the transformed program computes

$\forall$ k $\in$ [0...N−1] : out[k] = in1[k] + in2[k] + in1[2*k] + in2[2*k].

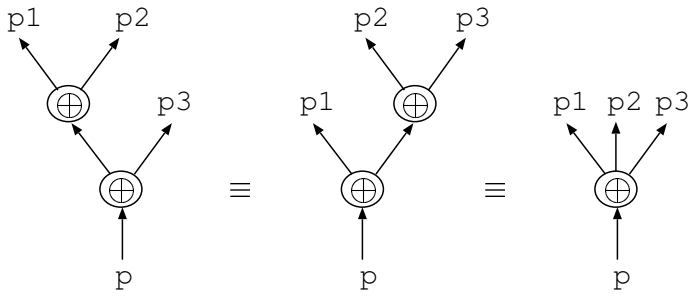


(a) The ADDG of the original version and its only slice

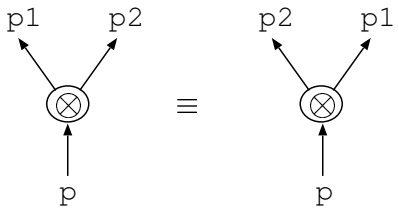


(b) The ADDG of the transformed version and its only slice

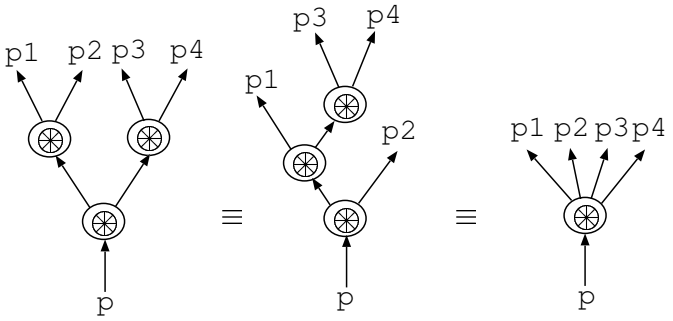
Figure 5.14: The ADDGs of the program versions in Figure 5.13.



(a) *Associative transformation*: Its effect is that the end-nodes are regrouped with respect to the chain of $\oplus$-nodes (the *associative chain*), while maintaining their order.



(b) *Commutative transformation*: Its effect is that the positions of the outgoing edges of the $\otimes$-node may be permuted.



(c) *Combined associative and commutative (AC) transformation*: Its effect is to maintain the same end-nodes, but with any possible tree of $\circledast$-nodes between them and the root $\circledast$-node.

Figure 5.15: Effect of basic algebraic transformations on an ADDG.

```
for (k = 0; k < N; k++)
    tmp[k] = in2[2*k] + in2[k];              // X
for (k = 0; k < N; k++)
    buf[2*k] = in1[2*k] + in1[k];            // Y
for (k = 0; k < N; k++)
    out[k] = tmp[k] + buf[2*k];              // Z
```

(a) Original version

```
for (k = 0; k < N; k++)
    buf[k] = in1[k] + in2[k];                // P
for (k = N; k <= 2*N-2; k += 2)
    buf[k] = in1[k] + in2[k];                // Q
for (k = 0; k < N; k++)
    out[k] = buf[k] + buf[2*k];              // R
```

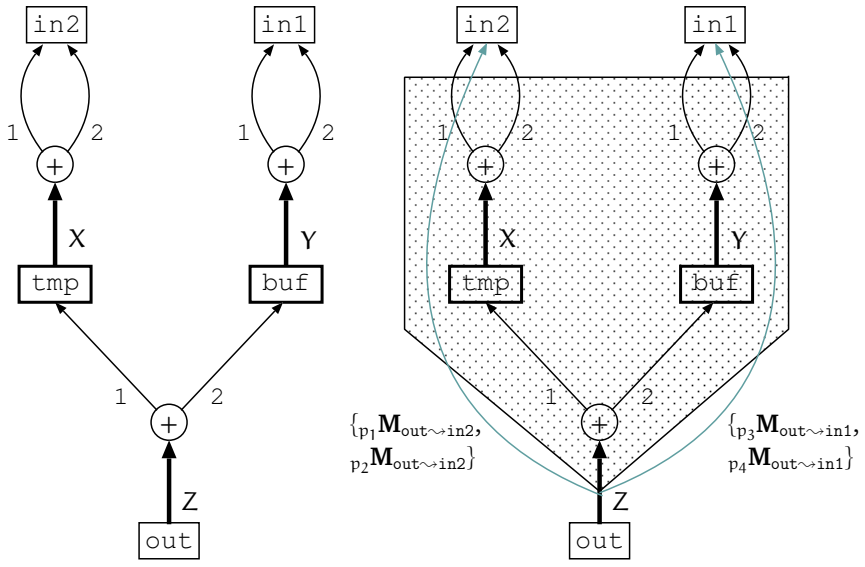
(b) Transformed version

Figure 5.16: A simple example to illustrate algebraic transformations.

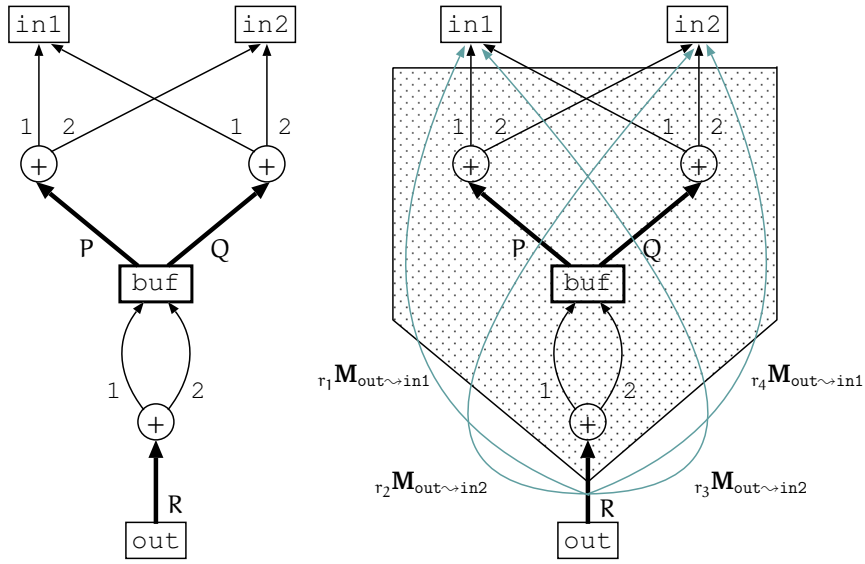
The ADDGs of the two versions are shown in Figures 5.17(a) and (b), along with their slices. As can be seen, the transformed ADDG is the result of several applications of the basic algebraic transformations discussed above. Here, the transformations are motivated by the observation that the transformed program performs $N/2$ (i.e., $3N-5N/2$) integer additions less when compared to the original program.

5.4 Operations for ADDG Normalization

In this section, we present two operations that help normalize an ADDG with respect to a given data dependence path. These normalizing operations are used on a demand-driven basis by our equivalence checking method. A remark here is that the operations return the information obtained by normalization and do not manipulate the internal data-structure representing the ADDG itself.



(a) ADDG of the original version and its only slice



(b) ADDG of the transformed version and its only slice

Figure 5.17: The ADDGs of the program versions in Figure 5.16.

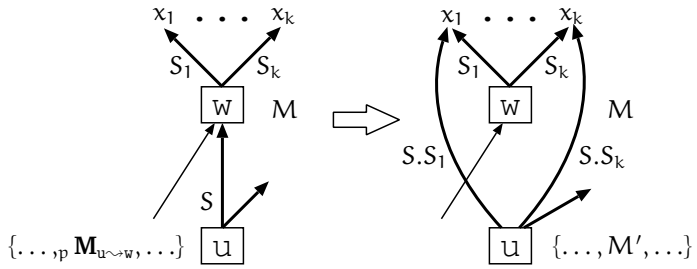
5.4.1 Internal Array Node Elimination

An internal array stores values for a later read, but otherwise has no influence on the actual computation in the program function. In its absence, the values would be directly available where they are required without being written into a store. Given a data dependence path in a $\mathscr{D}$ program, it is possible to eliminate it and update the dependency mappings of its predecessor array node. Elimination of an internal array from a $\mathscr{D}$ program, or *internal array node elimination* in an ADDG, with respect to a given data dependence path, is required as a primitive in our method, and hence we define it as an operation that we call upon whenever required. It is given by Algorithm 3 and it is illustrated in Figure 5.18.

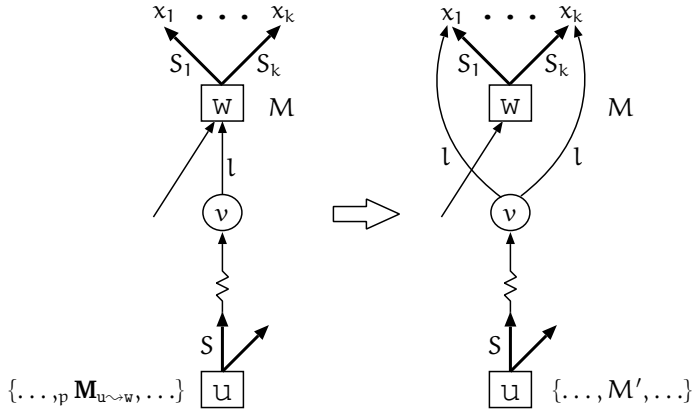
Algorithm 3: Internal array node elimination.

EliminateArrayNode$(G, w, {}_{p}\mathbf{M}_{out \rightsquigarrow w})$
Input: An ADDG G, an internal array node w, the transitive dependency mapping to w along a path p from output array node out.
Output: A list of successors as 3-tuples, where each tuple consists of a successor node, a label representing the path from the parent node of w to the successor of w, and the dependency mapping along this path.

```
begin
    v ⟵ parent node of w along p;
    l ⟵ label of (v, w) along p;
    u ⟵ most recent array node that precedes w along p;
    S ⟵ label of the outgoing edge of u along p;
    M ⟵ set of dependency mappings annotated at w;
    X ⟵ set of successor nodes of w;
    succList ⟵ ∅;
    foreach x ∈ X do
        S′ ⟵ label of (w, x);
        label ⟵ if u = v then S.S′ else l ;
        if range(pM_out⇝w) ∩ S′W_w ≠ ∅ then
            foreach m ∈ M such that m is due to S′ do
                succList ⟵ succList ∪{(x, label, m)};
    return succList
end
```



(a) For the case $u = v$



(b) For the case $u \neq v$

Figure 5.18: Illustration of the array node elimination operation.

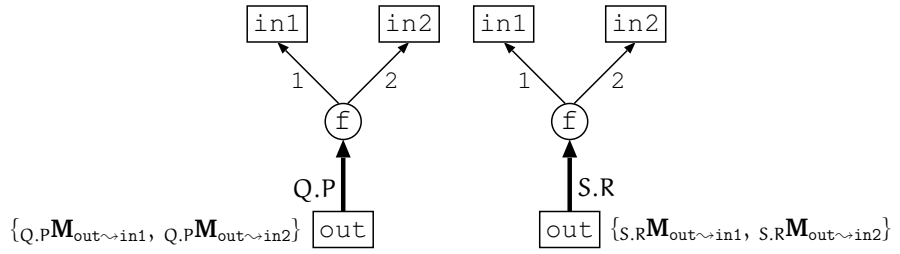


Figure 5.19: The data dependence slices in Figure 4.10 after elimination of `tmp[]`.

Example 14

Consider the slices shown in Figure 5.5. Figure 5.19 shows the slices after internal array node `tmp[]` has been eliminated on each of the four paths, viz., p_1, p_2, r_1 and r_2.

5.4.2 Flattening of an Associative Chain

Suppose that an associative operator ($\oplus$) is reached on an ADDG. The flattening operation, described in Algorithm 4, involves a lookahead traversal of the sub-ADDG of the associative chain rooted at the $\oplus$-node. It constructs a set of tuples, where in each tuple the first component is an ordered list of successor nodes after flattening and the second component is a domain that is a subset of the (input) domain for which the flattening holds. The algorithm processes each of the operand nodes flattening them one by one maintaining the ordering of the successors which preserves their argument position after flattening. This requires that we process the nodes in the rightmost-first order, so that the position of the unprocessed elements remains intact (notice the loop (n **downto** 1) in the algorithm). The effect of flattening is that it brings all operands of the chain to the same level as successor nodes of the root $\oplus$-node. Any intermediate array nodes that exist on the path between the nodes in the list and the root $\oplus$-node are eliminated by (mutually) recursively calling Algorithm 5. Fig. 5.20 illustrates this. Note that a singleton is returned when no internal arrays are involved, and more elements appear when internal array nodes have writes to them from multiple statements. The design of the flattening operation in terms of Algorithms 4 and 5 is motivated by the fact that it is cleaner to separate the concerns that arise from the presence of associative operator

nodes and the array nodes in the associative chain. In the former, it is required to gather the successors of the identical successor associative operator nodes, and in the latter, it is required to distribute the entire set of computed successor operands among the different writes to the array node.

Algorithm 4: Flattening an associative chain.

```
FlattenAsso(G, v, op, path, dom)
Input: An ADDG G, the node v to be flattened, the associative
       operator op, the path path from out to v, and the domain
       of the mapping between out and v on the path.
Output: A set of tuples, where in each tuple the first component
        is an ordered list of successor nodes after flattening and
        the second component is a domain that is a subset of the
        (input) dom for which the flattening holds.
begin
    map ⟵ the dependency mapping along path restricted to
    domain dom;
    succ ⟵ the ordered list of successors of v;
    n ⟵ |succ|;
    S ⟵ {(succ, Domain(map))};
    for i ⟵ n downto 1 do
        succ_i ⟵ the ith element in succ;
        if succ_i is an operator node different from op or an input
        array node then
            S' ⟵ S;
        else
            path' ⟵ path extended with (v, succ_i);
            if succ_i is an operator node equal to op then
                T ⟵ FlattenAsso(G, succ_i, op, path', dom);
            else
                T ⟵ FlattenArr(G, succ_i, op, path', dom);
            S' ⟵ ∅;
            foreach (s', d') ∈ T do
                foreach (s, d) ∈ S do
                    S' ⟵ S' ∪ {(UpdateList(s, i, s'), d')};
        S ⟵ S'
    return S
end
```

Algorithm 5: Flattening array nodes appearing in an associative chain or in the successors of a commutative node.

FlattenArr$(G, v, op, path, dom)$
Input: An ADDG G, the node v to be flattened, the associative operator op, the path $path$ from out to v, and the domain of the mapping between out and v on the $path$.
Output: A set of tuples, where in each tuple the first component is a list of successor nodes after flattening and the second component is a domain that is a subset of (input) dom for which the flattening holds.

```
begin
    map ⟵ the dependency mapping along path restricted to
    domain dom;
    succ ⟵ the set of successors of v;
    S ⟵ ∅;
    while succ ≠ ∅ do
        Select and remove node from succ;
        D ⟵ the domain of the assignment that labels (v, node);
        ndom ⟵ Domain(RestrictRange(map, D));
        if node is an operator node different from op or an input
        array node then
            S ⟵ S ∪ {([node], ndom)};
        else
            path' ⟵ path extended with (v, node);
            if node is an operator node equal to op then
                T ⟵ FlattenAsso(G, node, op, path', ndom);
            else
                T ⟵ FlattenArr(G, node, op, path', ndom);
            S ⟵ S ∪ T;
    return S
end
```

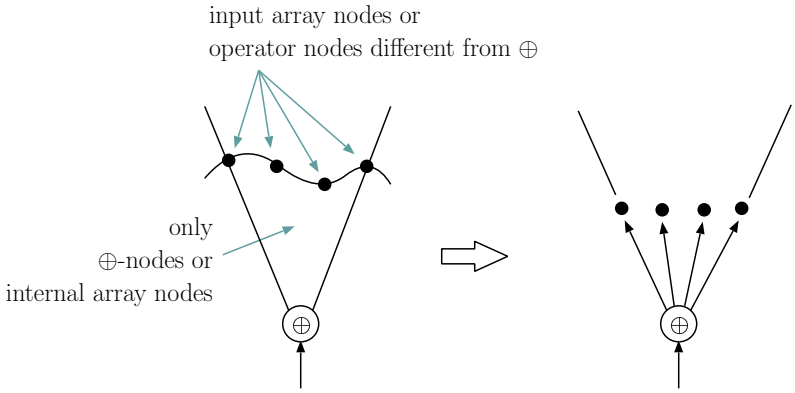


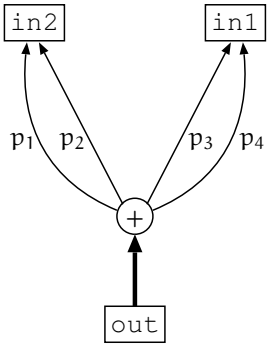
Figure 5.20: Illustration of the flattening operation.

Example 15

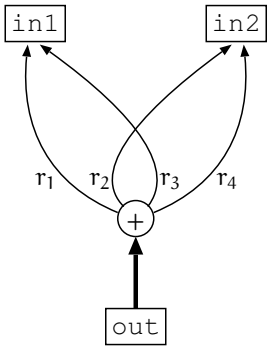
Consider the slices shown in Figure 5.17 of two program versions under associative and commutative transformations. Figure 5.21 shows the slices resulting from applying flattening on each of them.

5.5 Summary

In this chapter we have discussed global loop and data-flow transformations, the two categories of transformations that include the different possible instances of transformations that our method handles. We have shown the effect these transformations can have on the ADDG representation and discussed their invariant properties under the transformations. Establishing corresponding data dependence paths between the ADDGs of the two programs requires local normalizations on the individual transformations in the form of internal array node elimination and flattening. We have presented both of these operations.



(a) ADDG of the original version after flattening



(b) ADDG of the transformed version after flattening

Figure 5.21: The ADDGs shown in Figure 5.17 after flattening.

Chapter 6

Statement-Level Equivalence Checking Methods

6.1 Introduction

Central to an equivalence checking method that is compositional is the task of establishing points of correspondence between the original and the transformed programs. Depending on the transformations that have been applied on the original program, the *gap* between the two programs can vary, and accordingly, establishing points of the correspondence between them can vary in difficulty. Therefore, the amount of work that an equivalence checking method has to do depends on the categories of transformations and their combinations that it allows. From the discussion in the previous chapter, it follows that, for individual categories, it is apparent that it suffices to restrict the equivalence checking method to a level of scope that is just enough to establish the correspondence between the two programs. Based on this intuition, in this chapter we keep the scope of equivalence checking to the level of assignment statements and discuss how, not only loop transformations, but also their combination with data-reuse transformations can be verified. We first recall the equivalence checking method that allows only loop transformations (Section 6.2). It was developed by Samsom (1995). Next we discuss an efficient equivalence checking method that allows a combination of loop and data-reuse transformations (Section 6.3). Before

summarizing the chapter, we discuss the limitations that follow from restricting to the statement-level scope (Section 6.4) and thereby set the scene for a general method to be discussed in the next chapter.

6.2 Verification of Only Loop Transformations

Samsom (1995) has proposed an equivalence checking method for the full category of loop transformations. It is restricted in its scope only to the level of assignment statements. In this section, we provide a summary of his method. We discuss first the program representation that is used (Section 6.2.1), then the sufficient condition for equivalence of such representations (Section 6.2.2) and finally, the equivalence checking method that follows (Section 6.2.3).

6.2.1 Constrained Expressions (CEs)

Samsom uses a coarse program representation that collected information regarding each of the assignment statements taken as a whole. In his representation, each statement in the program is represented by, what he called a, *constrained expression* that comprises of an *expression* under a *precondition*. The expression is simply the *signature* of the entire statement. The precondition refers to the set of data dependencies between the elements of the defined array and the elements of the operand arrays and it is captured in the form of an integer domain. The integer domain is defined by the values of the indices of all the variables (defined and operand arrays) in the statement for all its instantiations. The domain has as many dimensions as the sum of the dimensions of the individual arrays where each dimension stands for an index function of the iterators controlling the statement and the iterators are defined by the iteration domain of the statement. It is not hard to see the relationship between the precondition and the geometrical model that we have already discussed for our representation in Section 4.2. We will elaborate on this in Section 6.3.1.

Consider the template code shown in Figure 4.2 in page 48. If the statement S is replaced by the statement in Figure 6.1, its constrained expression is defined as follows.

$$S: \quad v[f_{i_1}(\vec{k})]\ldots[f_{i_n}(\vec{k})] \;=\; exp(u_1[f_{j_{11}}(\vec{k})]\ldots[f_{j_{1m_1}}(\vec{k})], \ldots, u_k[f_{j_{k1}}(\vec{k})]\ldots[f_{j_{km_k}}(\vec{k})]);$$

Figure 6.1: Template statement in a $\mathscr{D}$ program.

Definition 6.2.1 (Constrained expression, C_S)
The constrained expression of a statement is a tuple consisting of a precondition and an expression. The precondition is an integer domain such that each point identifies the exact element of each array accessed by an execution instance of the statement. The expression is a signature that is obtained by replacing all the index functions of its arrays by distinct identifiers. For a template statement S, as shown in Figure 6.1, having an iteration domain **D***, its constrained expression is given by,*

$$\mathbf{C}_S := (\mathbf{P}_S, \mathbf{E}_S),$$

where,

$$\mathbf{P}_S := \{[d_1,\ldots,d_n,d_{11},\ldots,d_{1m_1},\ldots,d_{k1},\ldots,d_{km_k}] \mid (\exists\,\vec{k}\in\mathbf{D} \mid (\textstyle\bigwedge_{r=1}^{n} d_r = f_{i_r}(\vec{k})) \wedge (\bigwedge_{p=1}^{k}\bigwedge_{r=1}^{m_p} d_{pr} = f_{j_{pr}}(\vec{k})))\}$$

and

$$\mathbf{E}_S := \text{“}v[d_1]\ldots[d_n] \;=\; exp(u_1[d_{11}]\ldots[d_{1m_1}], \ldots, u_k[d_{k1}]\ldots[d_{km_k}])\text{”}.$$

Example 16

We illustrate the constrained expression representation on simple programs that we have already used as examples in Chapter 4 while presenting our representation. They are shown here again in Figures 6.2 (original program) and 6.3 (transformed program). The latter is obtained after applying certain loop transformations on the former.

First consider the program in Figures 6.2. It has two statements, X and Y, and therefore, in Samsom's representation it is described by two constrained expressions, one for each statement. They are as shown below.

Statement X:

$\mathbf{D}_X := \{[k] \mid 0 \le k \le 2 * N - 2 \wedge k \in \mathbb{Z}\}$

$\mathbf{P}_X := \{[d_1, d_2, d_3] \mid (\exists\, k \in \mathbf{D}_X \mid d_1 = k \wedge d_2 = k \wedge d_3 = 3 * k)\}$

$\mathbf{E}_X :=$ “`tmp[d1] = f(in1[d2], in2[d3])`”

$\mathbf{C}_X := (\mathbf{P}_X, \mathbf{E}_X)$

Statement Y:
$\mathbf{D}_Y := \{[i,j] \mid 0 \leq i \leq N-1 \wedge 0 \leq j \leq N-1 \wedge [i,j] \in \mathbb{Z}^2\}$

$\mathbf{P}_Y := \{[d_1,d_2,d_3] \mid (\exists\, [i,j] \in \mathbf{D}_Y \mid d_1 = i \wedge d_2 = j \wedge d_3 = i+j)\}$

$\mathbf{E}_Y :=$ "`out[`d_1`][`d_2`] = tmp[`d_3`]`"

$\mathbf{C}_Y := (\mathbf{P}_Y, \mathbf{E}_Y)$

Now consider the transformed program in Figure 6.3. It has four statements viz., P, Q, R and S, and therefore, it is described by the four constrained expressions that are shown below.

Statement P:
$\mathbf{D}_P := \{[k] \mid 0 \leq k \leq 2*N-2 \wedge k < N \wedge k \in \mathbb{Z}\}$

$\mathbf{P}_P := \{[d_1,d_2,d_3] \mid (\exists\, k \in \mathbf{D}_P \mid d_1 = k \wedge d_2 = k \wedge d_3 = 3*k)\}$

$\mathbf{E}_P :=$ "`tmp[`d_1`] = f(in1[`d_2`], in2[`d_3`])`"

$\mathbf{C}_P := (\mathbf{P}_P, \mathbf{E}_P)$

Statement Q:
$\mathbf{D}_Q := \{[k,p] \mid 0 \leq k \leq 2*N-2 \wedge k < N \wedge 0 \leq p \leq k \wedge [k,p] \in \mathbb{Z}^2\}$

$\mathbf{P}_Q := \{[d_1,d_2,d_3] \mid (\exists\, [k,p] \in \mathbf{D}_Q \mid d_1 = p \wedge d_2 = k-p \wedge d_3 = k)\}$

$\mathbf{E}_Q :=$ "`out[`d_1`][`d_2`] = tmp[`d_3`]`"

$\mathbf{C}_Q := (\mathbf{P}_Q, \mathbf{E}_Q)$

Statement R:
$\mathbf{D}_R := \{[k] \mid 0 \leq k \leq 2*N-2 \wedge k \geq N \wedge k \in \mathbb{Z}\}$

$\mathbf{P}_R := \{[d_1,d_2,d_3] \mid (\exists\, k \in \mathbf{D}_R \mid d_1 = k \wedge d_2 = k \wedge d_3 = 3*k)\}$

$\mathbf{E}_R :=$ "`tmp[`d_1`] = f(in1[`d_2`], in2[`d_3`])`"

$\mathbf{C}_R := (\mathbf{P}_R, \mathbf{E}_R)$

Statement S:
$\mathbf{D}_S := \{[k,p] \mid 0 \leq k \leq 2*N-2 \wedge k \geq N \wedge k-N+1 \leq p \leq N-1 \wedge (k,p) \in \mathbb{Z}^2\}$

$\mathbf{P}_S := \{[d_1,d_2,d_3] \mid (\exists\, [k,p] \in \mathbf{D}_S \mid d_1 = p \wedge d_2 = k-p \wedge d_3 = k)\}$

$\mathbf{E}_S :=$ "`out[`d_1`][`d_2`] = tmp[`d_3`]`"

$\mathbf{C}_S := (\mathbf{P}_S, \mathbf{E}_S)$

Once a given program has been represented as a set of constrained expressions, the set can be partitioned into equivalence classes by grouping together those that have identical expression parts. The assignment statements in the program that constrained expressions in the same class of such a partition represent are said to be (strongly) matching.

```
for (k = 0; k <= 2*N-2; k++)
    tmp[k] = f(in1[k], in2[3*k]);        // X

for (i = 0; i <= N-1; i++)
    for (j = 0; j <= N-1; j++)
        out[i][j] = tmp[i+j];            // Y
```

Figure 6.2: A program to illustrate Samsom's program representation.

```
for (k = 0; k <= 2*N-2; k++)
    if (k < N){
       tmp[k] = f(in1[k], in2[3*k]);     // P
       for (p = 0; p <= k; p++)
           out[p][k-p] = tmp[k];         // Q
    } else {
       tmp[k] = f(in1[k], in2[3*k]);     // R
       for (p = k-N+1; p <= N-1; p++)
           out[p][k-p] = tmp[k];         // S
    }
```

Figure 6.3: The program in Figure 6.2 after loop transformations.

Definition 6.2.2 (Matching statements)
Statements are (strongly) matching *iff they have identical operand variables in each of the operand positions, apply identical operators on them and define identical array variables. In other words, if* $(\mathbf{P}_1, \mathbf{E}_1)$ *and* $(\mathbf{P}_2, \mathbf{E}_2)$ *are the constrained expressions of statements* s_1 *and* s_2 *then they are matching iff* $\mathbf{E}_1 = \mathbf{E}_2$.

Definition 6.2.3 (Statement class)
A statement class *is a maximal subset of matching statements from the set of all statements in a given program. If* $\mathbf{S}$ *is the set of all statements in a program, the set of its statement classes is a partition of* $\mathbf{S}$ *that is given by the function* $\pi(\mathbf{S})$.

For a given statement class, due to the single assignment property of a $\mathscr{D}$ program, the preconditions of the individual constrained expressions are mutually disjoint and their union defines the precondition of the statement class. Therefore, every statement class has an expression and a precondition, which together define a constrained expression for the statement class that is called a *joined* constrained expression.

Example 17

We present the statement classes and their joined constrained expressions for our example programs using their representations in Example 16.

In Figure 6.2, the set of statements in the program, $\mathbf{S}_O = \{X, Y\}$. From the two constrained expressions we find that $\mathbf{E}_X \neq \mathbf{E}_Y$. Therefore, each statement represents a statement class by itself, that is, $\pi(\mathbf{S}_O) = \{\{X\}, \{Y\}\}$. The joined constrained expression of each class is trivially the constrained expression of its single statement. That is, for $\{X\}$ it is $(\mathbf{P}_X, \mathbf{E}_x)$ and for $\{Y\}$ it is $(\mathbf{P}_Y, \mathbf{E}_Y)$.

In the transformed program in Figure 6.3, the set of statements, $\mathbf{S}_T = \{P, Q, R, S\}$. From their constrained expressions, we find $\mathbf{E}_P = \mathbf{E}_R$ and $\mathbf{E}_Q = \mathbf{E}_S$. Therefore, $\pi(\mathbf{S}_T) = \{\{P, R\}, \{Q, S\}\}$. The joined constrained expressions for the classes $\{P, R\}$ and $\{Q, S\}$ are then given by $(\mathbf{P}_P \cup \mathbf{P}_R, \mathbf{E}_P)$ and $(\mathbf{P}_Q \cup \mathbf{P}_S, \mathbf{E}_Q)$, respectively.

6.2.2 Sufficient Condition for Equivalence of CEs

Given the representations of the original and the transformed $\mathscr{D}$ programs in terms of their constrained expressions, in what follows, we

discuss Samsom's sufficient condition for their equivalence under loop transformations.

From our discussion in Section 5.2, recall that the application of loop transformations neither creates new statement classes nor deletes existing ones. Therefore, similar to matching statements, there exists *matching classes*, i.e., statement classes that are in one-to-one correspondence between the original and the transformed programs. Provided the preconditions of the joined constrained expressions of the two matching classes are identical, there also exists identical instantiations of statements in the two programs that preserve all the data dependencies between the elements of the defined array and the elements of the operand variables. This implies equivalence of the two $\mathscr{D}$ programs. Hence, the following serves as the sufficient condition for equivalence in Samsom's method.

Definition 6.2.4 (Equivalence condition)
If $\mathrm{J_O}$ *and* $\mathrm{J_T}$ *are sets of joined constrained expressions of the original and the transformed* $\mathscr{D}$ *programs, then* $\forall\,(\mathbf{P_O}, \mathbf{E_O}) \in \mathrm{J_O}$, $\exists\,(\mathbf{P_T}, \mathbf{E_T}) \in \mathrm{J_T}$ *such that the following holds true:*

$$(\mathbf{E_O} = \mathbf{E_T}) \wedge (\mathbf{P_O} = \mathbf{P_T}).$$

A remark here is that when the transformed program has extraneous dependencies that define more output elements than the original program, the above condition fails even though it preserves all the dependencies in the original program. This may happen, for example, when a transformation erroneously increases the bound of a `for`-loop, but the error does not violate the functional equivalence of the two programs. Therefore, strictly speaking, in the equivalence condition, instead of requiring $\mathbf{P_O} = \mathbf{P_T}$, it suffices to require that $\mathbf{P_O} \subseteq \mathbf{P_T}$.

6.2.3 Checking Equivalence of CEs

The verification of loop transformations by functional equivalence checking of the original and the transformed programs proposed by Samsom directly implements the checking of the sufficient condition discussed in the previous section. The method is summarized in Algorithm 6. We will discuss generation of error diagnostics separately in Section 8.2.

Algorithm 6: Outline of Samsom's equivalence checker.

Input: J_O and J_T, sets of joined constrained expressions of the original and the transformed programs.
Output: If they are equivalent, return True, else return False, with error diagnostics.

```
begin
    foreach (P_O, E_O) ∈ J_O do
        if ∃ (P_T, E_T) ∈ J_T such that (E_O = E_T) then
            if P_O ⊈ P_T then
                return (False, errorDiagnostics)
        else
            return (False, errorDiagnostics)
    return True
end
```

Example 18

We show the equivalence of the pair of original and transformed programs given in Figures 6.2 and 6.3 by following Samsom's method. From Example 17 we have the sets of joined constrained expressions of the two programs.

$J_O := \{(\mathbf{P}_X, \mathbf{E}_X), (\mathbf{P}_Y, \mathbf{E}_Y)\}$

$J_T := \{(\mathbf{P}_P \cup \mathbf{P}_R, \mathbf{E}_P), (\mathbf{P}_Q \cup \mathbf{P}_S, \mathbf{E}_Q)\}$

Consider $(\mathbf{P}_X, \mathbf{E}_X) \in J_O$. Since,

$\mathbf{E}_X = \mathbf{E}_P =$ "`tmp[d1] = f(in1[d2], in2[d3])`",

we take their corresponding preconditions and check whether

$$\mathbf{P}_X \subseteq \mathbf{P}_P \cup \mathbf{P}_R.$$

We query this using a solver (for example, Omega calculator) and find that it holds.

We proceed with $(\mathbf{P}_Y, \mathbf{E}_Y) \in J_O$. We see that,

$\mathbf{E}_Y = \mathbf{E}_Q =$ "`out[d1][d2] = tmp[d3]`".

Therefore, we check whether

$$\mathbf{P}_Y \subseteq \mathbf{P}_Q \cup \mathbf{P}_S.$$

Again, we find that it holds.

Now each of the statement classes in the original program has been successfully matched with corresponding statement classes in the transformed

program and their data dependencies have also been found to be preserved. The original and the transformed $\mathscr{D}$ programs are hence guaranteed to be functionally equivalent.

6.2.4 Limitations of Samsom's Method

Samsom's method is able to handle only the full category of loop transformations and its main drawback is its restriction to this category. The loop transformations, as we mentioned earlier, are meant to increase the spatial and temporal locality of the data accesses to the data and instruction memory hierarchy by reordering the schedule of the accesses. But often, the full potential of loop transformations can be exploited only after a certain freedom is created by way of data-flow transformations. The loop transformations take the accesses to all the operand arrays in an assignment statement as a whole and are thereby tied down by multiple constraints. Therefore, it is common that data-flow transformations, like expression propagations and algebraic transformations, are applied in order to reduce the constraints for loop transformations. They are applied either in conjunction with or prior to applying loop transformations. As far as verification is concerned, in terms of the constrained expressions, data-flow transformations imply transformations on the expression in the statement. Therefore, the direct correspondence between the statement classes of the original and the transformed programs no longer holds as required by Samsom's method. This limits his method from being able to verify data-flow transformations.

The choice of constrained expressions as the program representation has yet another drawback related to their preconditions. Since the entire statement is taken as a whole, a precondition expresses all the data dependencies between the defined array and the operand arrays all at once. But when the number of operand arrays in the right-hand side of a statement is large and when they have large dimensions, the Presburger formula describing the precondition tends to be very large. This makes the very high computational complexity of checking the validity of the formulae to begin to have greater influence on the time taken to complete equivalence checking. In particular, consider that we are given programs in which there are no recurrences. In this case, it is not hard to see that expression propagation transformations can be handled in conjunction with loop transformations by inlining all the expressions completely. Implementing this workaround obviously explodes the number

of operand arrays of the inlined statement and renders its precondition so large as to make the validity checking involving such formulae altogether infeasible. In order to address the problem of handling large preconditions in some common cases, Čupák (1998) proposed a heuristic called *incremental dimension handling* (Čupák et al. 2003). However, this heuristic has not been automated and it is also not clear to what extent it is able to help in practice.

6.3 Verification of Loop and Data-Reuse Transformations

In this section, we present an equivalence checking method that can verify the application of a combination of any of the transformations from the full category of loop transformations and data-reuse transformations.

Let us first briefly recall the discussion we had on data-reuse transformations in Section 5.3.1. We noted that data-reuse transformations form a simple, but important, sub-class of the full category of data-flow transformations. They are simple in the sense that they may replace the operand arrays in some of the statements in the original program, and introduce new copy statements. This implies that there is a correspondence between the statements between the two programs, but only that it is weaker due to the mismatch in the operand arrays.

Example 19

Consider a simple example pair of programs under loop and data-reuse transformations. Let the original and the transformed programs be as shown in the Figure 6.4. Firstly, the inner loop has been unrolled once and secondly, a buffer `buf[]` has been introduced in place of the operand array `in1[]` in the first statement in the unrolled body (statement Y). A copy statement has also been inserted (statement X) as would be required to preserve the data-flow of the program.

In the transformed program, we find that the statements Y and Z are not matching statements (Definition 6.2.2) and they hence form separate statement classes. Following this, we see that statement class $\{Q\}$ in the original program matches only the statement class $\{Z\}$ in the transformed program, which does not account for a part of the original data-flow. This

results in a failure to satisfy the equivalence condition in Definition 6.2.4. However, it is clear that all that is required for combining the statements Y and Z into the same class is simply the elimination of the introduced array `buf[]` from statement Y.

In order to extend Samsom's method to also be able to verify data-reuse transformations, the statements that are non-matching only due to the newly introduced arrays in place of the original operand arrays have also to be matched. We can do so if we eliminate the newly introduced arrays. But in order to do so, it is clear that we need to examine the data dependencies inside the individual statements. That is, we need more fine-grained information about the data dependencies than what we have by way of the preconditions defined by Samsom. We have discussed such a fine-grained representation of the data dependencies in Section 4.2. We recall this representation and present it as a refinement of Samsom's precondition in Section 6.3.1. As a first step toward handling the full category of data-flow transformations, we use this representation and present a sufficient condition for their equivalence in Section 6.3.3. This, as we will see in Section 6.3.4, enables verification of loop and data-reuse transformations together. We conclude the section by recalling our experience with an implementation of the method in Section 6.3.5.

6.3.1 Preconditions as Tuples of Dependency Mappings

Let us take a fresh look at the nature of preconditions that we have discussed in Section 6.2.1. In the template statement in Figure 6.1, there is a defined array that is n-dimensional and k operand arrays with dimensions $m_1, m_2, \ldots, m_k$, respectively. As given in the definition of a constrained expression (Definition 6.2.1), the statement's precondition is an $(n + m_1 + m_2 + \cdots + m_k)$-dimensional integer domain that is defined by the index expressions of the defined and operand arrays, whose iterators are bound by the statement's iteration domain. Because of the single-assignment property of the $\mathscr{D}$ programs, it is clear that we do not loose any information by replacing this single integer domain with a tuple of k integer domains, where each domain is defined by the index expressions of the defined array and those of one of the operand arrays. An i-th integer domain in the tuple will then be $(n + m_i)$-dimensional. Moreover, each of these integer domains can instead be conveniently en-

```
for (i = 0; i < N; i++)
  for (j = 0; j < 2*N; j++)
    out[i][j] = f(in1[i], in2[i][j]);        // Q
```

(a) Original program

```
for (i = 0; i < N; i++){
  buf[i] = in1[i];                           // X
  for (j = 0; j < 2*N; j+=2){
    out[i][j]   = f(buf[i], in2[i][j]);      // Y
    out[i][j+1] = f(in1[i], in2[i][j+1]);    // Z
  }
}
```

(b) Transformed program.

Figure 6.4: An example for combined loop and data-reuse transformations.

$$S: \quad v[f_{i_1}(\vec{k})]\ldots[f_{i_n}(\vec{k})] = \exp(u_1[f_{j_{11}}(\vec{k})]\ldots[f_{j_{1m_1}}(\vec{k})], \ldots, u_k[f_{j_{k1}}(\vec{k})]\ldots[f_{j_{km_k}}(\vec{k})]);$$

Figure 6.5: Template statement in a $\mathscr{D}$ program.

coded as a mapping. The i-th mapping would then be a mapping from an n-dimensional domain to an m_i-dimensional domain. This gives the element-wise relationship between the defined array and each of the statement's operand arrays and this is precisely what we called a dependency mapping (Definition 4.2.4) earlier. It should now be clear that the precondition of a statement is just all its dependency mappings taken together and encoded as a domain instead of a mapping. Therefore, we conclude that we can replace the preconditions as defined by Samsom with sets of dependency mappings.

Now, we redefine Definition 6.2.1 in the light of the above observation. Again, consider a template statement S as shown in Figure 6.5, its constrained expression is defined in terms of dependency mappings as follows.

Definition 6.3.1 (Constrained expression, $\mathbf{C}_S$)
The constrained expression of a statement is a tuple comprising of a precondition and an expression. The precondition is in turn a tuple of dependency mappings from the defined array in the statement to each of the operand arrays in the right-hand side. The expression is a signature that is obtained by replacing all the index functions of its arrays by distinct identifiers. For a template statement S, as shown in Figure 6.5, having an iteration domain **D***, its constrained expression is given by,*

$$\mathbf{C}_S := (\mathbf{P}_S, \mathbf{E}_S),$$

where,
$\mathbf{P}_S := (\mathbf{M}_{v,u_1}, \ldots, \mathbf{M}_{v,u_k})$
and
$\mathbf{E}_S :=$ "$v[d_1]\ldots[d_n]$ = exp($u_1[d_{11}]\ldots[d_{1m_1}]$, $\ldots$, $u_k[d_{k1}]\ldots[d_{km_k}]$)".

Example 20

Consider the programs in the example pair at hand. We show their program representation in terms of dependency mappings and statement expressions. Consider the original program in Figure 6.4(a). Since it has only one statement, Q, its representation is just the representation of the statement.

Statement Q:
$\mathbf{D}_Q := \{[i,j] \mid 0 \leq i < N \wedge 0 \leq j < 2 * N \wedge (i,j) \in \mathbb{Z}^2\}$
$\mathbf{M}_{out,in1} := \{[d_1, d_2] \rightarrow [d_3] \mid (\exists\,(i,j) \in \mathbf{D}_Q \mid d_1 = i \wedge d_2 = j \wedge d_3 = i)\}$
$\mathbf{M}_{out,in2} := \{[d_1, d_2] \rightarrow [d_4, d_5] \mid (\exists\,(i,j) \in \mathbf{D}_Q \mid d_1 = i \wedge d_2 = j \wedge d_4 = i \wedge d_5 = j)\}$
$\mathbf{P}_Q := (\mathbf{M}_{out,in1}, \mathbf{M}_{out,in2})$
$\mathbf{E}_Q :=$ "out[d_1][d_2] = f(in1[d_3], in2[d_4][d_5])"
$\mathbf{C}_Q := (\mathbf{P}_Q, \mathbf{E}_Q)$

Now consider the transformed program in Figure 6.4(b). The individual representations of the three statements, X, Y and Z are as given below.

Statement X:
$\mathbf{D}_X := \{[i] \mid 0 \leq i < N \wedge i \in \mathbb{Z}\}$
$\mathbf{M}_{buf,in1} := \{[d_1] \rightarrow [d_2] \mid (\exists\, i \in \mathbf{D}_X \mid d_1 = i \wedge d_2 = i)\}$
$\mathbf{P}_X := (\mathbf{M}_{buf,in1})$
$\mathbf{E}_X :=$ "buf[d_1] = in1[d_2]"
$\mathbf{C}_X := (\mathbf{P}_X, \mathbf{E}_X)$

Statement Y:
$\mathbf{D}_Y := \{[i,j] \mid 0 \le i < \mathtt{N} \wedge 0 \le j < 2 * \mathtt{N} \wedge (i,j) \in \mathbb{Z}^2\}$
$\mathbf{M}_{\mathtt{out},\mathtt{buf}} := \{[d_1,d_2] \to [d_3] \mid (\exists\,(i,j) \in \mathbf{D}_Y \mid d_1 = i \wedge d_2 = j \wedge d_3 = i)\}$
$\mathbf{M}_{\mathtt{out},\mathtt{in2}} := \{[d_1,d_2] \to [d_4,d_5] \mid (\exists\,(i,j) \in \mathbf{D}_Y \mid d_1 = i \wedge d_2 = j \wedge d_4 = i \wedge d_5 = j)\}$
$\mathbf{P}_Y := (\mathbf{M}_{\mathtt{out},\mathtt{buf}}, \mathbf{M}_{\mathtt{out},\mathtt{in2}})$
$\mathbf{E}_Y :=$ "`out[`d_1`][`d_2`] = f(buf[`d_3`], in2[`d_4`][`d_5`])`"
$\mathbf{C}_Y := (\mathbf{P}_Y, \mathbf{E}_Y)$

Statement Z:
$\mathbf{D}_Z := \{[i,j] \mid 0 \le i < \mathtt{N} \wedge 0 \le j < 2 * \mathtt{N} \wedge (i,j) \in \mathbb{Z}^2\}$
$\mathbf{M}_{\mathtt{out},\mathtt{in1}} := \{[d_1,d_2] \to [d_3] \mid (\exists\,(i,j) \in \mathbf{D}_Z \mid d_1 = i \wedge d_2 = j+1 \wedge d_3 = i)\}$
$\mathbf{M}_{\mathtt{out},\mathtt{in2}} := \{[d_1,d_2] \to [d_4,d_5] \mid (\exists\,(i,j) \in \mathbf{D}_Z \mid d_1 = i \wedge d_2 = j+1 \wedge d_4 = i \wedge d_5 = j+1)\}$
$\mathbf{P}_Z := (\mathbf{M}_{\mathtt{out},\mathtt{in1}}, \mathbf{M}_{\mathtt{out},\mathtt{in2}})$
$\mathbf{E}_Z :=$ "`out[`d_1`][`d_2`] = f(in1[`d_3`], in2[`d_4`][`d_5`])`"
$\mathbf{C}_Z := (\mathbf{P}_Z, \mathbf{E}_Z)$

6.3.2 Statement Matching and Elimination of Copies

Similar to Samsom's method, we too begin by partitioning the set of statements in the two programs. For the original program, the partitioning is as defined by classes of strongly matching statements. However, for the transformed programs, the operand arrays might have been replaced by intermediate buffer arrays due to the data-reuse transformations. Therefore, we relax the definition of matching statements in order to be able to bring together all those statements that may match. We do so by letting *weakly matching statements* define a *weak statement class* and obtain a partition of the statements in the program.

Definition 6.3.2 (Weakly matching statements)
Statements are weakly matching *iff they define the same array variable and apply the same function on their operand variables.*

If s_1 and s_2 are weakly matching with constrained expressions $(\mathbf{P}_1, \mathbf{E}_1)$ and $(\mathbf{P}_2, \mathbf{E}_2)$, then their signatures are said to be weakly matching and we denote the same by $\mathbf{E}_1 \approx \mathbf{E}_2$.

Definition 6.3.3 (Weak statement class)
A weak statement class *is a maximal subset of weakly matching statements from the set of all statements in a given program. If* $\mathbf{S}$ *is the set of all statements in a program, the set of its weak statement classes is a partition of* $\mathbf{S}$ *that is given by the function* $\tau(\mathbf{S})$.

Example 21

Consider the representations of our current program pair shown in Example 20. For the original program, we partition its statements into strong statement classes. Since, the set of its statements, $\mathbf{S}_O = \{Q\}$, we trivially have, $\pi(\mathbf{S}_O) = \{\{Q\}\}$. For the transformed program, we partition its statements, $\mathbf{S}_T = \{X, Y, Z\}$, into weak statement classes. The statements Y and Z define the same array `out[]` and apply the same function `f/2` on the operand arrays. Hence they are weakly matching and we have the partition of weak statement classes given by $\tau(\mathbf{S}_T) = \{\{X\}, \{Y, Z\}\}$.

For each operand array that is replaced in the statements in the original program with a buffer array in the transformed program due to the data-reuse transformations, at least one additional copy statement is inserted in the transformed program that assigns values to the buffer array. The source of values in the copy statement may in turn be another buffer array, which requires insertion of at least another copy statement and so on, until the values are ultimately assigned from the original operand array as in the original program in a final copy statement. Such a flow of data values along the chain of buffers formed by the copy statements defines a data dependence path (Section 4.4.2) from the defined array upto the operand array, separated by intermediate buffer arrays. Therefore, we have a transitive dependency mapping (Section 4.4.1) associated with the path.

In the transformed program, let v and w be arrays and $b_1, \ldots, b_n$ be buffer arrays such that: (1) there is a statement s with a dependency mapping between v and b_1, (2) there are copy statements $s_1, \ldots, s_{n-1}$ where each s_i has a dependency mapping between b_i and b_{i+1}, and (3) there is a copy statement s_n with a dependency mapping between b_n and w. Then the transitive dependency mapping along the path, $p = (s, s_1, \ldots, s_{n-1}, s_n)$, is given by,

$$ {}_p\mathbf{M}_{v \rightsquigarrow w} := {}_{s_v}\mathbf{M}_{v,b_1} \bowtie {}_{s_1}\mathbf{M}_{b_1,b_2} \bowtie \ldots \bowtie {}_{s_{n-1}}\mathbf{M}_{b_{n-1},b_n} \bowtie {}_{s_n}\mathbf{M}_{b_n,w}. $$

In the event that the above path has a recurrence (Section 4.5) involving only the copy statements, we calculate the across-recurrence mapping and use it to avoid the inefficiency of having to calculate the transitive dependency mapping of the completely unrolled recurrence of the copies. That is, if we have a recurrence of copies anchored at b_i as $\ldots, b_{i-1}, b_i, \ldots, b_i, b_{i+1}, \ldots$, we calculate the across-recurrence mapping as described in Algorithm 2 and compute the transitive dependency mapping over the recurrence ${}_{s_{i-1}}\mathbf{M}^{R}_{b_{i-1},b_i}$ and use it in place of ${}_{s_{i-1}}\mathbf{M}_{b_{i-1},b_i}$ in the definition of ${}_{p}\mathbf{M}_{v \rightsquigarrow w}$ given above.

If there are no buffer arrays separating arrays v and w, then the transitive dependency mapping between them is trivially,

$$ {}_{p}\mathbf{M}_{v \rightsquigarrow w} := {}_{s}\mathbf{M}_{v,w}. $$

There may, however, be multiple paths starting at the same statement from the defined array to the operand array. Therefore, if $C(v, w, s)$ is the set of paths (chains of buffers) between v and w, all starting at s, then the combined transitive dependency mapping that completely eliminates the intermediate copies between v and w for statement s is given by,

$$ {}_{s}\mathbf{M}^{C}_{v,w} := \bigcup_{p \in C(v,w,s)} {}_{p}\mathbf{M}_{v \rightsquigarrow w}. $$

Recall that the precondition of the joined constrained expression of each statement class in Samsom's representation is given by the union of the preconditions of the individual statements belonging to the class. But due to our refinement of a precondition to a set of dependency mappings, in order to define the joined constrained expression of a statement class, we collect together the dependency mappings for each operand position from each of the preconditions of the statements belonging to the class and take their union separately. This lets us define the joined constrained expression of each statement class as the set of unions of dependency mappings.

For the original program, let $\mathbf{S}_O$ be the set of statements in the program and $\pi(\mathbf{S}_O)$ be its partition into (strong) statement classes. Then we have, for each class $x \in \pi(\mathbf{S}_O)$, with statements having k operand variables, the precondition of its joined constrained expression as given by,

$$ \mathbf{P}_x := \{ \bigcup_{s \in x} {}_{s}\mathbf{M}_{v,w_1}, \ldots, \bigcup_{s \in x} {}_{s}\mathbf{M}_{v,w_k} \}. $$

For the transformed program, let $\mathbf{S}_T$ be the set of statements in the program and $\tau(\mathbf{S}_T)$ be its partition into weak statement classes. Then we have, for each class $y \in \tau(\mathbf{S}_T)$, with statements having k operand variables, the precondition of its joined constrained expression as given by,

$$\mathbf{P}_y := \{ \bigcup_{s \in y} {}_s\mathbf{M}^C_{v,w_1}, \ldots, \bigcup_{s \in y} {}_s\mathbf{M}^C_{v,w_k} \}.$$

Example 22

From Example 20 we have the representations for the programs in Figures 6.4(a) and 6.4(b).

In the transformed program, `buf[]` is the only intermediate buffer array introduced between the arrays `out[]` and `in1[]`. The data dependence path defined by it is:

$$p : \texttt{out} \xrightarrow{Y} \texttt{buf} \xrightarrow{X} \texttt{in1},$$

and its transitive dependency mapping is given by,

$${}_p\mathbf{M}_{\texttt{out}\rightsquigarrow\texttt{in1}} := {}_Y\mathbf{M}_{\texttt{out,buf}} \bowtie {}_X\mathbf{M}_{\texttt{buf,in1}}.$$

There is only one path between `out[]` and `in1[]`. Therefore,

$${}_Y\mathbf{M}^C_{\texttt{out,in1}} := {}_p\mathbf{M}_{\texttt{out}\rightsquigarrow\texttt{in1}}.$$

The statement classes in the two programs have been discussed in Example 21. The original program has only one statement class $\{Q\}$, therefore its set of joined constrained expressions is a singleton and is trivially,

$$J_O := \{(\mathbf{P}_Q, \mathbf{E}_Q)\}.$$

The transformed program has two statement classes, namely $\{X\}$ and $\{Y, Z\}$. Therefore, its set of joined constrained expressions is,

$$J_T := \{(\mathbf{P}_X, \mathbf{E}_X),\ (\{{}_Y\mathbf{M}^C_{\texttt{out,in1}} \cup {}_Z\mathbf{M}^C_{\texttt{out,in1}},\ {}_Y\mathbf{M}^C_{\texttt{out,in2}} \cup {}_Z\mathbf{M}^C_{\texttt{out,in2}}\}, \mathbf{E})\},$$

where $\mathbf{E} =$ "`out[`d_1`][`d_2`] = f(_,_)`".

6.3.3 Sufficient Condition for Equivalence

Given the original and the transformed $\mathscr{D}$ programs, where the latter is the result of a combination of loop and data-reuse transformations, in what follows, we define a sufficient condition for their equivalence.

The condition we define is based on the representations of the two programs in terms of their constrained expressions, where preconditions are sets of dependency mappings. In the previous section, we have discussed how the statements in the transformed program are partitioned such that statements with mismatching operand arrays are still brought together. Again, recall that due to the loop and data reuse transformations the statements in the two programs may differ only in their operand arrays. Therefore, for each (strongly matching) statement class in the original program, there exists a (weakly matching) statement class in the transformed program such that they weakly match.

We have also discussed how we can recover the original dependency mappings in the transformed program by complete elimination of the copies. Now, suppose that we check that for each pair of weakly matching statement classes between the two programs, each of the mappings in the precondition of the joined constrained expression of the class in the original program are preserved in the joined constrained expression of the class in the transformed program. This implies that the computation and all the data dependencies between the output and the input arrays in the original program are preserved in the transformed program. Therefore, our checking is sufficient to guarantee the equivalence of the two $\mathscr{D}$ programs concerned. Hence, the following serves as the sufficient condition for equivalence of two programs under combined application of loop and data-reuse transformations.

Definition 6.3.4 (Equivalence condition)
If J_O *and* J_T *are sets of joined constrained expressions of the original and the transformed* $\mathscr{D}$ *programs, then* $\forall\,(\mathbf{P}_O, \mathbf{E}_O) \in J_O$, $\exists\,(\mathbf{P}_T, \mathbf{E}_T) \in J_T$ *such that the following holds true:*

$$(\mathbf{E}_O \approx \mathbf{E}_T) \wedge \bigwedge_{i=1}^{k} ({}^{O}\mathbf{M}_{v,w_i} \subseteq {}^{T}\mathbf{M}_{v,w_i}),$$

where $\mathbf{E}_O$ *is a class of (strongly) matching statements,* $\mathbf{E}_T$ *is a class of weakly matching statements,* ${}^{O}\mathbf{M}_{v,w_i} \in \mathbf{P}_O$, ${}^{T}\mathbf{M}_{v,w_i} \in \mathbf{P}_T$ *and the matching statements have* k *operands.*

The sufficiency of the equivalence condition is formulated in the following theorem.

Theorem 6.3.5 *Let* O *and* T *be a pair of* $\mathscr{D}$ *programs which have the same inputs and outputs and for which the equivalence condition holds. Then both programs compute the same input-output function.*

Proof (Sketch.) Without loss of generality, we assume arrays have only one index. We have to prove that if an output array element has a value **v** in O then it has the same value **v** in T.

Let $o[d]$ be an output array element. The value **v** assigned to $o[d]$ in O is given by a function $f(w_1[d_1], \ldots, w_k[d_k])$. The dependency mapping ${}^{O}\mathbf{M}_{o,w_i}$ identifies the element of $w_i[d_i]$ that serves as i-th input to f. In T, the value **v** assigned to $o[d]$ is given by $f(w'_1[l_1], \ldots, w'_k[l_k])$. The transitive dependency mapping ${}^{T}\mathbf{M}_{o,u_i}$ eliminates the copies and identifies the element $u_i[e_i]$ that is at the origin of the value of $w'_i[l_i]$ and hence serves as i-th input to f in T.

Hence $o[d]$ is assigned the same value in T when $w_i[d_i] = u_i[e_i]$ for all i. The equivalence condition ensures that $w_i = u_i$ and $d_i = e_i$. It remains to show that $w_i[d_i]$ has the same value in O and T. This holds trivially when it concerns an element of an input array. In the other case, one can apply the same reasoning as for $o[d]$ and conclude, by induction, that indeed all array elements $w_i[d_i]$ have the same value in O and T. □

6.3.4 Equivalence Checking Method

The verification of a combination of loop transformations and data-reuse transformations by functional equivalence checking of the original and the transformed programs implements the checking of the sufficient condition discussed in the previous section. The method is summarized in Algorithm 7. We will discuss generation of error diagnostics separately in Section 8.2.

Example 23

We show the equivalence of the pair of original and transformed programs given in Figures 6.4(a) and 6.4(b). Their representations are shown in Example 20 and their sets of joined constrained expressions are shown in Example 22.

There is only one statement class in the original that is to be matched in the transformed, namely, $(\mathbf{P}_Q, \mathbf{E}_Q) \in J_O$. Note that,

Algorithm 7: An equivalence checker for verification of loop and data-reuse transformations.

Input: J_O and J_T, sets of joined constrained expressions of the original and the transformed programs.
Output: If they are equivalent, return True, else return False, with error diagnostics.

```
begin
    foreach (P_O, E_O) ∈ J_O do
        if ∃ (P_T, E_T) ∈ J_T such that (E_O ≈ E_T) then
            for i ← 1 to k do
                Let ^O M_{v,w_i} ∈ P_O and ^T M_{v,w_i} ∈ P_T;
                if (^O M_{v,w_i} ⊈ ^T M_{v,w_i}) then
                    return (False, errorDiagnostics)
        else
            return (False, errorDiagnostics)
    return True
end
```

$$\mathbf{E}_Q \approx \mathbf{E} = \text{“}\texttt{tmp[d}_1\texttt{] = f(_,_)}\text{”}.$$

Therefore, we take the corresponding dependency mappings from their preconditions and check whether

$${}_Q\mathbf{M}_{\texttt{out,in1}} \subseteq ({}_Y\mathbf{M}^C_{\texttt{out,in1}} \cup {}_Z\mathbf{M}^C_{\texttt{out,in1}})$$

Querying this using a solver shows that it holds. We proceed with the checking of the other operand,

$${}_Q\mathbf{M}_{\texttt{out,in2}} \subseteq ({}_Y\mathbf{M}^C_{\texttt{out,in2}} \cup {}_Z\mathbf{M}^C_{\texttt{out,in2}})$$

Again, we find that this too holds. That completes the checking and we are done.

Now each of the statement classes in the original program have been successfully matched with the weakly corresponding statement classes in the transformed program and their data dependencies have also been found to be preserved. The original and the transformed $\mathscr{D}$ programs are hence guaranteed to be functionally equivalent.

6.3.5 Prototype Implementation and Experience

We have implemented our technique for verification of loop and data-reuse transformations in a preliminary prototype tool. It constructs the equivalence conditions based on the geometric models extracted from the two programs and uses the OMEGA calculator for checking their validity.

The tool has successfully verified some real life examples with many complex loops and multi-dimensional arrays, like data reuse transformations on MPEG-4 motion estimation kernel and loop transformations on implementations of signal processing application cores like Durbin and updating singular value decomposition (USVD) algorithm. The verification was possible in a *push-button* style and required only a few seconds.

In the USVD case, the tool detected an error in the transformed USVD (400 lines of C-code in the core). We were able to localize the error in the transformed code thanks to the error diagnostics generated by our tool. This helped trace the cause of the error to a bug in the constant propagation unit of the code generator that a prototype loop transformation tool used. In the past, both testing and manual paper-and-pencil based checking had taken unreasonable amounts of time to do this kind of checking, and yet without guarantee of correctness. Moreover, localizing the errors in an erroneously transformed code was an even more daunting task.

6.4 Limitations of Statement-Level Checking

Checking methods that are restricted to the statement-level scope rely on a rather close relationship between the original and the transformed programs. These methods assume that all array variables of the original program are preserved in the transformed program and also that the statements defining these array variables have identical signatures for the expressions in their right-hand side. This prevents them from handling all but loop and data-reuse transformations that guarantee existence of such a close relationship between the programs. As discussed earlier, ambitious optimizing transformations often include expression propagations and exploit algebraic properties of the operators. This im-

plies that statements may not match between the two programs and therefore, cause a failure of the statement-level checking to prove their equivalence. Hence, extensions to the discussed method are needed (see Chapter 7).

6.5 Summary

In this chapter, we have discussed Samsom's statement-level equivalence checking method that is able to handle the full category of loop transformations. After a discussion of its limitations, we refined the program representation so that we could handle the full category of loop transformations combined with the data-reuse transformations. The method accounted for the weak matching of the statements in the transformed program by recovering the original data dependencies via elimination of copies. The new method, although more powerful and more applicable in design practice, is still limited to the transformations that it handles. From the discussion of these methods, it is clear that establishing correspondences under data-flow transformations inherently requires methods that look inside the statements and match individual operators in the two programs. We address this requirement by developing operator-level checking methods that are able to do this in the following chapter.

Chapter 7

Operator-Level Equivalence Checking Methods

7.1 Introduction

This chapter presents our general equivalence checking method that can verify the application of a combination of any of the transformations from the full categories of global loop and data-flow transformations. The method overcomes the limitations of the statement-level checking by reasoning at the level of the operators and the data dependencies that exist between the individual arrays and operators in the original and the transformed programs.

The exposition of the general method is split into two sections. In Section 7.2, we present a method for the verification of global loop transformations when they are applied in combination with the main sub-category of data-flow transformations, namely, expression propagations. Following this, in Section 7.3, we discuss how the remaining sub-category of algebraic transformations is included in the presented method to obtain our general method that is able to seamlessly verify transformations from the full categories of both global loop and data-flow transformations. Among algebraic transformations, we limit our discussion to only the most commonly occurring instances of the algebraic transformations, namely, those based on the associative and commutative properties of arithmetic and logic operators. Before summariz-

```
for (i = 0; i < 3*N; i++)
    out1[i] = in1[i+1] + f(in2[i+N]);    // Q
for (i = N; i < 2*N; i++)
    out2[i-N] = f(in2[i]) * in3[i];      // R
```

(a) Original program

```
for (i = 0; i < 3*N; i++)
  if (i < N) {
     tmp[i] = f(in2[i+N]);               // W
     out1[i] = in1[i+1] + tmp[i];        // X
     out2[i] = tmp[i] * in3[i+N];        // Y
  } else {
     out1[i] = in1[i+1] + f(in2[i+N]);   // Z
  }
```

(b) Transformed program

Figure 7.1: An example program pair under combined loop and expression propagation transformations.

ing the chapter, in Section 7.4, we discuss the limitations that remain in our general operator-level equivalence checking method.

7.2 Verification of LTs and Expression Propagations

In this section, we present an equivalence checking method that can verify a combination of transformations from the full category of loop transformations and expression propagation transformations.

Let us first briefly recall the discussion we had on expression propagation transformations in Section 5.3.2. We noted that expression propagations either introduce or eliminate intermediate variables in the original program. When an intermediate variable is introduced, the new variable copies values from a sub-expression on the right-hand side of a statement and replaces it in the statement, and when an intermediate variable is eliminated, the deleted variable is replaced in the statements

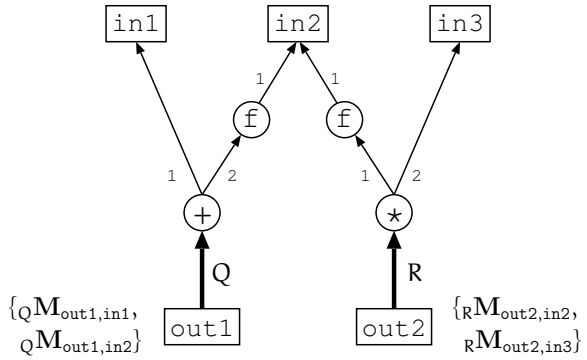
where it is read by the expression assigning values to it. In either case, the statements in question in the original program may no longer be even weakly matchable to the statements in the transformed program. However, the introduction or elimination of intermediate variables does not alter the computation in the data-flow of the program. Therefore, when the transformations have been applied correctly, there exists a correspondence between the data-flows of the two programs at the level of their individual operators. That is, given the ADDG representation of the two programs (see Section 4.3), components of the data-flow, or slices, in the original program have corresponding slices in the transformed program.

Example 24

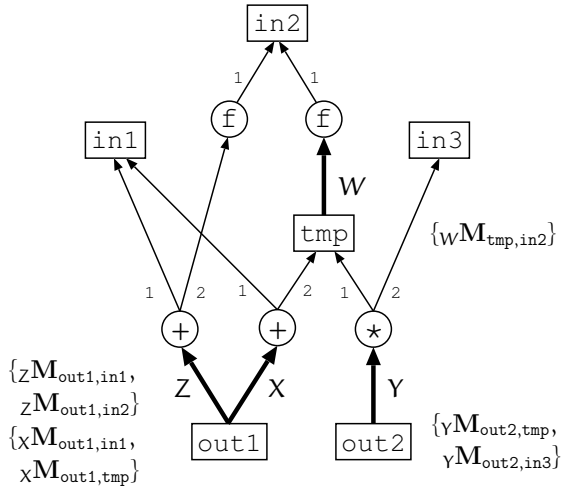
Figure 7.1 shows a simple program and its transformed version obtained after applying a combination of loop and expression propagation transformations on the original. It is not difficult to see that, in the original program, some of the values of `f(in2[])` computed in statement Q are again computed in statement R. In the transformed program, this recomputation has been avoided by propagating the sub-expression and regrouping the statement instances by loop transformations.

Two observations can be made in this example that show why equivalence condition for verification of loop and data-reuse transformations given in Definition 6.3.4 in page 114 will not suffice for verification of loop and expression propagation transformations. Firstly, the statement class {Q} matches the statement class {Z} in the transformed program, but, whereas the definition domain of `out1[]` in the former has `3*N` elements, the same domain has only `2*N` elements in the latter. As a result, the condition fails for {Q}. Secondly, the statement class {R} finds no match in the transformed program. Therefore, the condition fails for this class as well. The main cause for failure in both the cases is the inability to match the related data-flows in the two programs by only looking at the signatures of the statements in the two programs.

Let us now observe the ADDGs of the two programs. As shown in Figure 7.2, they reveal the operators inside the individual statements in the two programs. This additional information helps identify the individual components of the data-flow in each of the programs, and match the related components between them. The individual components in a data-flow of a program are what we have defined as data dependence slices in Definition 4.6.2 in page 66. For the two programs at hand, they are as shown in Figure 7.3.

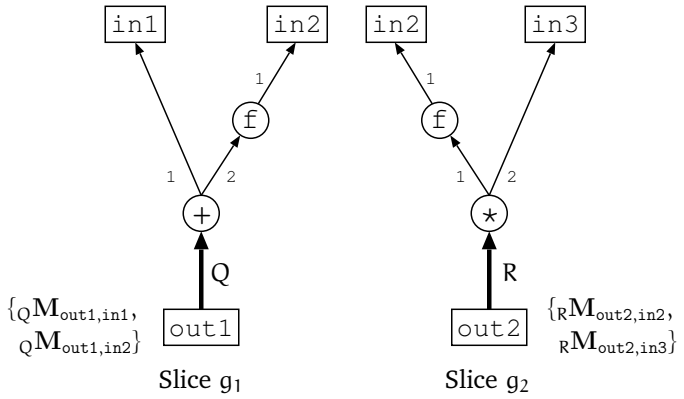


(a) The ADDG of the original program

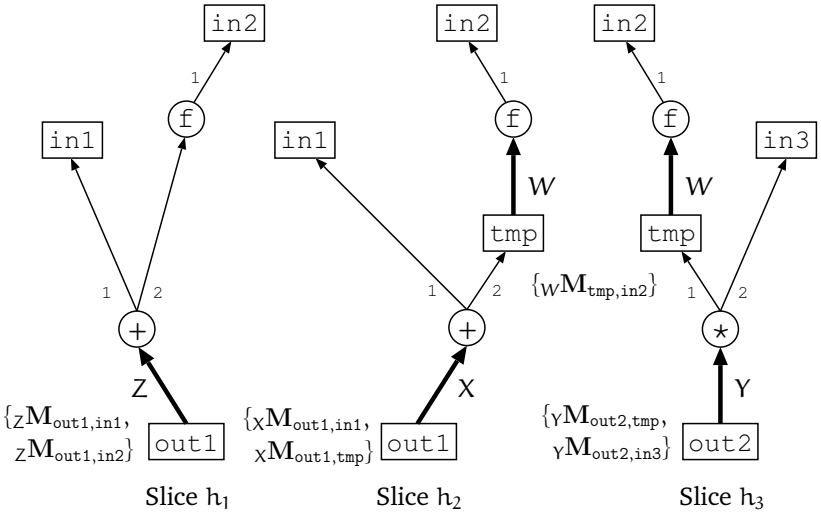


(b) The ADDG of the transformed program

Figure 7.2: The ADDGs of the example program pair for combined loop and expression propagation transformations given in Figure 7.1.



(a) The slices in the ADDG of the original program



(b) The slices in the ADDG of the transformed program

Figure 7.3: The slices in the ADDGs of the example program pair for combined loop and expression propagation transformations shown in Figure 7.2.

In Section 7.2.1, we introduce a sufficient condition for equivalence of programs defined over their ADDGs. The introduced condition is able to verify any combination of loop and expression propagation transformations. Next, in Section 7.2.2, we present a method to check this condition that is based on the traversal of the ADDGs of the input programs. Finally, in Section 7.2.3 we discuss how recurrences, when present in the programs, are tackled by the method.

7.2.1 Sufficient Condition for Equivalence

Given the original and the transformed $\mathscr{D}$ programs, where the latter is the result of application of some combination of loop and expression-propagation transformations, in what follows, we define a sufficient condition for their equivalence. The condition is based on the representations of the two programs as ADDGs. It relies on the notions of data dependence paths and slices that we have discussed in Chapter 4.

Let us start by recalling the two sufficient conditions of equivalence, Definition 6.2.4 on page 103 and Definition 6.3.4 on page 114, that we have already discussed. In each case, based on the level of scope that encompasses the effects of the transformations that are to be verified, the condition isolates components in the data-flow of the programs with respect to their output variables and imposes a constraint that has to hold for each of these components separately. The imposed constraint follows a common template. It requires that, for the given component in the original program, there exist a component in the transformed program such that, between the identical variables that are preserved by the applied transformations, they have (1) identical computation scheme (under a certain notion that is defined), *and* (2) identical data dependencies. We follow the same reasoning here.

For the verification of a combination of loop transformations and expression propagations, as we have discussed previously, it is necessary to match the individual components in the data-flows of the original and the transformed programs at the level of operators. Also, the transformations can both introduce or eliminate intermediate variables, and the only variables that are guaranteed to be preserved are the input and output variables of the programs. This implies that a component has to contain the complete data-flow that connects the input variables to its output variable. In an ADDG of a program, the set of paths from each

output array to the input arrays that take part in the data-flow, or what we call a data dependence slice (Definition 4.6.2 in page 66), represents such a component. Therefore, the sufficient condition for equivalence of the original and the transformed $\mathscr{D}$ programs under loop and expression propagation transformations, is defined by the constraint that, for each slice in the original program, there exists a slice in the transformed program such that, they have (1) identical computation scheme, and (2) identical output-to-input mappings. We make this condition precise in what follows.

Recall that a data dependence slice is defined by a set of paths from the output array to the input arrays with non-empty output-to-input mappings. Therefore, the computation scheme and the output-to-input mapping of each of the individual paths in a given slice, taken together, defines the computation scheme and the output-to-input mappings of the slice. Each path is composed of labeled nodes that represent arrays and operators, and labeled edges that represent the dependencies between them. The particular internal array nodes present on the path, and the labels of the statements that assign them, do not contribute to the computation defined by the slice. Moreover, they may not be preserved by the transformations, which makes them inconsequential as far as establishing correspondence between the programs is concerned. Therefore, the complete contribution of the path toward the computation scheme defined by the slice comes only from the operator nodes on it along with the labels on their out-going edges. They define the signature of the path or *path signature*.

Definition 7.2.1 (Path signature, $\mathbf{e}_p$)
Let p *be a data dependence path from array node* v *over array nodes* $v_1, \ldots, v_n$ *until array node* w*, let* $o_1, \ldots o_r$ *be (in that order) the operators on the path and,* $\forall i \in (1, \ldots, r)$*, let* l_i *be the label of the out-going edge of* o_i*. The path signature of* p *is a tuple* $\mathbf{e}_p$ *defined as*

$$\mathbf{e}_p := (v, o_1, l_1, o_2, l_2, \ldots, o_r, l_r, w).$$

Again, note from the above definition that even though a data dependence path may have internal array nodes, they are abstracted away in its path signature. We can now use the signatures of paths to define a matching relation between paths that have identical signatures.

Definition 7.2.2 (Matching paths)
Paths are matching iff they have the same source and destination array nodes with an identical order of operators and labels on their respective out-going edges on them. That is, two paths p *and* q *are matching iff* $\mathbf{e}_p = \mathbf{e}_q$.

Note that, by the definition of a data dependence slice, it follows that each path in a data dependence slice has a unique signature.

Proposition 7.2.3
For a data dependence slice g*, if paths* $p, q \in g$ *and* $p \neq q$*, then* $\mathbf{e}_p \neq \mathbf{e}_q$.

Example 25

Consider the slices shown in the Figure 7.3. The signature of the path

$$p_2 : \texttt{out1} \xrightarrow{Q} + \xrightarrow{2} \texttt{f} \xrightarrow{1} \texttt{in2},$$

in slice g_1 is

$$\mathbf{e}_{p_2} := (\texttt{out1}, +, 2, \texttt{f}, 1, \texttt{in2}).$$

Similarly, the signature of the path

$$q_2 : \texttt{out1} \xrightarrow{X} + \xrightarrow{2} \texttt{tmp} \xrightarrow{W} \texttt{f} \xrightarrow{1} \texttt{in2},$$

in slice h_2 is

$$\mathbf{e}_{q_2} := (\texttt{out1}, +, 2, \texttt{f}, 1, \texttt{in2}).$$

Since $\mathbf{e}_{p_2} = \mathbf{e}_{q_2}$, p_2 and q_2 are said to be matching paths.

We can now characterize a slice in terms of its constituent paths by combining their output-to-input mappings and signatures. Let g be a data dependence slice that has k paths ${}_gp_1, \ldots, {}_gp_k$ from the output array v to the input arrays $w_1, \ldots, w_k$. We assume an ordering on the paths. The ordering that is presented by the lexicographical ordering of tuples of edge labels on the paths is intuitive and we use it in our discussion. The slice g is characterized by the tuple

$$(\mathbf{P}_g, \mathbf{E}_g),$$

where $\mathbf{P}_g$ is the tuple of output-to-input mappings of the paths, that is,

$$\mathbf{P}_g := ({}_gp_1\mathbf{M}_{v \rightsquigarrow w_1}, \ldots, {}_gp_k\mathbf{M}_{v \rightsquigarrow w_k}),$$

and $\mathbf{E}_g$ is the tuple of signatures of the paths, or *slice signature*, that is,

$$\mathbf{E}_g := (\mathbf{e}_{g\,p_1}, \ldots, \mathbf{e}_{g\,p_k}).$$

The notion of slice signatures helps in relating slices with identical computation scheme or *matching slices* similar to matching statements and matching paths.

Definition 7.2.4 (Matching slices)
Slices are matching iff they have the same set of input arrays, compute the same function on them and define the same output array. That is, two slices g *and* h *are matching iff* $\mathbf{E}_g = \mathbf{E}_h$.

Example 26

In the Figure 7.3, the slice g_1 is characterized by the tuple $(\mathbf{P}_{g_1}, \mathbf{E}_{g_1})$, where

$$\mathbf{P}_{g_1} := ({}_{p_1}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{in1}}, {}_{p_2}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{in2}}),$$

and the slice signature,

$$\begin{aligned}\mathbf{E}_{g_1} &:= (\mathbf{e}_{p_1}, \mathbf{e}_{p_2})\\ &:= ((\mathtt{out1}, +, 1, \mathtt{in1}), (\mathtt{out1}, +, 2, \mathtt{f}, 1, \mathtt{in2})).\end{aligned}$$

Similarly, the slice h_2 is characterized by $(\mathbf{P}_{h_2}, \mathbf{E}_{h_2})$, where

$$\mathbf{P}_{h_2} := ({}_{q_1}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{in1}}, {}_{q_2}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{in2}}),$$

and the slice signature,

$$\begin{aligned}\mathbf{E}_{h_2} &:= (\mathbf{e}_{q_1}, \mathbf{e}_{q_2})\\ &:= ((\mathtt{out1}, +, 1, \mathtt{in1}), (\mathtt{out1}, +, 2, \mathtt{f}, 1, \mathtt{in2})).\end{aligned}$$

Since $\mathbf{E}_{g_1} = \mathbf{E}_{h_2}$, g_1 and h_2 are said to be matching slices.

Note that a matching between two slices defines a bijection between their sets of paths that uniquely pairs the paths with identical signatures together. The equivalence class of matching slices, or *slice class*, provides a unique partition of the set of data dependence slices in an ADDG.

Definition 7.2.5 (Slice class)
A slice class is a maximal subset of matching slices from the set of all slices in a given ADDG. If $\mathbf{G}$ *is the set of all slices in the ADDG of a program, the set of its slice classes is a partition of* $\mathbf{G}$ *that is given by the function* $\pi(\mathbf{G})$.

By the above definition, a single signature identifies all the slices in a class. Also, for a given slice class, due to the single assignment property of the $\mathscr{D}$ programs, the slices in the class have mutually disjunct definition domains. Therefore, we can extend the characterization of a slice to that of a slice class. Given a slice class $x \in \pi(\mathbf{G})$, it is given by the tuple

$$(\mathbf{P}_x, \mathbf{E}_x),$$

where $\mathbf{P}_x$ is the tuple of unions of the output-to-input mappings of matching paths among the matching slices, that is,

$$\mathbf{P}_x := (\bigcup_{g \in x} {}_g p_1 \mathbf{M}_{v \rightsquigarrow w_1}, \ldots, \bigcup_{g \in x} {}_g p_k \mathbf{M}_{v \rightsquigarrow w_k}),$$

and $\mathbf{E}_x$ is the signature of the matching slices.

Example 27

In the Figure 7.3, the set of data dependence slices in the ADDG of the original program, $\mathbf{G}_O := \{g_1, g_2\}$. Since, g_1 and g_2 have mismatching slice signatures, we have $\pi(\mathbf{G}_O) := \{\{g_1\},\{g_2\}\}$. However, in the transformed program, $\mathbf{G}_T := \{h_1, h_2, h_3\}$, and h_1 and h_2 have matching slice signatures. Therefore, we have $\pi(\mathbf{G}_T) := \{\{h_1, h_2\},\{h_3\}\}$.

The slice class $x := \{h_1, h_2\}$ in the transformed program is characterized by the tuple $(\mathbf{P}_x, \mathbf{E}_x)$, where

$\mathbf{P}_x := ({}_{h_1} q_1 \mathbf{M}_{\mathtt{out1} \rightsquigarrow \mathtt{in1}} \cup {}_{h_2} q_1 \mathbf{M}_{\mathtt{out1} \rightsquigarrow \mathtt{in1}}, {}_{h_1} q_2 \mathbf{M}_{\mathtt{out1} \rightsquigarrow \mathtt{in2}} \cup {}_{h_2} q_2 \mathbf{M}_{\mathtt{out1} \rightsquigarrow \mathtt{in2}})$

and $\mathbf{E}_x = ((\mathtt{out1}, +, 1, \mathtt{in1}), (\mathtt{out1}, +, 2, \mathtt{f}, 1, \mathtt{in2}))$.

Now the sufficient condition for equivalence of the two programs under combined loop and expression propagation transformations that we have discussed earlier can be defined as follows.

Definition 7.2.6 (Equivalence condition)
If $\mathbf{G}_O$ and $\mathbf{G}_T$ are sets of data dependence slices of the original and the transformed $\mathscr{D}$ programs, then $\forall (\mathbf{P}_O, \mathbf{E}_O) \in \pi(\mathbf{G}_O)$, $\exists (\mathbf{P}_T, \mathbf{E}_T) \in \pi(\mathbf{G}_T)$ such that the following holds true:

$$(\mathbf{E}_O = \mathbf{E}_T) \wedge \bigwedge_{i=1}^{k} ({}^{O}\mathbf{M}_{v \rightsquigarrow w_i} \subseteq {}^{T}\mathbf{M}_{v \rightsquigarrow w_i}),$$

where ${}^{O}\mathbf{M}_{v \rightsquigarrow w_i}$ and ${}^{T}\mathbf{M}_{v \rightsquigarrow w_i}$ are the i-th elements of $\mathbf{P}_O$ and $\mathbf{P}_T$, respectively, and the matching slice classes have k paths defining them.

The sufficiency of the above equivalence condition is formulated in the following theorem.

Theorem 7.2.7 *Let* O *and* T *be a pair of $\mathscr{D}$ programs which have the same input arrays and output arrays and for which the equivalence condition holds. Then both programs compute the same input-output function.*

Proof (Sketch.) Without loss of generality, we assume arrays have only one index. We have to prove that if an output array element has a value **v** in O then it has the same value **v** in T. In what follows, we refer to the part of the condition that requires $\mathbf{E}_O = \mathbf{E}_T$ as COND-A and the remaining part of the condition that requires $({}^{O}\mathbf{M}_{v \rightsquigarrow w_i} \subseteq {}^{T}\mathbf{M}_{v \rightsquigarrow w_i})$ as COND-B.

Let $o[d]$ be an output array element. The value **v** is assigned to $o[d]$ in O is given by a function $f(w_1[d_1], \ldots, w_k[d_k])$. In T, the value **v** assigned to $o[d]$ is given by $g(w'_1[l_1], \ldots, w'_k[l_k])$. But by COND-A, we have $f = g$. Now it remains to show that both functions have the same values in their operands, that is, $\forall i\, w_i[d_i] = w'_i[l_i] = u_i$.

We consider all possible cases for each pair of the i-th operands $w_i[d_i]$ and $w'_i[l_i]$.

- $w_i[d_i]$ and $w'_i[l_i]$ are input array elements: By COND-A, $w_i = w'_i$ and by COND-B, $d_i = l_i$. This trivially implies that they get the same values for any execution of the program.
- $w_i[d_i]$ is an input array element, but $w'_i[l_i]$ is not: By COND-A, $w'_i[l_i]$ can no more be assigned a value by a function, but only a copy of a value at an input array. ${}^{T}\mathbf{M}_{o \rightsquigarrow u_i}$ identifies the input array element $u_i[e_i]$ that is at the origin of the value of $w'_i[l_i]$. Now, the reasoning for the first case follows.
- $w'_i[l_i]$ is an input array element, but $w_i[d_i]$ is not: Similar to the previous case.
- $w_i[d_i]$ and $w'_i[l_i]$ are intermediate array elements: One can apply the same reasoning as for $o[d]$ and, by induction, reduce it to the identical input array elements.

□

Example 28

In the program pair at hand, from the previous discussion we have that slice class $\{g_1\}$ in the original program matches slice class $\{h_1, h_2\}$. It remains to check that the following hold.

$${}_{g_1}p_1\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{in1}} \subseteq {}_{h_1}q_1\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{in1}} \cup {}_{h_2}q_1\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{in1}}$$
$${}_{g_1}p_2\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{in2}} \subseteq {}_{h_1}q_2\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{in2}} \cup {}_{h_2}q_2\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{in2}}$$

Querying this with a solver shows that they indeed hold.

Now we are left with class $\{g_2\}$ and it matches the only remaining class in the transformed program $\{h_3\}$. Again the following conditions are checked and found to hold.

$${}_{g_2}p_1\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{in2}} \subseteq {}_{h_3}q_1\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{in2}}$$
$${}_{g_2}p_2\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{in3}} \subseteq {}_{h_3}q_2\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{in3}}$$

The two programs therefore meet the sufficient condition for equivalence.

7.2.2 Synchronized Traversal of Two ADDGs

In this section, we discuss our equivalence checking method for verification of combined loop and expression propagation transformations. The method efficiently implements the sufficient condition for equivalence that we have discussed in the previous section.

In the methods that we have discussed in Chapter 6, it suffices to translate the equivalence condition naïvely into a method implementing it. This is due to the fact that, although they were verifying transformations with a global scope, all variables in the original program were preserved in the transformed program. This has meant that they can operate on the statements or *horizontal* components of the data-flow of the program. In contrast, the equivalence condition that we want to implement now operates on slices, or *vertical* components of the data-flow of the program, that typically overlap. This implies that unlike for statements, it can be an overkill to enumerate the individual slices in the data-flow of a program to build their signatures and output-to-input mappings. Therefore, a naïve implementation of the condition can lead to an inefficient equivalence checking method. An efficient alternative is to exploit the presence of sharing of the data-flow that exists between paths within a slice and between slices to check the condition. We present such an alternative in this section.

In brief, given the ADDGs of the original and the transformed programs, the equivalence checking method we describe verifies the sufficient con-

dition by a *synchronized* traversal from outputs to inputs of the two ADDGs. The traversal attempts to simultaneously build the matching slices between the ADDGs for which the equivalence condition holds. Firstly, the corresponding output array nodes, the operator nodes and the labels on their outgoing edges serve as points of synchronization that guide the traversal in the search for matching slices. Secondly, the transitive dependency mappings for the paths in a slice are progressively updated during the traversal, so that, when the matching slices are built, their output-to-input mappings are available. Once they have been checked to hold, the equivalence condition has been proved to hold on the matching slices. If the traversal similarly succeeds for all output arrays, then the checking of the equivalence condition is complete for the given program pair. This way, the task of construction of the slices and the task of checking of the equivalence condition between them are combined.

We now elaborate on the process of traversal and present it in the flavor of a proof scheme. The traversal starts from pairs of corresponding output array nodes in the two ADDGs and proceeds bottom-up along the paths that originate from them. Whenever there is a branching of the paths, the initial pair of corresponding nodes is reduced to pairs of corresponding nodes among the successor nodes. Therefore, at any given instance during the traversal, we have with us the frontier of the end nodes of partially matching slices between the two ADDGs. For each pair of corresponding nodes at the frontier, the paths from their output arrays are both guaranteed to have the same operator nodes appearing in the same order on them with the same labels on their outgoing edges. In order to successfully construct the full matching slices, the remaining traversal is obliged to also result in similar matching of the rest of the path. Such correspondences between the ADDGs being traversed serve as the *proof obligations* for the equivalence proof to succeed for a given output array. This is made precise in the definitions that follow.

Definition 7.2.8 (Primitive proof obligation)
Given two ADDGs, G_1 *and* G_2*, a primitive proof obligation is of the form* $(v_1, v_2, \mathbf{M}_{\mathtt{out} \rightsquigarrow \mathtt{u}_1}, \mathbf{M}_{\mathtt{out} \rightsquigarrow \mathtt{u}_2})$*, where* v_1 *and* v_2 *are nodes from* G_1 *and* G_2*, respectively, and* $\mathbf{M}_{\mathtt{out} \rightsquigarrow \mathtt{u}_1}$ *and* $\mathbf{M}_{\mathtt{out} \rightsquigarrow \mathtt{u}_2}$ *are transitive dependency mappings with identical domains, that is,* $\mathrm{Domain}(\mathbf{M}_{\mathtt{out} \rightsquigarrow \mathtt{u}_1}) = \mathrm{Domain}(\mathbf{M}_{\mathtt{out} \rightsquigarrow \mathtt{u}_2})$*, where* $\mathtt{u}_1$ *and* $\mathtt{u}_2$ *are the array nodes that appear last on the concerned paths from the output array* out *to* v_1 *and* v_2*, respectively.*

Definition 7.2.9 (Proof obligation)
A proof obligation is a conjunction of primitive proof obligations.

Note that in our discussion, for a node v, we sometimes follow the convention of denoting its counterpart in `teletype` font to refer to the array node that appears last on a given path to v from the output array.

Definition 7.2.10 (Truth of proof obligation)
A proof obligation is true if each of its primitive proof obligations is true. A primitive proof obligation $(v_1, v_2, \mathbf{M}_{\mathtt{out}\rightsquigarrow\mathtt{u}_1}, \mathbf{M}_{\mathtt{out}\rightsquigarrow\mathtt{u}_2})$ *is true if*

$$\mathtt{u}_1[\,\mathbf{M}_{\mathtt{out}\rightsquigarrow\mathtt{u}_1}(\vec{\mathtt{i}})] = \mathtt{u}_2[\,\mathbf{M}_{\mathtt{out}\rightsquigarrow\mathtt{u}_2}(\vec{\mathtt{i}})]$$

$\forall\,\vec{\mathtt{i}} \in \mathrm{domain}(\mathbf{M}_{\mathtt{out}\rightsquigarrow\mathtt{u}_1})$*, for any execution of the two programs on identical inputs.*

The traversal maintains a proof obligation at all instances. The equivalence checking method initializes the proof process by constructing the initial proof obligation as shown in Algorithm 8. It is trivially given by a pair of array nodes for each output array and a pair of identity mappings that represent the transitive dependency mappings from each node in the pair to itself. The domains for these mappings are obtained by grouping together all elements of the output array that are defined by the original program.

Proposition 7.2.11
The truth of the initial proof obligation implies that, for the output variables that the transformed program has in common with the original program, both programs assign identical values for any execution of the two programs on identical inputs, that is, they are equivalent.

Example 29

Consider the original and the transformed program pair shown in Figure 7.1 and their ADDGs in Figure 7.2. The programs have two output arrays `out1[]` and `out2[]`, and their definition domains in the original program are as below.

${}_{\mathrm{Q}}\mathbf{W}_{\mathtt{out1}} := \{[\mathtt{i}] \mid 0 \leq \mathtt{i} < 3 * \mathtt{N} \wedge \mathtt{i} \in \mathbb{Z}\}$

${}_{\mathrm{R}}\mathbf{W}_{\mathtt{out2}} := \{[\mathtt{i}] \mid 0 \leq \mathtt{i} < \mathtt{N} \wedge \mathtt{i} \in \mathbb{Z}\}$

Algorithm 8: Proof initialization.

ProofInitialization(G_1, G_2)
Input: ADDGs G_1 and G_2.
Output: A set P of primitive proof obligations.
begin
  $P \longleftarrow \emptyset$;
  $\mathcal{O} \longleftarrow$ set of output arrays;
  foreach $v \in \mathcal{O}$ **do**
    $S \longleftarrow$ set of all statements that assign to v in G_1;
    $\mathbf{W}_v \longleftarrow \emptyset$;
    foreach $s \in S$ **do**
      $\mathbf{W}_v \longleftarrow \mathbf{W}_v \cup {}_s\mathbf{W}_v$;
    // Restrict the identity mapping;
    $\mathbf{M}_{v \rightsquigarrow v} \longleftarrow$ RestrictDomain$(I, \mathbf{W}_v)$;
    $P \longleftarrow P \cup \{(v, v, {}_{()}\mathbf{M}_{v \rightsquigarrow v}, {}_{()}\mathbf{M}_{v \rightsquigarrow v})\}$;
  return P
end

The identity transitive dependency mappings are as follows.

$\mathbf{M}_{\mathtt{out1} \rightsquigarrow \mathtt{out1}} := \{[i] \rightarrow [i] \mid i \in {}_Q\mathbf{W}_{\mathtt{out1}}\}$

$\mathbf{M}_{\mathtt{out2} \rightsquigarrow \mathtt{out2}} := \{[i] \rightarrow [i] \mid i \in {}_R\mathbf{W}_{\mathtt{out2}}\}$.

Therefore, the initial proof obligation for the equivalence proof of the program pair is given by

$$P := \{\ (\mathtt{out1}, \mathtt{out1}, {}_{()}\mathbf{M}_{\mathtt{out1} \rightsquigarrow \mathtt{out1}}, {}_{()}\mathbf{M}_{\mathtt{out1} \rightsquigarrow \mathtt{out1}}),$$
$$(\mathtt{out2}, \mathtt{out2}, {}_{()}\mathbf{M}_{\mathtt{out2} \rightsquigarrow \mathtt{out2}}, {}_{()}\mathbf{M}_{\mathtt{out2} \rightsquigarrow \mathtt{out2}})\}.$$

At each step, the traversal selects a primitive proof obligation from its current proof obligation and first checks whether it is a *terminal proof obligation*.

Definition 7.2.12 (Terminal proof obligation)
A primitive proof obligation $p = (v_1, v_2, \mathbf{M}_{\mathtt{out} \rightsquigarrow \mathtt{v}_1}, \mathbf{M}_{\mathtt{out} \rightsquigarrow \mathtt{v}_2})$ *is terminal iff* v_1 *and* v_2 *are input arrays.*

When the obligation at hand is terminal, it is checked whether it is true. The following proposition follows from Definition 7.2.10.

Proposition 7.2.13
A terminal proof obligation is true iff $v_1 = v_2$ *and* $\mathbf{M}_{\mathtt{out}\rightsquigarrow \mathtt{v}_1} = \mathbf{M}_{\mathtt{out}\rightsquigarrow \mathtt{v}_2}$, *i.e., the output-to-input mappings select the same elements in the same input arrays.*

If the terminal obligation is true, it implies that a pair of matching paths between the two programs has been successfully identified and their output-to-input mappings have been shown to be identical. On the contrary, if the obligation is not true, the equivalence proof cannot proceed, at least for the output array at the origin of the obligation. Hence it reports an error and terminates.

Algorithm 9: Reduction of a primitive proof obligation.

ReduceObligation(G_1, G_2, p)
Input: ADDGs G_1, G_2 and $p = (v_1, v_2, {}_{q_1}\mathbf{M}_{\mathtt{out}\rightsquigarrow \mathtt{v}_1}, {}_{q_2}\mathbf{M}_{\mathtt{out}\rightsquigarrow \mathtt{v}_2})$, the primitive proof obligation to be reduced.
Output: If successful, a set R of new primitive proof obligations obtained by reduction of p, otherwise, error diagnostics.
begin
  $Q \longleftarrow$ PromoteNodes(G_1, G_2, p); // Algorithm 10;
  $R \longleftarrow$ UpdateMappings(G_1, G_2, p, Q); // Algorithm 11;
  if $R \neq \emptyset$ **then**
    // Ensure that the reduction is lossless;
    $M \longleftarrow \emptyset$;
    foreach $(v_1, v_2, M_1, M_2) \in R$ **do**
      $M \longleftarrow M \cup M_1$;
    if Domain$({}_{q_1}\mathbf{M}_{\mathtt{out}\rightsquigarrow \mathtt{v}_1}) \neq$ Domain(M) **then**
      return (errorDiagnostics)
  return R
end

Reduction of a Primitive Proof Obligation.
In the event that the selected primitive proof obligation is not a terminal obligation, it is reduced into successor primitive proof obligations, that are added to the current proof obligations. The reduction of a primitive proof obligation is given by Algorithm 9. It promotes the nodes (Algorithm 10) and then updates the mappings to the new nodes (Algorithm 11). Note that, as a result of reduction, for a correct transformation, there can be a split in the domain leading to slices which distributes

the domain across the reduced proof obligations, but no loss of elements in the domain. Therefore, an important check that still needs to be made following a non-empty reduction is whether the reduction has been *lossless* or not. A lossless reduction implies that the domain of the output array of the proof obligation that is being reduced is preserved under reduction. This helps ensure that we still maintain the full domain of the output array for which we started the equivalence checking traversal. This check is incorporated as shown in Algorithm 9.

Promotion of Nodes in the ADDG.
This operation is called by Algorithm 9 and is described in Algorithm 10. Its purpose is to identify the successor nodes of each of the nodes in the primitive proof obligation in their respective ADDGs and depending on the nature of the two current nodes, return a valid pairing between them. In the first case, when the nodes to be promoted are both (identical) operators, the operators are *peeled* to result in a pairing of the operands returned by the Algorithm 14. If the transformations based on any algebraic properties of the operator like associativity, commutativity, etc., (i.e., the algebraic transformations) have not been applied, the pairing (or matching) of the operands is trivially decided by the operand position. Otherwise, the matching can be quite involved and we describe the necessary theory and algorithms in Section 7.3. In the other valid cases, either of the two nodes is an internal array node or an output array node. The nodes are then promoted by pairing each of the successors of the array node with the other node.

Update of transitive dependency mappings.
This operation is called by Algorithm 9 and is described in Algorithm 11. Once the new node pairs are obtained by promotion of the nodes, its purpose is to update (or extend) the transitive dependency mappings in the obligation to the new nodes. The update takes place only when there exist array nodes on the path between the old node and the new node obtained by its promotion. This is computed by Algorithm 1. Note that an update might result in a mapping with an empty domain, this corresponds to the pairing of nodes on the branches that do not lead to data dependence paths. In such a case, the new pair of nodes does not result in the creation of a new primitive proof obligation. In the event that a recurrence is detected on one or both ADDGs, the reduction is adapted so that it avoids stepping through the recurrence by calling

Algorithm 10: Promotion of nodes in the ADDG.

PromoteNodes(G_1, G_2, p)
Input: ADDGs G_1, G_2 and primitive obligation
$p = (v_1, v_2, {}_{q_1}\mathbf{M}_{out \rightsquigarrow v_1}, {}_{q_2}\mathbf{M}_{out \rightsquigarrow v_2})$ whose nodes are to be promoted.
Output: If successful, a set Q of node pairs obtained by promotion of v_1 and v_2, otherwise, error diagnostics.
begin
  $Q \longleftarrow \emptyset$;
  case v_1 *and* v_2 *are identical operator nodes*
    // Find correspondence relation between their operands;
    // Apply Algorithm 14, page 161;
    $Q \longleftarrow$ MatchNodes(G_1, G_2, p);
  case v_1 *is an internal or output array node*
    $X \longleftarrow$ list of successor nodes of v_1;
    foreach $x \in X$ **do**
      $Q \longleftarrow Q \cup \{(x, v_2)\}$;
  case v_2 *is an internal or output array node*
    $Y \longleftarrow$ list of successor nodes of v_2;
    foreach $y \in Y$ **do**
      $Q \longleftarrow Q \cup \{(v_1, y)\}$;
  otherwise
    return (errorDiagnostics)
  return Q
end

Algorithm 13. This we discuss separately in Section 7.2.3.

Algorithm 11: Update of transitive dependency mappings.

UpdateMappings(G_1, G_2, p, Q)
Input: ADDGs G_1, G_2, a primitive proof obligation $p = (v_1, v_2, {}_{q_1}\mathbf{M}_{\mathtt{out}\rightsquigarrow u_1}, {}_{q_2}\mathbf{M}_{\mathtt{out}\rightsquigarrow u_2})$, and a set Q of node pairs obtained by promotion of nodes v_1 and v_2. Note that, u_1 and u_2 in the transitive dependency mappings in p are the last array nodes on the paths q_1 and q_2 leading to the nodes v_1 and v_2, respectively.
Output: A set R of new primitive proof obligations obtained by reduction of p.

```
begin
    R ⟵ ∅;
    foreach (x, y) ∈ Q do
        q1' ⟵ if x = v1 then q1 else (q1, pathLabel(v1, x)) ;
        q2' ⟵ if y = v2 then q2 else (q2, pathLabel(v2, y)) ;
        // Apply Algorithm 13, page 149;
        (MO, MT) ⟵HandleRecurrences(G1, G2, p, q1', q2');
        // Update mapping in the original program;
        // Apply Algorithm 1, page 58;
        M1' ⟵ ComputeMapping(G1, MO, v1, x);
        // Update mapping in the transformed program;
        M2' ⟵ ComputeMapping(G2, MT, v2, y);
        // Restrict the domains of the updated mappings to the
        // maximal common subset of the domain of out;
        d ⟵ Domain(M1') ∩ Domain(M2');
        if d ≠ ∅ then
            M1 ⟵ RestrictDomain(M1', d);
            M2 ⟵ RestrictDomain(M2', d);
            R ⟵ R ∪ {(x, y, M1, M2)};
    return R
end
```

The traversal selects a new primitive obligation from the updated proof obligation and repeats the process of checking whether it is a terminal obligation, and if it is not, reducing it to obtain a new proof obligation, until all the primitive obligations are exhausted. The method is summarized in Algorithm 12.

Algorithm 12: Outline of the equivalence checker.

```
EquivalenceChecker(G1, G2)
Input: ADDGs G1 and G2 of the two functions.
Output: If they are equivalent, True, else Failure, with error
        diagnostics.
begin
    P ⟵ ProofInitialization(G1, G2);
    while P ≠ ∅ do
        p ⟵ SelectObligation(P);
        if p is a terminal proof obligation then
            if p is not True then
                return (Failure, errorDiagnostics)
        else
            // Reduce p using Algorithm 9;
            newObligations ⟵ ReduceObligation(G1, G2, p);
            if newObligations = ∅ then
                return (Failure, errorDiagnostics)
            else
                P ⟵ (P − {p}) ∪ newObligations;
    return True
end
```

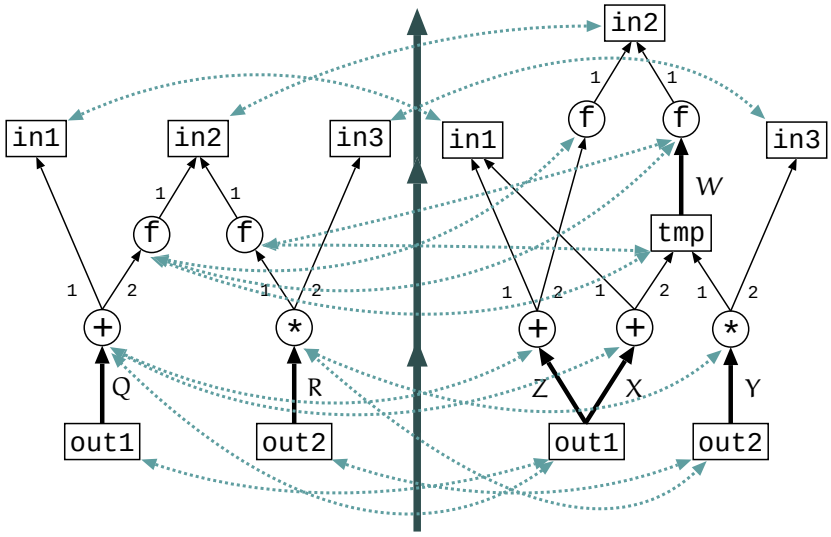


Figure 7.4: The matching nodes established by the equivalence checker between the example programs given in Figure 7.1 on their ADDGs shown in Figure 7.2.

Example 30

For the current example program pair (Figure 7.1), here we show the trace of the working of our equivalence checker. For the purpose of illustration, we maintain the working list of proof obligations as a stack and process the top proof obligation of the stack at each step. In order to aid the following of the trace, we show the node pairs that are matched by the checker in Figure 7.4.

Step 0: Initialization:

$(\texttt{out1}, \texttt{out1}, {}_{()}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}}, {}_{()}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}})$
$(\texttt{out2}, \texttt{out2}, {}_{()}\mathbf{M}_{\texttt{out2}\rightsquigarrow\texttt{out2}}, {}_{()}\mathbf{M}_{\texttt{out2}\rightsquigarrow\texttt{out2}})$
$\quad {}_{()}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}} := \{[i] \rightarrow [i] | 0 \leq i < 3 * N\}$
$\quad {}_{()}\mathbf{M}_{\texttt{out2}\rightsquigarrow\texttt{out2}} := \{[i] \rightarrow [i] | 0 \leq i < N\}$

Step 1: Reduction

$(+, \texttt{out1}, {}_{(Q)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}}, {}_{()}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}})$
$\quad {}_{(Q)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}} := \{[i] \rightarrow [i] | 0 \leq i < 3 * N\}$

$(\texttt{out2}, \texttt{out2}, {}_{()}\mathbf{M}_{\texttt{out2}\rightsquigarrow\texttt{out2}}, {}_{()}\mathbf{M}_{\texttt{out2}\rightsquigarrow\texttt{out2}})$

Step 2: Reduction

$(+, +, {}_{(Q)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}}, {}_{(Z)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}})$
$\quad {}_{(Q)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}} := \{[i] \rightarrow [i] | N \leq i < 3 * N\}$
$\quad {}_{(Z)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}} := \{[i] \rightarrow [i] | N \leq i < 3 * N\}$
$(+, +, {}_{(Q)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}}, {}_{(X)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}})$
$\quad {}_{(Q)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}} := \{[i] \rightarrow [i] | 0 \leq i < N\}$
$\quad {}_{(X)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}} := \{[i] \rightarrow [i] | 0 \leq i < N\}$

$(\texttt{out2}, \texttt{out2}, {}_{()}\mathbf{M}_{\texttt{out2}\rightsquigarrow\texttt{out2}}, {}_{()}\mathbf{M}_{\texttt{out2}\rightsquigarrow\texttt{out2}})$

Step 3: Reduction

$(\texttt{in1}, \texttt{in1}, {}_{(Q,1)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{in1}}, {}_{(Z,1)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{in1}})$
$\quad {}_{(Q,1)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{in1}} := \{[i] \rightarrow [i+1] | N \leq i < 3 * N\}$
$\quad {}_{(Z,1)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{in1}} := \{[i] \rightarrow [i+1] | N \leq i < 3 * N\}$
$(\texttt{f}, \texttt{f}, {}_{(Q,2)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}}, {}_{(Z,2)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}})$
$\quad {}_{(Q,2)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}} := \{[i] \rightarrow [i] | N \leq i < 3 * N\}$
$\quad {}_{(Z,2)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}} := \{[i] \rightarrow [i] | N \leq i < 3 * N\}$

$(+, +, {}_{(Q)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}}, {}_{(X)}\mathbf{M}_{\texttt{out1}\rightsquigarrow\texttt{out1}})$
$(\texttt{out2}, \texttt{out2}, {}_{()}\mathbf{M}_{\texttt{out2}\rightsquigarrow\texttt{out2}}, {}_{()}\mathbf{M}_{\texttt{out2}\rightsquigarrow\texttt{out2}})$

Step 4: Terminal Obligation
TRUE
———————
$(\mathtt{f},\mathtt{f},{}_{(\mathrm{Q},2)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{out1}},{}_{(\mathrm{Z},2)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{out1}})$
$(+,+,{}_{(\mathrm{Q})}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{out1}},{}_{(\mathrm{X})}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{out1}})$
$(\mathtt{out2},\mathtt{out2},{}_{()}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}},{}_{()}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}})$
———————

Step 5: Reduction
$(\mathtt{in2},\mathtt{in2},{}_{(\mathrm{Q},2,1)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{in2}},{}_{(\mathrm{Z},2,1)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{in2}})$
$\quad{}_{(\mathrm{Q},2,1)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{in2}} := \{[\mathrm{i}] \rightarrow [\mathrm{i}+\mathrm{N}] | \mathrm{N} \leq \mathrm{i} < 3*\mathrm{N}\}$
$\quad{}_{(\mathrm{Z},2,1)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{in2}} := \{[\mathrm{i}] \rightarrow [\mathrm{i}+\mathrm{N}] | \mathrm{N} \leq \mathrm{i} < 3*\mathrm{N}\}$
———————
$(+,+,{}_{(\mathrm{Q})}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{out1}},{}_{(\mathrm{X})}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{out1}})$
$(\mathtt{out2},\mathtt{out2},{}_{()}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}},{}_{()}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}})$
———————

Step 6: Terminal Obligation
TRUE
———————
$(+,+,{}_{(\mathrm{Q})}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{out1}},{}_{(\mathrm{X})}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{out1}})$
$(\mathtt{out2},\mathtt{out2},{}_{()}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}},{}_{()}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}})$
———————

Step 7: Reduction
$(\mathtt{in1},\mathtt{in1},{}_{(\mathrm{Q},1)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{in1}},{}_{(\mathrm{X},1)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{in1}})$
$\quad{}_{(\mathrm{Q},1)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{in1}} := \{[\mathrm{i}] \rightarrow [\mathrm{i}+1] | 0 \leq \mathrm{i} < \mathrm{N}\}$
$\quad{}_{(\mathrm{X},1)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{in1}} := \{[\mathrm{i}] \rightarrow [\mathrm{i}+1] | 0 \leq \mathrm{i} < \mathrm{N}\}$
$(\mathtt{f},\mathtt{tmp},{}_{(\mathrm{Q},2)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{out1}},{}_{(\mathrm{X},2)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{tmp}})$
$\quad{}_{(\mathrm{Q},2)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{out1}} := \{[\mathrm{i}] \rightarrow [\mathrm{i}] | 0 \leq \mathrm{i} < \mathrm{N}\}$
$\quad{}_{(\mathrm{X},2)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{tmp}} := \{[\mathrm{i}] \rightarrow [\mathrm{i}] | 0 \leq \mathrm{i} < \mathrm{N}\}$
———————
$(\mathtt{out2},\mathtt{out2},{}_{()}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}},{}_{()}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}})$
———————

Step 8: Terminal Obligation
TRUE
———————
$(\mathtt{f},\mathtt{tmp},{}_{(\mathrm{Q},2)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{out1}},{}_{(\mathrm{X},2)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{tmp}})$
$(\mathtt{out2},\mathtt{out2},{}_{()}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}},{}_{()}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}})$
———————

Step 9: Reduction

$(\mathtt{f}, \mathtt{f}, {}_{(\mathrm{Q},2)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{out1}}, {}_{(\mathrm{X},2,W)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{tmp}})$

$\quad {}_{(\mathrm{Q},2)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{out1}} := \{[\mathrm{i}] \rightarrow [\mathrm{i}] | 0 \leq \mathrm{i} < \mathrm{N}\}$

$\quad {}_{(\mathrm{X},2,W)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{tmp}} := \{[\mathrm{i}] \rightarrow [\mathrm{i}] | 0 \leq \mathrm{i} < \mathrm{N}\}$

$(\mathtt{out2}, \mathtt{out2}, {}_{()}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}}, {}_{()}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}})$

Step 10: Reduction

$(\mathtt{in2}, \mathtt{in2}, {}_{(\mathrm{Q},2,1)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{in2}}, {}_{(\mathrm{X},2,W,1)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{in2}})$

$\quad {}_{(\mathrm{Q},2,1)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{out1}} := \{[\mathrm{i}] \rightarrow [\mathrm{i}+\mathrm{N}] | 0 \leq \mathrm{i} < \mathrm{N}\}$

$\quad {}_{(\mathrm{X},2,W,1)}\mathbf{M}_{\mathtt{out1}\rightsquigarrow\mathtt{tmp}} := \{[\mathrm{i}] \rightarrow [\mathrm{i}+\mathrm{N}] | 0 \leq \mathrm{i} < \mathrm{N}\}$

$(\mathtt{out2}, \mathtt{out2}, {}_{()}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}}, {}_{()}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}})$

Step 11: Terminal Obligation

TRUE

$(\mathtt{out2}, \mathtt{out2}, {}_{()}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}}, {}_{()}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}})$

Step 12: Reduction

$(*, \mathtt{out2}, {}_{(\mathrm{R})}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}}, {}_{()}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}})$

$\quad {}_{(\mathrm{R})}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}} := \{[\mathrm{i}] \rightarrow [\mathrm{i}] | 0 \leq \mathrm{i} < \mathrm{N}\}$

Step 13: Reduction

$(*, *, {}_{(\mathrm{R})}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}}, {}_{(\mathrm{Y})}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}})$

$\quad {}_{(\mathrm{Y})}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}} := \{[\mathrm{i}] \rightarrow [\mathrm{i}] | 0 \leq \mathrm{i} < \mathrm{N}\}$

Step 14: Reduction

$(\mathtt{f}, \mathtt{tmp}, {}_{(\mathrm{R},1)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}}, {}_{(\mathrm{Y},1)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{tmp}})$

$\quad {}_{(\mathrm{R},1)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}} := \{[\mathrm{i}] \rightarrow [\mathrm{i}] | 0 \leq \mathrm{i} < \mathrm{N}\}$

$\quad {}_{(\mathrm{Y},1)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{tmp}} := \{[\mathrm{i}] \rightarrow [\mathrm{i}] | 0 \leq \mathrm{i} < \mathrm{N}\}$

$(\mathtt{in3}, \mathtt{in3}, {}_{(\mathrm{R},2)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{in3}}, {}_{(\mathrm{Y},2)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{in3}})$

$\quad {}_{(\mathrm{R},2)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{in3}} := \{[\mathrm{i}] \rightarrow [\mathrm{i}+\mathrm{N}] | 0 \leq \mathrm{i} < \mathrm{N}\}$

$\quad {}_{(\mathrm{Y},2)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{in3}} := \{[\mathrm{i}] \rightarrow [\mathrm{i}+\mathrm{N}] | 0 \leq \mathrm{i} < \mathrm{N}\}$

Step 15: Reduction

$(\mathtt{f}, \mathtt{f}, {}_{(\mathrm{R},1)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}}, {}_{(\mathrm{Y},1,W)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{tmp}})$

$_{(\mathtt{R},1)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{out2}} := \{[\mathtt{i}] \rightarrow [\mathtt{i}] | 0 \leq \mathtt{i} < \mathtt{N}\}$
$_{(Y,1,W)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{tmp}} := \{[\mathtt{i}] \rightarrow [\mathtt{i}] | 0 \leq \mathtt{i} < \mathtt{N}\}$

———————

$(\mathtt{in3}, \mathtt{in3}, {}_{(\mathtt{R},2)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{in3}}, {}_{(Y,2)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{in3}})$

———————————

Step 16: Reduction

$(\mathtt{in2}, \mathtt{in2}, {}_{(\mathtt{R},1,1)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{in2}}, {}_{(Y,1,W,1)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{in2}})$
$_{(\mathtt{R},1,1)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{in2}} := \{[\mathtt{i}] \rightarrow [\mathtt{i}] | 0 \leq \mathtt{i} < \mathtt{N}\}$
$_{(Y,1,W,1)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{in2}} := \{[\mathtt{i}] \rightarrow [\mathtt{i}] | 0 \leq \mathtt{i} < \mathtt{N}\}$

———————

$(\mathtt{in3}, \mathtt{in3}, {}_{(\mathtt{R},2)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{in3}}, {}_{(Y,2)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{in3}})$

———————————

Step 17: Terminal Obligation

TRUE

———————

$(\mathtt{in3}, \mathtt{in3}, {}_{(\mathtt{R},2)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{in3}}, {}_{(Y,2)}\mathbf{M}_{\mathtt{out2}\rightsquigarrow\mathtt{in3}})$

———————————

Step 18: Terminal Obligation

TRUE

———————————

All the proof obligations have now been disposed and the equivalence checker successfully terminates. This proves the equivalence of the original and the transformed programs given in Figure 7.1.

7.2.3 Handling Recurrences in ADDGs

We have discussed what constitutes a recurrence in a path in Section 4.5 and how an across-recurrence mapping is computed. In this section, we discuss how the method deals with recurrences during its traversal relying on the across-recurrence mappings.

When the nodes of a primitive proof obligation are promoted during reduction and their transitive dependency mappings are updated (Algorithm 11), it is required that we check whether in arriving at the new nodes the traversal has entered a recurrence. When this is the case, depending on the nodes that appear in the cycle of recurrence, we distinguish two possible cases of recurrences in an ADDG, viz., a recurrence

```
tmp[0] = f2(in[0]);                    // X
for (k = 1; k < 256; k++)
    tmp[k] = tmp[k-1];                 // Y
out[0] = f1(tmp[255]);                 // Z
```

Figure 7.5: A program with a recurrence over a copy.

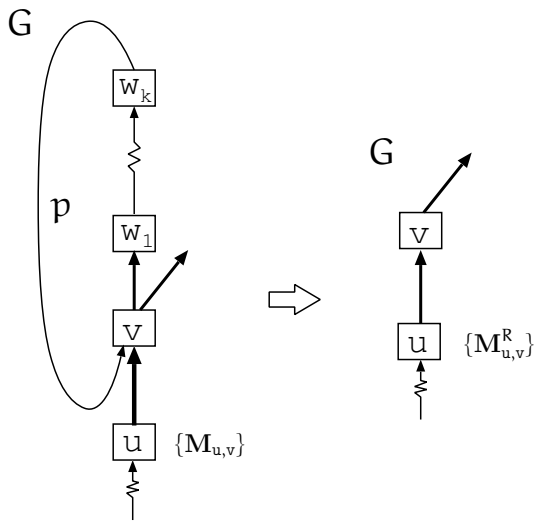


Figure 7.6: Updating the transitive dependency mapping when there exists recurrence over only copies.

over only copies and recurrence over computation. Each case of recurrence is handled separately as discussed below. The outline of the steps is presented in Algorithm 13.

Recurrences over only copies.
In this case, no operator nodes are present in the recurrence cycle and hence, the cycle does not contribute to the path signature. This implies that the recurrence has no influence on the matching and it suffices to record the overall transitive dependency mapping of the recurrence. Therefore, during traversal (or during array node elimination), if such a recurrence is encountered on a given path, the across-recurrence mapping is computed making use of Algorithm 2 and this essentially elimi-

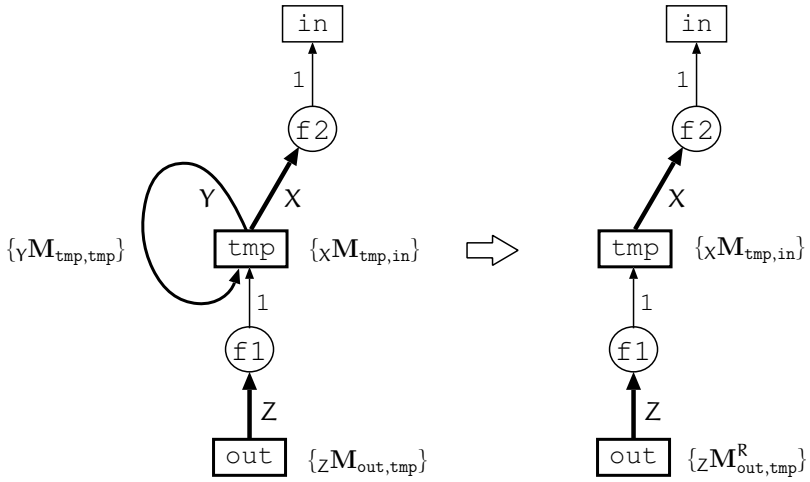


Figure 7.7: ADDG of the example program in Figure 7.5.

nates the cycle on the path. This is illustrated in the Figure 7.6, where `v[]` is the array at the entry to the cycle and no operator nodes exist on the path `p`. If we would unroll the loop (even partially), it is clear that the intermediate nodes can just be substituted into one another and no trace would be left of them: no operators are accumulating because only the identity operation is present. So this is only a "non-functional" type of recurrence that is due to a specific use of the loop construct to produce identical copies (typically for use further on in another loop nest).

Example 31

Figure 7.5 shows a simple example program with a recurrence over only copies. Its ADDG is as shown in Figure 7.7, along with the ADDG after the recurrence has been eliminated on its only path. The mappings are as given below.

$_{Z}\mathbf{M}_{\mathtt{out},\mathtt{tmp}} := \{[0] \rightarrow [255]\}$

$_{Y}\mathbf{M}_{\mathtt{tmp},\mathtt{tmp}} := \{[k] \rightarrow [k-1] \mid 1 \leq k \leq 255 \wedge k \in \mathbb{Z}\}$

$_{X}\mathbf{M}_{\mathtt{tmp},\mathtt{in}} := \{[0] \rightarrow [0]\}$

The across-recurrence mapping, computed as discussed in Example 6 on Page 64 results in

$_{Z}\mathbf{M}^{R}_{\mathtt{out},\mathtt{tmp}} := \{[0] \rightarrow [0]\}$.

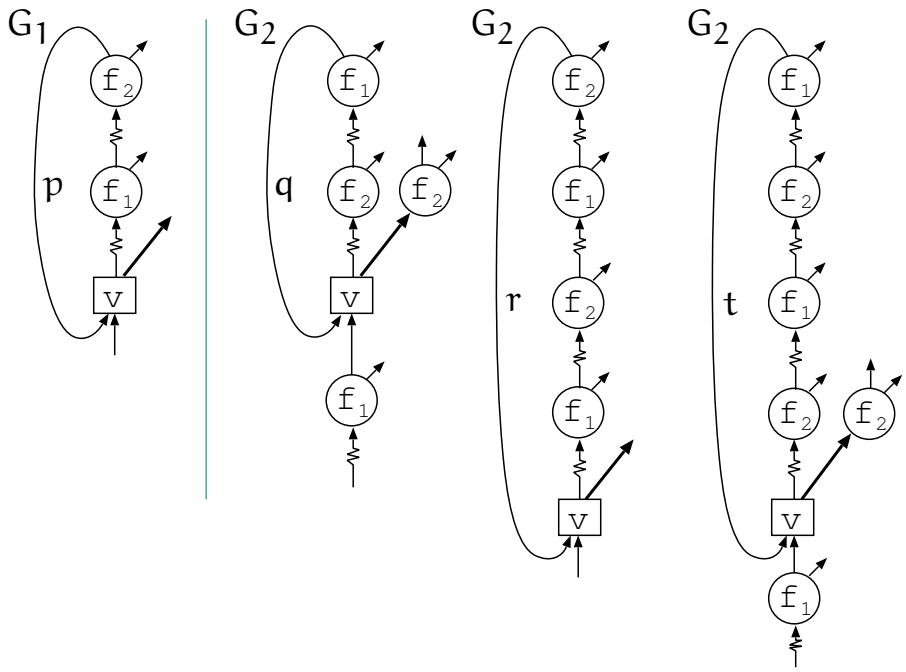


Figure 7.8: Recurrence over computation: Base cases in the ADDG of the transformed program.

Recurrences over computation.
In this case, operator nodes are present in the recurrence cycle and hence, they contribute to the path signature. Unrolling the loop would not allow to remove the dependence chain here to a single node: the operators will keep on growing in complexity. When it is detected on a path in one of the ADDGs, unless a complete unrolling transformation has been applied, a corresponding recurrent loop construct is also detected on the other ADDG. When confronted with this recurrence, it is required that we ensure that the across-recurrence mapping is computed in such a way that it matches an identical number of operators on both the matching paths. That is, we need to ensure that the new dependency mappings computed are across the same signatures.

In order to be able to compute the across-recurrence mapping on the two

corresponding paths together, we first have to get an identical sequence of operators on the recurrence cycles on both paths. This is achieved by *unfolding*. We explain this in the following.

Suppose G_1 and G_2 are the ADDGs being traversed in synchronization and we detect a recurrence on one of them, say, G_1, with $(f_1, \ldots, f_k, f_1)$ as operator nodes on the cycle. The traversal ensures that the corresponding nodes traversed on G_2 are also $(f_1, \ldots, f_k, f_1)$. If a recurrence is also detected at this point on G_2, we are done. Otherwise, we step through the recurrence in G_1 along with G_2 as long as it takes to reveal a cycle with an identical sequence of operators on G_2. It just implies that we decide the anchor points on the two cycles such that both cycles have the same signatures. This procedure will always converge, assuming that no algebraic transformations have been applied to the body of the recurrence loop. That is an assumption we have anyway made upfront in this chapter.

Figure 7.8 shows G_1 with cycle p and G_2 with the basic possibilities for a cycle, viz., operators shifted by one (q), unfolded once completely (r) and both unfolded once and shifted by one (t).

Once we have established matching cycles on the two sides by sufficient unfolding, we have transitive dependency mappings for the two corresponding cycles, namely

$$M_1 := \{[\vec{a_1}] \rightarrow [\vec{a_2}] \,|\, C_1\}$$

and

$$M_2 := \{[\vec{c_1}] \rightarrow [\vec{c_2}] \,|\, C_2\},$$

where C_1 and C_2 are affine constraint expressions. Now, in order to compute the recurrence mapping that ensures the same computation on both sides, we combine the two transitive dependency mappings and use the *combined mapping* M as the dependency mapping for the cycle, given by,

$$M := \{[\vec{a_1}, \vec{c_1}] \rightarrow [\vec{a_2}, \vec{c_2}] \,|\, C_1 \wedge C_2\}.$$

This combined mapping is used for the computation of the mapping M' as described in the Algorithm 2. M' is then split into M'_1 and M'_2 along the same indices that were combined earlier. However, the outcome is (usually) not identical to the original mappings M_1 and M_2. So that is why we need this intermediate enabling step in order to prepare the

```
buf[0] = f1(f2(in[0]));              // Q
for (k = 1; k < 256; k++)
    buf[k] = f1(f2(buf[k-1]));       // R
out[0] = buf[255];                   // S
```

(a) A program having a recurrence over computation

```
tmp[0] = f2(in[0]);                  // X
for (k = 1; k < 256; k++)
    tmp[k] = f2(f1(tmp[k-1]));       // Y
out[0] = f1(tmp[255]);               // Z
```

(b) A program equivalent to the program in (a)

Figure 7.9: Equivalent programs with a recurrence over computation.

input for the next step. The new split mappings are used in calculating the across-recurrence mappings on the respective ADDGs.

Example 32

In this example, we demonstrate the functioning of the HandleRecurrences() function with a simple pair of original and transformed programs. Figure 7.9(a) and (b) show an example of a pair of equivalent programs that has a recurrence with computation. Their ADDGs are as shown in Figure 7.10. It is clear from the figure that the operator is shifted by one in the ADDG of the transformed program.

Proof initialization gives a single primitive proof obligation:

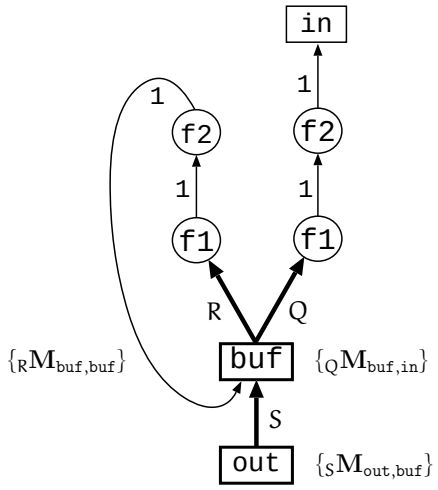
$P := (\texttt{out}, \texttt{out}, {}_{()}\mathbf{M}_{\texttt{out}\rightsquigarrow\texttt{out}}, {}_{()}\mathbf{M}_{\texttt{out}\rightsquigarrow\texttt{out}})$

The reduction of the above obligation proceeds as follows:

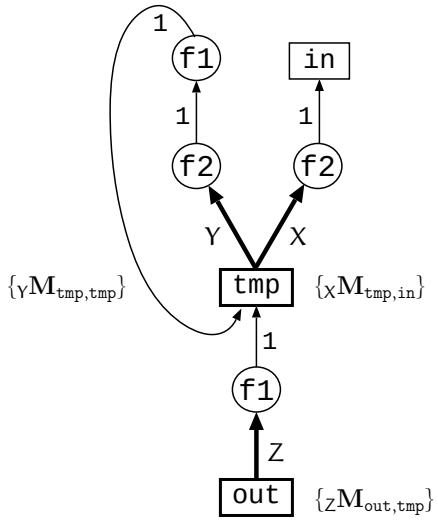
Original		Transformed	
Path	Node	Node	Path
()	out	out	()
(S)	buf	out	()
(S,R)	f1	out	()
(S,R)	f1	f1	(Z)
(S,R,1)	f2	tmp	(Z,1)
(S,R,1)	f2	f2	(Z,1,Y)
(S,R,1,1)	buf	f1	(Z,1,Y,1)
(S,R,1,1,R)	f1	f1	(Z,1,Y,1)

Algorithm 13: Handle recurrences in the paths.

HandleRecurrences$(G_1, G_2, p, q'_1, q'_2)$
Input: ADDGs G_1, G_2, a primitive proof obligation $p = (v_1, v_2, {}_{q_1}\mathbf{M}_{\mathtt{out}\rightsquigarrow \mathtt{v}_1}, {}_{q_2}\mathbf{M}_{\mathtt{out}\rightsquigarrow \mathtt{v}_2})$, and the updated paths q'_1 and q'_2 for the promotion of obligation p.
Output: Across-recurrence mapping(s).
begin
  if q'_1 *and* q'_2 *both imply a recurrence over computation* **then**
    Let c_1 and c_2 be the recurrence cycles on the two sides and M_1 and M_2 their transitive dependency mappings;
    Construct combined mapping M from M_1 and M_2;
    // Algorithm 2 on page 62;
    $M' \longleftarrow$ AcrossRecurrenceMapping(M);
    Split M' into corresponding mappings, M'_1 and M'_2;
    ${}_{q'_1}\mathbf{M}^{R}_{\mathtt{out},v_1} \longleftarrow {}_{q_1}\mathbf{M}_{\mathtt{out}\rightsquigarrow \mathtt{v}_1} \bowtie M'_1$;
    ${}_{q'_2}\mathbf{M}^{R}_{\mathtt{out},v_2} \longleftarrow {}_{q_2}\mathbf{M}_{\mathtt{out}\rightsquigarrow \mathtt{v}_2} \bowtie M'_2$;
    $M_O \longleftarrow {}_{q'_1}\mathbf{M}^{R}_{\mathtt{out},v_1}$;
    $M_T \longleftarrow {}_{q'_2}\mathbf{M}^{R}_{\mathtt{out},v_2}$;
  else
    // Process q'_1;
    if q'_1 *implies a recurrence over only copies* **then**
      Let c be the recurrence cycle and M its transitive dependency mapping;
      ${}_{c}\mathbf{M}'_{v_1,v_1} \longleftarrow$ AcrossRecurrenceMapping(M);
      ${}_{q'_1}\mathbf{M}^{R}_{\mathtt{out},v_1} \longleftarrow {}_{q_1}\mathbf{M}_{\mathtt{out}\rightsquigarrow \mathtt{v}_1} \bowtie {}_{c}\mathbf{M}'_{v_1,v_1}$;
      $M_O \longleftarrow {}_{q'_1}\mathbf{M}^{R}_{\mathtt{out},v_1}$;
    else if q'_1 *is a recurrence over computation* **then**
      Record that recurrence over computation has been detected;
      $M_O \longleftarrow {}_{q_1}\mathbf{M}_{\mathtt{out}\rightsquigarrow \mathtt{v}_1}$;
    else
      $M_O \longleftarrow {}_{q_1}\mathbf{M}_{\mathtt{out}\rightsquigarrow \mathtt{v}_1}$;
    Process q'_2 similar to q'_1 in computing M_T;
  return (M_O, M_T)
end



(a) The ADDG of the original program



(b) The ADDG of the transformed program

Figure 7.10: The ADDGs of the example program pair with recurrence over computation.

Here a recurrence is detected in the original program when the traversal arrives at node `f1` through array node `buf` along the previously traversed assignment statement R. But, at this point, the recurrence is yet to be detected on the path in the transformed program. Therefore, for the moment we only note already that a recurrence has been detected in the original (Line 21 of HandleRecurrences()) and proceed.

(S,R,1,1,R,1)	f2	tmp	(Z,1,Y,1,1)
(S,R,1,1,R,1)	f2	f2	(Z,1,Y,1,1,Y)

Here we see that the reduction has revealed a cycle on the path in the transformed program as well. The signature of the corresponding recurrences is found to be $(\texttt{f2}, \texttt{f1})$ and corresponding nodes are $(\texttt{f2}, \texttt{f2})$.

The current transitive dependency mappings of the matching cycles, $(1, 1, R, 1)$ and $(Y, 1, 1, Y)$ are:

$_{(1,1,R,1)}\mathbf{M}_{\texttt{buf}\rightsquigarrow\texttt{buf}} := \{[k] \rightarrow [k-1] \,|\, 1 \leq k \leq 254 \wedge k \in \mathbb{Z}\}$, and

$_{(Y,1,1,Y)}\mathbf{M}_{\texttt{tmp}\rightsquigarrow\texttt{tmp}} := \{[k] \rightarrow [k-1] \,|\, 1 \leq k \leq 254 \wedge k \in \mathbb{Z}\}$

The above mappings correspond to mappings M_1 and M_2 referred to in Line 4. Now, their combined mapping as constructed at Line 5 is given by:

$M := \{[k, k'] \rightarrow [k-1, k'-1] \,|\, 1 \leq k \leq 254 \wedge k \in \mathbb{Z} \wedge 1 \leq k' \leq 254 \wedge k' \in \mathbb{Z}\}$.

The transitive dependency mapping between the two ends of the recurrence, that is, the across-recurrence mapping obtained using Algorithm 2 is as given below:

$M' := \{[254, 254] \rightarrow [0, 0]\}$. (Line 6)

Next, mapping M' is split into corresponding mappings, M'_1 and M'_2 along the two respective indices. They are given by,

$M'_1 := \{[254] \rightarrow [0]\}$, and

$M'_2 := \{[254] \rightarrow [0]\}$.

These are joined with the transitive dependency mapping that led into the recurrence, to obtain the following mappings.

$$
\begin{aligned}
{}_{(S,R,(1,1,R,1)^+)}\mathbf{M}^{R}_{out,buf} &:= \{[0] \rightarrow [254]\} \bowtie \{[254] \rightarrow [0]\} \\
&:= \{[0] \rightarrow [0]\}, \text{and} \\
{}_{(Z,1,(Y,1,1,Y)^+)}\mathbf{M}^{R}_{out,tmp} &:= \{[0] \rightarrow [254]\} \bowtie \{[254] \rightarrow [0]\} \\
&:= \{[0] \rightarrow [0]\}
\end{aligned}
$$

The above mappings are returned (Line 26) by HandleRecurrences() to UpdateMappings(), which results in the following obligation to proceed further.

$(\texttt{f2}, \texttt{f2}, \{[0] \rightarrow [0]\}, \{[0] \rightarrow [0]\})$.

From the next reduction we have a correspondence between `buf[0]` and `f1(tmp[0])`. But by statements Q and X, we have `buf[0] = f1(f2(in[0]))` and `tmp[0] = f2(in[0])`, respectively. Therefore, after two more reduction steps, we have the terminal obligation $(\texttt{in}, \texttt{in}, \{[0] \rightarrow [0]\}, \{[0] \rightarrow [0]\})$, which holds. This concludes the formal equivalence proof.

7.3 Verification of Loop and Data-Flow Transformations

In the previous section, we have presented our operator-level equivalence checking method that is able to verify a combination of loop and expression propagation transformations. But in order to be able to handle the full category of data-flow transformations, we need to be able to verify the algebraic transformations as well. In this section, we add the additional capability to the previously discussed method and hence obtain a general equivalence checking method that can verify any combination of the transformations from the full categories of loop and data-flow transformations.

Recall that, in the previous method, we have relied on the existence of matching paths and matching slices in establishing the correspondence between the data-flows of the original and the transformed programs. This has sufficed since the applied transformations treated the operators as uninterpreted functions and did not take advantage of their properties. But when the operators are interpreted and their algebraic properties exploited by the transformations, the existence of matching paths can no longer be guaranteed between the data-flows of the two programs. Therefore, additional work that is described in this section is needed in order to discover the hidden correspondence between the two data-flows. In brief, this additional work is required whenever the traversal discussed in the previous section encounters an operator node, whose algebraic properties it is aware of, on a path. At that point, it employs a specific *demand-driven normalization* that is able to generate a normal form that can be used to establish the correspondence with a matching path in the transformed program.

In Section 7.3.1, we first present a sufficient condition for the equiva-

```
for (k = 0; k < 256; k++)
    tmp1[k] = in1[2*k] + f(in2[k+1]);          // A
for (k = 10; k < 138; k++)
    tmp2[k] = in2[k-8];                        // B
for (k = 10; k < 266; k++){
    if (k >= 138)
       tmp2[k] = in2[k-8];                     // C
    tmp3[k-10] = f(in1[2*k-19]) + tmp2[k];     // D
}
for (k = 255; k >= 0; k--)
    out[3*k] = tmp1[k] + tmp3[k];              // E
```

(a) Original program

```
for (k = 0; k < 256; k++){
    buf1[k] = f(in1[2*k+1]) + in1[2*k];        // X
    buf2[k] = in2[k+2] + buf1[k];              // Y
    out[3*k] = f(in2[k+1]) + buf2[k];          // Z
}
```

(b) Transformed program

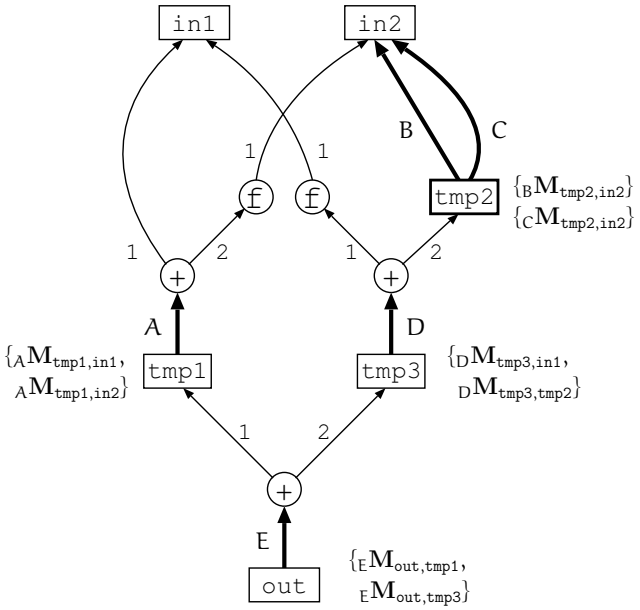
Figure 7.11: An example program pair under combined loop and data-flow transformations.

lence of two programs under loop and data-flow transformations and then in Section 7.3.2, we discuss how this is plugged into the previously discussed equivalence checking method for loop and expression propagation transformations.

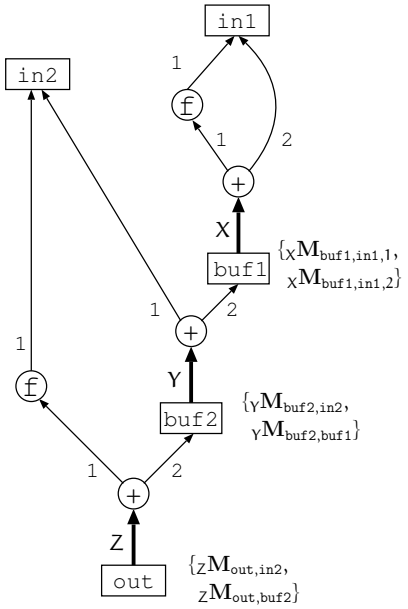
Example 33

Consider the program pair shown in Figure 7.11. The transformed program has been obtained from the original through a combination of loop transformations, expression propagations and algebraic transformations. The programs take input values from arrays `in1[]` and `in2[]` and assign the computed values to the array `out[]`. The original program assigns values to `out[]` as, $\forall \mathtt{k} : 0 \leq \mathtt{k} < 256$,

$$\mathtt{out}[3*\mathtt{k}] = \mathtt{in1}[2*\mathtt{k}] + \mathtt{f}(\mathtt{in2}[\mathtt{k}+1])] + \mathtt{f}(\mathtt{in1}[2*\mathtt{k}+1]) + \mathtt{in2}[\mathtt{k}+2],$$



(a) The ADDG of the original program



(b) The ADDG of the transformed program

Figure 7.12: The ADDGs of the example program pair for combined loop and data-flow transformations.

whereas, the transformed function assigns `out[]` as,

$\texttt{out}[3*\texttt{k}] = \texttt{f}(\texttt{in2}[\texttt{k}+1]) + \texttt{in2}[\texttt{k}+2] + \texttt{f}(\texttt{in1}[2*\texttt{k}+1]) + \texttt{in1}[2*\texttt{k}].$

It can be observed that there is a correspondence between the terms in the function computed by the original program and the one computed by the transformed program. The index expressions too are identical in the input array variables in the terms. Obviously, if we can ignore a possible overflow in the evaluation of integer expressions, the integer addition is both associative and commutative. Therefore, both programs are functionally equivalent. However, observing the ADDGs of the two programs that are shown in Figure 7.12, we see that there are paths in the ADDG of the original program that have no matching in the ADDG of the transformed program. Therefore, the sufficient condition given in Definition 7.2.6 in page 128 for verification of loop and expression propagations no longer suffices to verify the current program pair that is under loop and data-flow transformations.

7.3.1 Sufficient Condition for Equivalence

Let us first recall the discussion on algebraic transformations we had in Section 5.3.3 with particular reference to Figure 5.15 in page 86.

We noted that, in a given slice, when an associative transformation has been applied at an operator node, the nodes at the ends of the associative chains that are rooted at the node may only be regrouped while the original order is maintained. Therefore, irrespective of how the grouping is transformed by the transformation, the number of end nodes remain the same and the new set of associative chains are such that their end nodes are in the same order as before the transformation. This implies that the root operator node for which the associative property holds and has n end nodes can essentially be viewed as a n-ary operator. Such a view serves as a *normal form* for slices that are under associative transformations. Given the root operator node, the flattening operation that we presented in Section 5.4.2 generates this normal form.

Let us now observe the effect of flattening on the signature of a path (Definition 7.2.1) with an associative chain. If $\oplus$ is an associative operator, then the signature of such a path p has the following template prior to flattening.

$$\mathbf{e}_p := (v, o_1, l_1, \ldots, o_{i-1}, l_{i-1}, \underbrace{\oplus, l_i, \ldots, \oplus, l_j}_{\text{only } \oplus \text{ operators}}, o_{j+1}, l_{j+1}, \ldots, o_r, l_r, w).$$

Once the flattening operation has been applied at the root $\oplus$ node, in the obtained normal form, the above signature is transformed into the following signature.

$$\mathfrak{a}(\mathbf{e}_p) := (v, o_1, l_1, \ldots, o_{i-1}, l_{i-1}, \oplus, l', o_{j+1}, l_{j+1}, \ldots, o_r, l_r, w).$$

In the new signature, the chain of $j-i+1$ $\oplus$ nodes is replaced by a single $\oplus$ node and it gets an adjacent label l' that gives the position of the associative chain in the order of end nodes after flattening. Note that when there are no associative operators on the path, we assume that $\mathfrak{a}(\mathbf{e}_p) := \mathbf{e}_p$. We can now extend the above to the signature of a slice. If the signature of a slice g, with k paths to inputs, prior to flattening is given by

$$\mathbf{E}_g := (\mathbf{e}_1, \ldots, \mathbf{e}_k),$$

then the signature of the normal form of the slice obtained via flattening is given by

$$\mathcal{A}(\mathbf{E}_g) := (\mathfrak{a}(\mathbf{e}_1), \ldots, \mathfrak{a}(\mathbf{e}_k)).$$

It can be observed that, once the slices with associative operators in them have been flattened and their normal forms obtained, there exists a matching relation between the slices in their normal forms.

We now turn to the commutative transformation. The effect of the transformation, as we have earlier noted, is that the argument positions of the successor nodes of a commutative operator may be permuted. This implies that the labels of the outgoing-edges of the commutative operators no longer qualify to be part of a path signature. Therefore, when faced with commutative transformations, we can summarily drop them from our signatures. Suppose we have a path p with a commutative operator $\otimes$ on it and having the following signature.

$$\mathbf{e}_p := (v, o_1, l_1, \ldots, o_{i-1}, l_{i-1}, \otimes, l_i, o_{i+1}, l_{i+1}, \ldots, o_r, l_r, w).$$

We define the following function that drops the label l_i that is to the right of the $\otimes$ operator.

$$\mathfrak{c}(\mathbf{e}_p) := (v, o_1, l_1, \ldots, o_{i-1}, l_{i-1}, \otimes, o_{i+1}, l_{i+1}, \ldots, o_r, l_r, w).$$

Again, note that, when there are no commutative operators on the path, we assume that $\mathfrak{c}(\mathbf{e}_p) := \mathbf{e}_p$.

We can now extend the definition of a signature for a path with commutative operators to a slice with commutative operators. If the signature of a slice g, with k paths to inputs, is given by

$$\mathbf{E}_g := (\mathbf{e}_1, \ldots, \mathbf{e}_k),$$

then the signature of the slice, with the labels of the outgoing-edges from the commutative operators elided from them, is given by

$$\mathcal{C}(\mathbf{E}_g) := (\mathfrak{c}(\mathbf{e}_1), \ldots, \mathfrak{c}(\mathbf{e}_k)).$$

Note that the slice thus obtained may have matching paths. That is, the Proposition 7.2.3 in page 126 may not hold on it. Therefore, a matching relation between two slices with commutative operators in them need not be unique.

From the discussion we have had above on the associative and commutative transformations, it is clear that the correspondence relation between paths and slices in the ADDGs of the original and the transformed programs under algebraic transformations is of a weaker form. Therefore, we relax the earlier defined relations of matching paths and slices with relations of *weakly matching* paths and slices. We limit ourselves to associative and commutative transformations and define the relations as follows.

Definition 7.3.1 (Weakly matching paths)
Two paths p *and* q *are weakly matching iff* $\mathfrak{c}(\mathfrak{a}(\mathbf{e}_p)) = \mathfrak{c}(\mathfrak{a}(\mathbf{e}_q))$.

Definition 7.3.2 (Weakly matching slices, $\mathbf{E}_g \approx \mathbf{E}_h$)
Let $g = (\mathbf{P}_g, \mathbf{E}_g)$ *and* $h = (\mathbf{P}_h, \mathbf{E}_h)$ *be two slices such that* $\mathcal{C}(\mathcal{A}(\mathbf{E}_g)) = ({}_g\mathbf{e}_1, \ldots, {}_g\mathbf{e}_k)$ *and* $\mathcal{C}(\mathcal{A}(\mathbf{E}_h)) = ({}_h\mathbf{e}_1, \ldots, {}_h\mathbf{e}_k)$. *The slices* g *and* h *are weakly matching slices iff there exists a bijection* m *between the paths of* $\mathcal{C}(\mathcal{A}(\mathbf{E}_g))$ *and* $\mathcal{C}(\mathcal{A}(\mathbf{E}_h))$ *such that*

$$({}_g\mathbf{e}_1, \ldots, {}_g\mathbf{e}_k) = ({}_h\mathbf{e}_{m(1)}, \ldots, {}_h\mathbf{e}_{m(k)}).$$

In what follows, we call the bijective function m in the above definition as the *matching function*. The weakly matching relation on a set of slices defines an equivalence class, that is referred to as the *weak slice class*.

Definition 7.3.3 (Weak slice class)
A weak slice class is a maximal subset of weakly matching slices from the set of all slices in a given ADDG. If $\mathbf{G}$ *is the set of all slices in the ADDG of a program, the set of its weak slice classes is a partition of* $\mathbf{G}$ *that is given by the function* $\tau(\mathbf{G})$.

We can now characterize a weak slice class, $x \in \tau(\mathbf{G})$ by the tuple

$$(\mathbf{P}_x, \mathbf{E}_x),$$

where $\mathbf{P}_x$ is the tuple of unions of the output-to-input mappings of weakly matching paths, that is,

$$\mathbf{P}_x := (\bigcup_{g \in x} {}_g p_1 \mathbf{M}_{v \rightsquigarrow w_{m_g(1)}}, \ldots, \bigcup_{g \in x} {}_g p_k \mathbf{M}_{v \rightsquigarrow w_{m_g(k)}}),$$

where m_g is the matching function that relates paths in slice g such that g is weakly matching with other slices and $\mathbf{E}_x$ is the signature of the weakly matching slices.

Now two programs under the combined loop and data-flow transformations are functionally equivalent if there are weak slice classes in the original and the transformed programs that are weakly matching and all the output-to-input mappings in the original are also in the transformed. We define the sufficient condition for equivalence as follows.

Definition 7.3.4 (Equivalence condition)
If $\mathbf{G}_O$ *and* $\mathbf{G}_T$ *are sets of data dependence slices of the original and the transformed* $\mathscr{D}$ *programs, then* $\forall (\mathbf{P}_O, \mathbf{E}_O) \in \tau(\mathbf{G}_O)$, $\exists (\mathbf{P}_T, \mathbf{E}_T) \in \tau(\mathbf{G}_T)$ *such that the following holds true:*

$$(\mathbf{E}_O \approx \mathbf{E}_T) \wedge \bigwedge_{i=1}^{k} ({}^O\mathbf{M}_{v \rightsquigarrow w_i} \subseteq {}^T\mathbf{M}_{v \rightsquigarrow w_{m(i)}}),$$

where ${}^O\mathbf{M}_{v \rightsquigarrow w_i}$ *and* ${}^T\mathbf{M}_{v \rightsquigarrow w_{m(i)}}$ *are the* i*-th and* $m(i)$*-th elements of* $\mathbf{P}_O$ *and* $\mathbf{P}_T$*, respectively, and the weakly matching slice classes have* k *paths defining them.*

It is important to note that this condition is independent of any particular algebraic transformations. It is defined over weak slice classes in the

ADDGs of the programs and it is on the definition of weakly matching slices that the specific transformations have an impact. This implies that, when it is required to add another transformation that is based on some other algebraic property, we only need to redefine the weakly matching relation between slices.

7.3.2 General Equivalence Checking Method

Observe that the difference between the equivalence condition for loop and data-flow transformations and the previously defined equivalence condition for loop transformations and expression propagations lies in the matching relation between the slices. Otherwise, the two conditions essentially check the same condition over the output-to-input mappings of the slices. This implies that we can extend the equivalence checking method developed previously to implement the present equivalence condition by accounting for the additional work that is required for the matching under algebraic transformations.

The key step in the earlier method that is responsible for establishing the correspondence relation is the PromoteNodes() step shown in Algorithm 10. In this step, notice that there is a call for MatchNodes() that relates the operands for identical operator nodes. In the absence of algebraic transformations, this function trivially returns the identity pairing of operands that is based on their argument positions. But, in the presence of algebraic transformations, this function has to establish the operands based on the particular algebraic properties of the operator in question. The details of this matching process are presented in Algorithm 14.

Matching the Operand Nodes.
Given two identical operator nodes in the proof obligation, the matching function depends on the algebraic property of the operator. The trivial case occurs when no algebraic properties of the operator have been exploited by the transformations, the function matches the operand nodes on either side based on their position as arguments. When the operator is associative or commutative or both (the properties that we consider), the matching function has additional work to do. Unlike the trivial case, when algebraic data-flow transformations are applied, finding a unique matching can only be based on either operator nodes or input

array nodes. Therefore, the main complication in matching operands for operators with algebraic properties like associativity and commutativity comes from the presence of any internal array nodes among the operands. A given internal array node in the operand may be written by multiple statements implying a different operand in its position for each write. Therefore, each array node present as an operand, for each of its writes, successively distributes over the list of operands, until only operator nodes and/or input array nodes remain as the operands. This means, matching has to take place on the corresponding list of operands on either side (determined by the domain of the path from the output array). When the operator is associative, the operand lists are provided by the normalization routine described by Algorithm 4 (page 92) in Section 5.4.2 for flattening associative chains. This routine is called on both the ADDGs, the corresponding lists are established based on their domains, and finally, the nodes for each list pair are matched.

Matching the Operands of Commutative Nodes.
When the operator is commutative, initially we have to unfold any internal array nodes that are present among the operands in each of the two ADDGs as discussed previously. This is achieved by the normalization routine described by the Algorithm 5 (page 93) in Section 5.4.2. Note that, this is a routine originally designed for Algorithm 4 but can be used for flattening internal array nodes only by passing it an extraneous operator ("$\perp$"). After flattening, the operand lists contain only operator nodes and/or input array nodes. The correspondence between the returned operand lists is established based on the common domain and the function HandleCommutativeNodes() (Algorithm 16) is called to pair the nodes for a given list pair. This involves establishing a unique bijection between the operands in the two lists for which we provide here only a description. In the trivial case, each operand in one list has a unique match in the other and they are paired (see Example 34). But in the general case, each operand in one list may have multiple matches in the other list. When an input array node has multiple matches, the bijection can be established based on the domains of the output array node at the origin of their paths (see Example 35). But when an operator node has multiple matches, establishing the bijection is rather involved, again due to the presence of input array nodes among their operands (see Example 36). We mentioned in our discussion that a slice containing commutative operators may not have paths with unique signatures.

Algorithm 14: Match the operands.

MatchNodes(G_1, G_2, p)
Input: ADDGs G_1, G_2 and a primitive obligation $p = (v_1, v_2, {}_{q_1}\mathbf{M}_{out \rightsquigarrow u_1}, {}_{q_2}\mathbf{M}_{out \rightsquigarrow u_2})$ with two identical operator nodes v_1 and v_2, whose successors nodes have to be matched.
Output: If successful, a matching relation between the operands of v_1 and v_2, otherwise, $\emptyset$ indicating a failure.

```
begin
    m ⟵ ∅; ⊙ ⟵ v1.operator; XYList ⟵ ∅;
    if ⊙ is an associative operator then
        // Apply Algorithm 4 on page 92;
        XList ⟵ FlattenAsso(G1, v1, ⊙, q1, domain(q1M out⇝u1));
        YList ⟵ FlattenAsso(G2, v2, ⊙, q2, domain(q2M out⇝u2));
        XYList ⟵ MatchXYLists(XList, YList);
    else
        X ⟵ list of successor nodes of v1;
        Y ⟵ list of successor nodes of v2;
        if arity(X) ≠ arity(Y) then return ∅; // error;
        XYList ⟵ {(X, Y, Domain(q1M out⇝u1))};
    if ⊙ is a commutative operator then
        if ⊙ is not an associative operator then
            // Apply Algorithm 5 on page 93;
            XList ← FlattenArr(G1, v1, ⊥, q1, domain(q1M out⇝u1));
            YList ← FlattenArr(G2, v2, ⊥, q2, domain(q2M out⇝u2));
            XYList ⟵ MatchXYLists(XList, YList);
        foreach (X, Y, Dom_XY) ∈ XYList do
            //Algorithm 16 on page 162;
            m ⟵ m ∪
            HandleCommutativeNodes(G1, G2, p, X, Y, Dom_XY);
    else
        foreach (X, Y, Dom_XY) ∈ XYList do
            if arity(X) ≠ arity(Y) then return ∅; // error;
            Let X = (x1, ..., xk) and Y = (y1, ..., yk);
            for i ⟵ 1 to k do m ⟵ m ∪ {(xi, yi)};
    return m
end
```

Algorithm 15: Match two lists of node-lists based on their output array domains.

MatchXYLists(XList, YList);
Input: Two lists of tuples, wherein each tuple consists of a list of nodes and a domain referring to the output array domain for which they together represent the operands for an operator in a slice.
Output: A list of 3-tuples, wherein each 3-tuple contains a list from the XList, a list from the YList and a domain for which they both represent the operands for an operator in a slice.

```
begin
    XYList ⟵ ∅;
    foreach (X, Dom_X) ∈ XList do
        foreach (Y, Dom_Y) ∈ YList do
            Dom_XY ⟵ Dom_X ∩ Dom_Y;
            if Dom_XY ≠ ∅ then
                if arity(X) ≠ arity(Y) then
                    return ∅; // error;
                XYList ⟵ XYList ∪ {(X, Y, Dom_XY)};
        return XYList
end
```

Algorithm 16: Handle the commutative nodes while matching operands.

HandleCommutativeNodes($G_1, G_2, p, X, Y, Dom_{XY}$)
Input: ADDGs G_1, G_2 and a primitive obligation $p = (v_1, v_2, M_1, M_2)$ with two identical commutative operator nodes v_1 and v_2, whose successors node sets X and Y, with a maximal common output array domain Dom_{XY}, have to be matched.
Output: If successful, a matching relation between the operands of v_1 and v_2, otherwise, $\emptyset$ indicating a failure.

```
begin
    return FindBijection(G_1, G_2, p, X, Y, Dom_XY);
end
```

This implies that for a given operand for a commutative operator in one slice, deciding on the matching operands in the other may, in the worst case, be based on the output-to-input mappings of the paths. This scenario occurs when all the operators, including the final input arrays, are all identical on multiple paths. This decision essentially boils down to an application of the equivalence checking method for different tentative matchings of the operands. However, given the unlikelihood of this scenario in programs seen in practice, we need not call the checker, but instead apply a breadth-first lookahead traversal and eliminate possible candidates for a matching operand.

Example 34

Let us consider the case when HandleCommutativeNodes() is called with the following two lists:

$X = \{\texttt{in1}, \texttt{f}, \texttt{g}\}$; and
$Y = \{\texttt{g}, \texttt{in1}, \texttt{f}\}$.

As can be seen, each operand in the list X has a unique match in the list Y. Therefore, the FindBijection() routine can easily match them.

Example 35

Let us now consider the case when HandleCommutativeNodes() is called with the following two lists:

$X = \{\texttt{in1}, \texttt{f}, \texttt{in1}\}$; and
$Y = \{\texttt{in1}, \texttt{in1}, \texttt{f}\}$.

As can be seen, each in1 in the list X has two matches in the list Y. Therefore, the FindBijection() routine has to look into the transitive dependency mappings in the proof obligation, extended to the input arrays, in order to find a unique matching. Suppose M_1 and M_2 are the transitive dependency mappings to the last array nodes on the paths to the commutative nodes on either sides, respectively, and ${}_{q_1}\mathbf{M}_{x \rightsquigarrow \texttt{in1}}$, ${}_{q_2}\mathbf{M}_{x \rightsquigarrow \texttt{in1}}$ are the dependency mappings upto the first and the second in1 nodes in the X list, and similarly, ${}_{r_1}\mathbf{M}_{y \rightsquigarrow \texttt{in1}}$, ${}_{r_2}\mathbf{M}_{y \rightsquigarrow \texttt{in1}}$ are the dependency mappings upto the first and the second in1 nodes in the Y list. Then the first in1 node in the X list is matched with the first in1 node on the Y list if $M_1 \bowtie {}_{q_1}\mathbf{M}_{x \rightsquigarrow \texttt{in1}} = M_2 \bowtie {}_{r_1}\mathbf{M}_{y \rightsquigarrow \texttt{in1}}$, otherwise, with the second in1 node on the Y list if $M_1 \bowtie {}_{q_1}\mathbf{M}_{x \rightsquigarrow \texttt{in1}} = M_2 \bowtie {}_{r_2}\mathbf{M}_{y \rightsquigarrow \texttt{in1}}$. Once this is done, given that the matching has taken place for the first in1 node, similar checking of the equivalence of mappings for the only remaining second in1 node on the X list is obliged to match with the other unmatched in1 node on the Y list.

```
for (i = 0; i < 10; i++)
 out[i] = in1[i] + f(in1[i]) +  f(in2[i]); // R
```

(a) Original program

```
for (i = 0; i < 10; i++){
 if (i < 5){
  u[i] = in1[i];                          // A
  v[i] = in2[i];                          // B
 } else {
  u[i] = in2[i];                          // C
  v[i] = in1[i];                          // D
 }
 out1[i] = f(u[i]) + in1[i] + f(v[i]);    // E
}
```

(b) Transformed program

Figure 7.13: An example program pair to illustrate a complication that may arise in matching the operand nodes of commutative nodes.

Example 36

Let us now consider the case when HandleCommutativeNodes() is called with the following two lists:

X = {in1, f, f}; and
Y = {f, in1, f}.

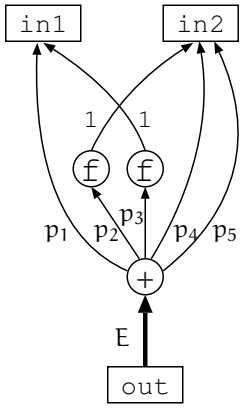
As can be seen, the input array in1 in the X list has a unique match in the Y list, but each f in the list X has two matches in the list Y. Therefore, the FindBijection() routine has to look further at the sub-ADDGs rooted at them in order to establish a unique matching. It is to be noted here that it is possible to have different matchings for different subdomains of Dom_{XY}.

For example, consider the example pair of the original and the transformed programs shown in Figure 7.13 which requires matching with the X and Y lists as given above with $Dom_{XY} = [0..9]$. As can be seen, in the transformed program, there is a split in the domain due to the if condition into $d_1 = [0..4]$ and $d_2 = [5..9]$. Due to this, once the internal arrays u[] and v[] have been unfolded, we can see that the first f node in the X list matches with the first f node in the Y list for the domain $d_1 = [0..4]$ and with the second f node in the Y list for the domain $d_2 = [5..9]$. Similarly, the second f node in the X list also matches with both the f nodes in the Y list, for disjunct sub-domains of Dom_{XY}. The important thing is that the matchings for the same sub-domains lead upto the correct pairing of the input arrays and the union of the sub-domains equals Dom_{XY}, that is it is lossless. Note that, establishing the above described pairing essentially requires calling the equivalence checker for each of the sub-domains and their proof obligations have been successfully disposed in arriving at the valid matching. This implies they need not actually be added to the list of unproven proof obligations. We can obtain this effect by adding true terminal obligations with the respective mappings in their place. Therefore, the pairing (in1, in1) is returned alongwith dummy pairings of input arrays for terminal obligations (for obligations already proved while matching) by FindBijection() here.

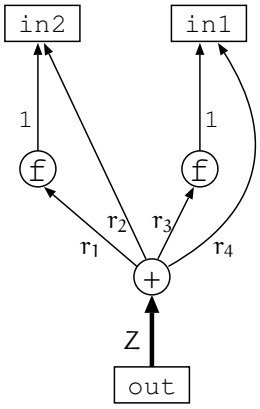
The above mentioned algorithms along with the method previously discussed provides us with a general method that implements the sufficient equivalence condition for verification of loop and data-flow transformations.

Example 37

Consider the example program pair shown in Figure 7.11 and their ADDGs shown in Figure 7.12. When the traversal hits the first addition operator



(a) ADDG of the original version after flattening



(b) ADDG of the transformed version after flattening

Figure 7.14: The ADDGs shown in Figure 7.12 after flattening.

at the root of the data-flow, the flattening operation is called resulting in the flattened ADDGs shown in Figure 7.14. The labels on the edges stand for the paths that are reduced in the process, as given below.

$$\begin{array}{ll} p_1 = (1, A, 1) & r_1 = (1) \\ p_2 = (1, A, 2) & r_2 = (2, Y, 1) \\ p_3 = (2, D, 1) & r_3 = (2, Y, 2, X, 1) \\ p_4 = (2, D, 2, B) & r_4 = (2, Y, 2, X, 2) \\ p_5 = (2, D, 2, C) & \end{array}$$

Now the matching function is called upon to establish the relation between the operands of the commutative addition operator. The following relation is returned by the matching function.

$\{(\mathtt{in1}_{(p_1)}, \mathtt{in1}_{(r_4)}), (\mathtt{f}_{(p_2)}, \mathtt{f}_{(r_1)}), (\mathtt{f}_{(p_3)}, \mathtt{f}_{(r_3)}),$
$(\mathtt{in2}_{(p_4)}, \mathtt{in2}_{(r_2)}), (\mathtt{in2}_{(p_5)}, \mathtt{in2}_{(r_2)})\}$

Note that, p_4 and p_5, both relate to r_2. They are due to the branching of the data dependencies at the `tmp2` array node in the ADDG of the original program.

7.4 Limitations of our Operator-Level Checking

The main limitations of the checking method that we have presented are with respect to its handling of the algebraic transformations. Firstly, we assume that the applied algebraic transformations do not redistribute the computation across a recurrence. This in general requires sophisticated methods of recognition of so-called scans and reductions (Blelloch 1989; Fisher and Ghuloum 1994), and, at present, it is not clear how they can be integrated into our method. Secondly, the method is limited to handling those algebraic transformations for which normalization methods are available. For instance, handling transformations based on the distributivity property requires plugging in another normalization method that will help the checker in establishing the correspondence between the data values in the two programs. Finally, the method is also limited to the transformations on the source code and does not address the transformations that are applied on the flow graphs taking the *timing* primitives into account. An example of this kind are the multi-dimensional retiming transformations (Passos and Sha 1996). In general, since the checker views all operators as uninterpreted functions, for any new sub-category of algebraic transformations involving

the properties of operators other than those for which the normal forms are already known to the checker, additional generalization and extensions are required to be incorporated. However, the mentioned limitations of the method are not of fundamental nature since the number of instances of algebraic transformations that are applied in practice are rather limited and suitable normal forms for them are known in the relevant literature.

7.5 Summary

In this chapter we have presented an operator-level equivalence checking method that overcomes the limitations of statement-level checking presented in the previous chapter and verifies any combination of loop transformations and expression propagations. The method is based on a sufficient equivalence condition over the matching data dependence slices in the two programs and is implemented by a synchronized traversal based scheme on the ADDGs of the two programs. We have shown how algebraic transformations can also be handled by relaxing the matching relation on the data dependence slices and provided a general sufficient equivalence condition for the verification of combinations of loop and data-flow transformations. For the common associative and commutative transformations, we have shown how demand-driven normalizations can be called upon to relate the matching slices. An implementation of this matching function has been presented to obtain our general equivalence checking method for loop and data-flow transformations.

Chapter 8

Features of the General Method

8.1 Introduction

In this chapter, we discuss the nice properties our general equivalence checking method exhibits. In Section 8.2, we discuss the error diagnostics generation capability of our method and how good these diagnostics are in localizing the errors in the transformed program. In Section 8.3, we discuss how the general method can be adapted to perform less work by taking advantage of the properties of the structure of the ADDG. In Section 8.4, we discuss the time complexity of our method and report some experimental results with regard to performance.

8.2 Errors and their Diagnosis

One of the motivating arguments that we provided for the use of program analysis-based approach instead of the simulation-based approach is the ability of the former to provide additional information to the designer in the event that the verification fails. This additional information, called the error diagnostics, should be able to help the designer in localizing the cause for the failure and correct it. In this section, we discuss the error diagnostics that our method is able to generate (Sec-

```
for (k = 0; k < N; k++)
    tmp[k] = in2[2*k] + in2[k];          // X
for (k = 0; k < N; k++)
    buf[2*k] = in1[2*k] + in1[k];        // Y
for (k = 0; k < N; k++)
    out[k] = tmp[k] + buf[2*k];          // Z
```

(a) Original program

```
for (k = 0; k <= 2*N-2; k += 2)
    buf[k] = in1[k] + in2[k];            // P
for (k = 1; k < N; k += 2)
    tmp[k] = in1[k] + in2[k];            // Q
for (k = 0; k < N-1; k += 2){
    out[k] = buf[k] + buf[k];            // R
    out[k+1] = tmp[k+1] + buf[2*k+2];    // S
}
```

(b) Erroneously transformed program.

Figure 8.1: An example program pair under erroneous transformations.

tion 8.2.1) and discuss their limitations (Section 8.2.2).

8.2.1 Types of Detected Errors

It is clear from the algorithms implementing our method that there are various hooks to capture a possibility for failure of equivalence proof. But as far as the program text is concerned, they correspond to three situations that we discuss in the following.

The first situation arises when the transformations have erroneously changed an operator in a statement in the original program. This corresponds to a mismatch in the operators in the data-flow and our checker is able to detect the mismatch and able to provide the precise location of the mismatching operator in the program text.

The second situation arises when the transformations have erroneously transformed the index expressions of the arrays in the original program.

This corresponds to a mismatch in the output-to-input mappings in the data-flow and our checker detects this when it reaches the input array on an affected path. It is able to then provide the information regarding the output array and its domain of elements for which the data-flow is in error.

Finally, the third situation arises when the transformations have erroneously transformed the loop bounds in the original program. This corresponds to a slice in the original program that has not been accounted for in the transformed program. In this case, the checker detects the point of departure when a certain domain is not accounted for and is able to report the array variable at which this happens and provide information on the statements assigning it and the domain of elements for which the assignment is missing.

Example 38

Consider the program pair shown in Figure 8.1. It shows an original program in Figure 8.1(a) and a version of it in Figure 8.1(b) obtained when it is erroneously transformed. The original program computes

$\forall$ `k` $\in [0 \ldots N-1]$: `out[k] = in2[2*k] + in2[k] + in1[2*k] + in1[k]`,

while the erroneously transformed program computes

$\forall$ `even k` $\in [0 \ldots N-1]$:
`out[k] = in1[k] + in2[k] + in1[k] + in2[k]`
and, $\forall$ `odd k` $\in [0 \ldots N-1]$:
`out[k] = in1[k] + in2[k] + in1[2*k] + in2[2*k]`

Clearly, the transformed program is not equivalent to the original for the even elements of the output array `out[]`, while being equivalent for the odd elements.

Now let us see the error diagnostics generated for it. We have seen the ADDG of our original program in Figure 5.17 in page 88 as an example for algebraic transformations. The ADDGs have also been shown after flattening in Figure 5.21 in page 95. The transformed program here is exactly similar to the transformed program considered there, except for the fact that the assignment to `out` has been interleaved among statements R and S for even and odd elements, respectively. Therefore, the flattened ADDG of the transformed program here duplicates the sub-ADDG for each of the two branches at `out`.

As explained earlier, the presence of non-unique successor nodes requires that the matching between the flattened ADDGs be based on the (transitive) dependency mappings. The matching between the flattened ADDG

of the original program and the left sub-ADDG of the transformed program succeeds for two paths, viz., $\{(p_2, r_2), (p_4, r_1)\}$, but fails for the other two paths, viz., $\{(p_1, r_4), (p_3, r_3)\}$. This is shown below.

$$({}^{O}{}_{p_1}\mathbf{M}_{\mathtt{out}\rightsquigarrow\mathtt{in2}} \Leftrightarrow \{[x] \rightarrow [2x]\}) \quad \not\Leftrightarrow \quad ({}^{T}{}_{r_4}\mathbf{M}_{\mathtt{out}\rightsquigarrow\mathtt{in2}} \Leftrightarrow \{[x] \rightarrow [x]\})$$
$$({}^{O}{}_{p_3}\mathbf{M}_{\mathtt{out}\rightsquigarrow\mathtt{in1}} \Leftrightarrow \{[x] \rightarrow [2x]\}) \quad \not\Leftrightarrow \quad ({}^{T}{}_{r_3}\mathbf{M}_{\mathtt{out}\rightsquigarrow\mathtt{in1}} \Leftrightarrow \{[x] \rightarrow [x]\})$$

where $x \in \{[k] \mid (\exists j \mid 2j = k \wedge 0 \leq k < \mathtt{N} \wedge k \in \mathbb{Z})\}$. The above mismatch in dependency mappings implies that paths r_4 and r_3 are in error, which correspond to statements `R` and `P` in the program text. The diagnostic points the user to these two statements, displays the index expressions of variables `out`, $\mathtt{buf}_2$, `in1` and `in2` in the statements as possible places of error and the difference in the output-input mappings. A further heuristic on this information can deduce that variable $\mathtt{buf}_2$ is common to the two paths and hence its index expression is most likely to be in error. This is indeed the case in statement `R` of the transformed program, where, it should have been `buf[2*k]`.

8.2.2 Limits to Error Localization

Barring the first situation of error detection, that of mismatching operators, the usefulness of the diagnostics generated for the other two situations depends on the nature of the program. We discuss the limitations of diagnostics generation in each of these situations below.

In the first situation there is an erroneous transformation of the index expression of an array. The checker is able to output the path that has led to the error. Here the usefulness of the diagnostic directly depends on the number of array nodes that appear on the path in question. However, it is not difficult to come up with a heuristic that continues with the traversal in spite of the error and identifies the array node that is at the root of all the failing paths. But this heuristic is useful only when there is a single erroneously transformed index expression. In the presence of multiple such errors the effectiveness of such heuristics declines.

In the second situation there is an erroneous transformation of the loop bounds. The checker outputs the array node and the domain for which the dependencies are lost during the reduction operation. The usefulness of this diagnostic depends on the number of statements the array in question is assigned in the transformed program. Again, a heuristic is to lookahead at the data-flow for which the matching has to hold and

hence determine the erroneous loop nest in question. Again, the effectiveness of this heuristic depends on the distinctness of different slices that assign the array.

8.3 Optimizations to Speed-up the Method

In this section, we discuss how the checker can be implemented in such a way as to do just enough work to show the equivalence of the programs. We discuss the optimization opportunities that are possible due to the presence of shared data-flow between different slices in a program, as is typically the case in the programs in the application domain of our interest. We first discuss how the reuse of values in a program translates to reuse of sub-proofs for the equivalence proof by way of *tabling* (Section 8.3.1). We then discuss how the traversal can be adapted in the situation of *reconvergent paths*, where the data-flow is such that there is a reuse of computation scheme but not the values (Section 8.3.2). Finally, we discuss how the user has a number of ways to help restrict the tool's work (Section 8.3.3).

8.3.1 Tabling Mechanism

During the checking process, once a pair of nodes in the two ADDGs is established to correspond for a set of values, it is tabled as a known correspondence. The next time the traversal finds itself with the same pair of nodes with a path domain that is the same or a subset of the domain for which the nodes have already been proved equivalent, the proof obligation is already known to hold, and hence, the traversal can dispose the obligation without reducing it.

8.3.2 Reconvergent Paths

Tabling is able to help in the event that there is reuse of domains of values. However, in many cases, a sub-ADDG that has been traversed for a domain is again required to be traversed for another mutually disjunct domain. This happens due to the presence of so-called re-convergent paths in the data-flow. When there are multiple points of reconvergence in the data-flow, depending on the nature of data dependencies in the

program, there can be a cascading effect resulting in repeated traversals, the number of which can possibly explode. This problem has been noted in the literature related to fault testing techniques for combinational circuits and several methods (like, for example, the D-Frontier identification) exist that address this problem.

As far as our method is concerned, note that in the outline of the equivalence checker shown in Algorithm 12, we have a call to SelectObligation(). We deliberately left it unspecified. The selection of the next primitive obligation to reduce determines which proof paths the traversal next chooses to make progress upon. Therefore, in the event that at a certain node there are multiple incoming edges in the ADDG the traversal can delay the selection of the primitive obligation and proceed with one that has only a single entry into it or one for which primitive obligations for all its incoming paths are already part of the proof obligation. In the latter case, the multiple primitive obligations are combined into one obligation.

Note that this does not however imply that all paths need be traversed only once. This is because an expression propagation transformation can result in the splitting of the reconvergent paths in the original program. Therefore, the primitive obligations may not correspond on both the ADDGs which is required in order to combine them.

8.3.3 Focused Checking

Often the transformations are applied only with respect to a certain output and/or only to a portion of the data-flow. Our method provides options for focused checking by allowing the user to restrict its work to only certain slices or sub-slices. The method is typically used, while applying transformations, in an apply-and-check mode. Another option that comes in handy here is to let the tool log the established correspondences and use it across successive equivalence checking runs during focused checking. An advantage of focused checking, more important than reducing the checking time, is that it improves the quality of the generated error diagnostics.

8.4 Performance Analysis

In this section, we provide a brief discussion of the time complexity of our method followed by a modest experimental evaluation of its prototype implementation.

8.4.1 Complexity Analysis

As we described, the method is a synchronized traversal on the two ADDGs. Our method traverses corresponding paths only once and tables all established equivalences. Therefore, if we assume that the number of maximal slices of computation in the ADDGs is very small compared to their sizes, the complexity of the traversal is linear in the size of the larger of the two ADDGs, i.e., $O(\max(|V_1| + |E_1|, |V_2| + |E_2|))$.

The condition checks as described evaluates the validity of the constraints and the best known upper bound for determining validity in Presburger arithmetic is $2^{2^{2^{pn}}}$ on the length of the formula (Oppen 1978), where $p > 1$ is some constant. The OMEGA test framework (Pugh 1992) based on Fourier-Motzkin variable elimination and a host of heuristics provides an integer programming solver for the Presburger arithmetic which is very efficient in practice. This has prompted us to use the OMEGA calculator (Kelly et al. 1996a) to perform the condition checks on our domains and mappings. The mappings that we check are taken separately for each definition-operand variable pairs and hence, the length of the formula depends solely on the size of the statement classes and in all practical cases the problem size remains reasonable. Therefore, we can assume that the time for these operations is bounded by a small constant. Hence, the overall complexity is still in the order of the traversal.

8.4.2 Experimental Analysis

We have implemented the equivalence checking procedure in a prototype tool developed in the Python programming language. The verification times that we observe on some real life examples are in the order of a few seconds. A characterization of the problem size is the nature of the data-flow in the program function. The depth and the width (number of leaf nodes) of the ADDG, determines the load on the checker. At

ADDG size (Depth, Width)	Lines of code (Ver.1, Ver.2)	Time taken (in seconds)
(4, 16)	(140, 50)	20
(4, 24)	(220, 110)	28
(4, 32)	(320, 230)	40
(8, 16)	(250, 170)	44
(8, 24)	(380, 210)	62
(8, 32)	(520, 240)	86

Table 8.1: Times required for verification (on a Pentium 4, 1.7 GHz).

present, none of the optimizations proposed in Section 8.3 have been implemented in the prototype. Still we believe that our method is scalable to large function sizes in practice. This is substantiated by the examples in Table 8.1, where we have used successively larger ADDGs constructed with realistic complexity in control-flow and variable index expressions.

8.5 Summary

In this chapter we have presented the important features of our method. Firstly it is able to provide useful error diagnostics for debugging an erroneously transformed program. Secondly, the method can be implemented in such a way that it takes, in most cases, only a single pass on the ADDGs and it provides ways for the designer to focus the checking to only parts of interest in the program. Finally, we have discussed that the method is efficient as the time it takes is, in practice, only linear in the larger of the two ADDGs. We have also experimentally validated it with some constructed examples with realistic complexity.

Chapter 9

Method in Practice

9.1 Introduction

In this chapter, we discuss how our method can be applied in practice. We take the original and the transformed program pairs from real-life application design context and discuss the scheme for checking their equivalence using our method.

The equivalence checking scheme is as shown in Figure 9.1. The input to the checker are the texts of an original program and a program obtained by applying one or more of the transformations discussed in Chapter 5 on it, called the transformed program. The checker also allows an *optional* set of inputs. These help either reduce the work for the checker by providing the focus of interest or supply it with additional information to handle algebraic transformations. The following specific options are allowed by our current implementation:

1. output variables, and optionally, their definition domains,
2. declaration of some of the intermediate variables as input variables,
3. whether statement-level or operator-level transformations have been applied, and
4. declaration of arithmetic properties of the operators in the input programs (only if algebraic transformations have been applied).

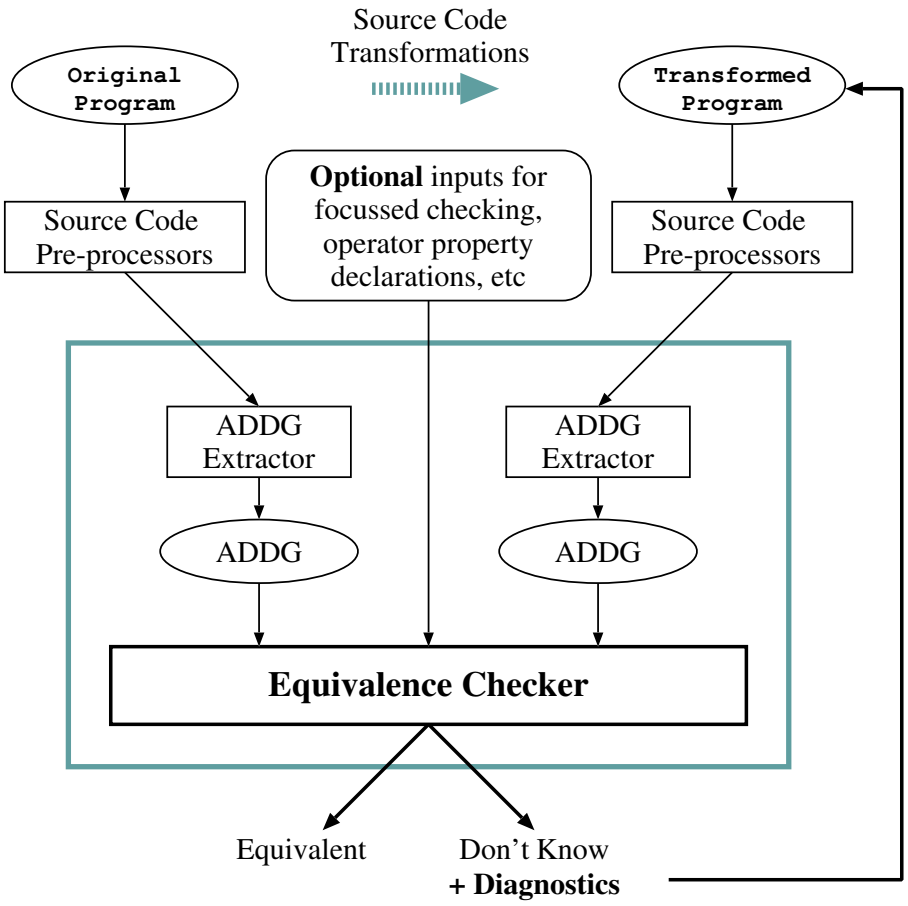


Figure 9.1: The verification and debugging scheme.

The first two options help focus the equivalence checking as described in Section 8.3.3 and the third option indicates to the checker that it can use the less expensive method that is applicable for only loop and data-reuse transformations, discussed in Section 6.3. The last one provides the additional information required by the checker when certain properties have been assumed for algebraic transformations over specific operators appearing in the programs.

We have discussed that our program equivalence checking method requires that the original and the transformed programs belong to the class of $\mathscr{D}$ programs (see Chapter 3). However, the original and the transformed programs may not belong to the class of $\mathscr{D}$ programs to start with. In such a case, source code pre-processing tools are used in order to translate them to a form that is acceptable. We discuss this further in Section 9.2. Once the two programs have been pre-processed, we use an ADDG extractor that we have implemented in order to represent them as ADDGs. Central to this extractor is `per` (De Greef 1998; De Greef 2005), a tool that provides various polyhedral domains used in constructing an ADDG as discussed in Chapter 4. Our equivalence checker takes the two ADDGs as input and applies the method we discussed in Chapter 7. The checker either terminates with a successful completion of the traversal proving the two programs to be functionally input-output equivalent or produces diagnostic information in the case of a failure (see Section 8.2).

9.2 Pre-processing the Source Code

Typically, as can be expected in practice, the original and the transformed program pairs do not fall in the $\mathscr{D}$ class of programs that we have assumed for our method, at least not in all respects. But as discussed in Chapter 3, several crucial restrictions can be relaxed by using source code pre-processing tools. They are used to pre-process the initial and the transformed programs separately, before passing them to our equivalence checker. We use four specific source code pre-processors in our tool chain and they are user-demand driven. The sequence of tools in the chain are as shown in Figure 9.2. They are – (1) selective inlining of functions in order to handle inter-procedural transformations (2) *if-conversion*, i.e., removal of any data-dependent control-flow; (3)

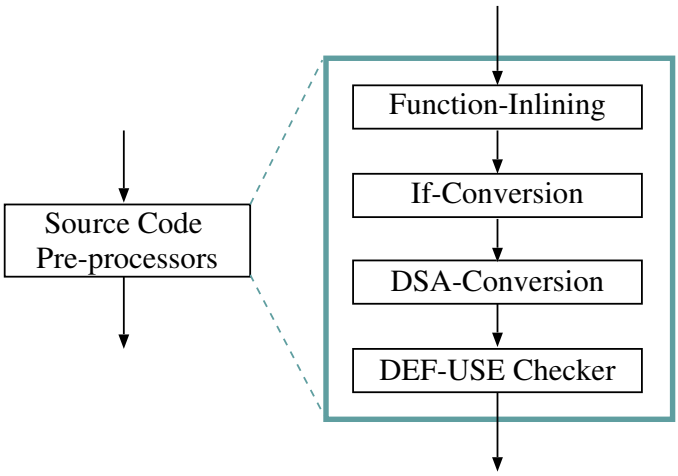


Figure 9.2: Chaining available code pre-processing tools.

conversion to *dynamic* single assignment form, i.e., removal of all false dependencies; and (4) DEF-USE checking that validates the schedule of reads and writes. Once a program is pre-processed, the true data dependencies between the variables and the operators in the program is represented as an *array data dependence graph* (ADDG). The constructed ADDGs of the two programs are input to the equivalence checker. In this section, we discuss each of the individual source code pre-processors.

9.2.1 Selective Function-Inlining

As discussed earlier, our checker functions intra-procedurally. But it is quite possible, for instance, that in the transformed program a program function has been inlined at its call-site and its internal loops exposed for transformations with other loops in the calling function. In order to handle such transformations, we use a function inlining tool (Absar et al. 2005) that selectively inlines all functions that are called by the root function that provides scope for application of global loop and data-flow transformations.

9.2.2 If-Conversion

A data-dependent control-flow arises when there are `if`-statements or `while`-loops with references to variables in their conditional expres-

sions[1]. It imposes additional dependence constraints on the execution of assignment statements in the program and hinders its analyses and transformations. This is particularly so for analyses that are primarily based on reasoning on the data-flow of the program. For such analyses, it is convenient if there is a possibility of encoding the control-flow as a data-flow. An example for such an encoding is the well known *if-conversion* (Allen et al. 1983) that removes data-dependent control dependencies in the program by converting them into data dependencies.

As discussed in the previous chapters, our equivalence checking method is based on reasoning on the data dependencies and the ADDG representation that it uses is able to capture only data dependencies. Hence, it becomes necessary that a program is free from all data-dependent control dependencies before it can be represented as an ADDG. For every assignment statement in the program within the body of the `if`-statement, if-conversion introduces into the data-flow an `if`-operator. The operator has two operands, viz., (1) a conditional expression and (2) the righthand-side expression of the assignment statement. We use an if-conversion tool that has been developed in-house in order to achieve this (Palkovic et al. 2005). Once if-conversion has been applied on a program, its representation as an ADDG is straightforward. The `if` operator is treated in the same way as any other operator.

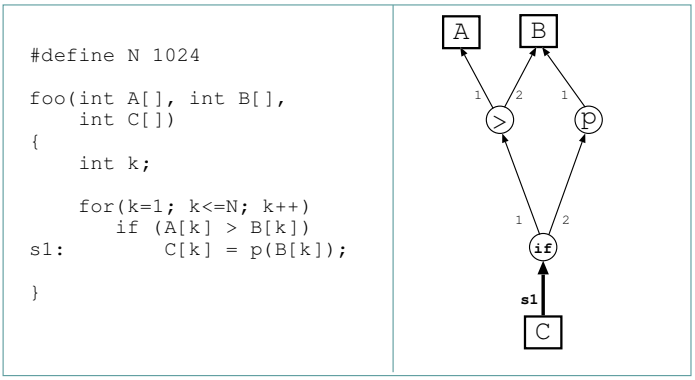


Figure 9.3: An example program function with a data-dependent `if`-statement and its ADDG obtained by if-conversion.

The examples that follow show the basic cases of program codes with

[1]Data-*independent* control-flow is based on conditional expressions on the iterators.

`if`-statements. Figure 9.3 shows a program function with a simple data-dependent `if`-statement on a single assignment statement and its ADDG representation.

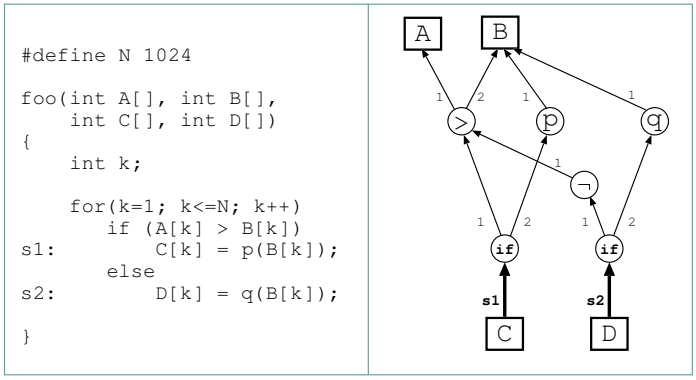


Figure 9.4: An example program function with a data-dependent `if-then-else`-statement and its ADDG obtained by if-conversion.

When the `if`-statement also has an `else`-body, the assignment of values is controlled by the negation of the predicate in the condition of the `if`-statement. The natural candidate for representing an `if-then-else`-statement, is by adding another `if`-operator for the `else`-body with a logical negation operator inserted before the condition. Figure 9.4 shows an example program in this representation.

The examples that follow in Figures 9.5-9.8 show some transformations on programs with data-dependent if-conditions and the effect they have on the ADDG representation. The equivalence checker with the knowledge of the algebraic properties of the logical operators invokes the flattening, some normalizing reductions and matching operations in identifying corresponding traversal paths.

9.2.3 DSA-Conversion

As discussed in Chapter 3 (Section 3.2.4, Page 40), we require that input programs be in dynamic single assignment (DSA) form, that is, other than iterator variables, all variables in the program are written only once during program execution. When the original and the transformed programs are not in DSA form we first apply DSA-conversion to them. This is achieved by using a prototype tool that implements a generic and

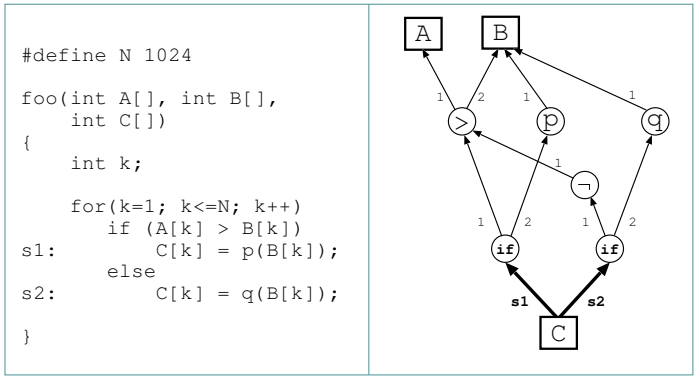


Figure 9.5: An example program function with a data-dependent `if-then-else`-statement and its ADDG obtained by if-conversion. Here, the statements `s1` and `s2` do not assign values to mutually disjunct domains of `C[]`.

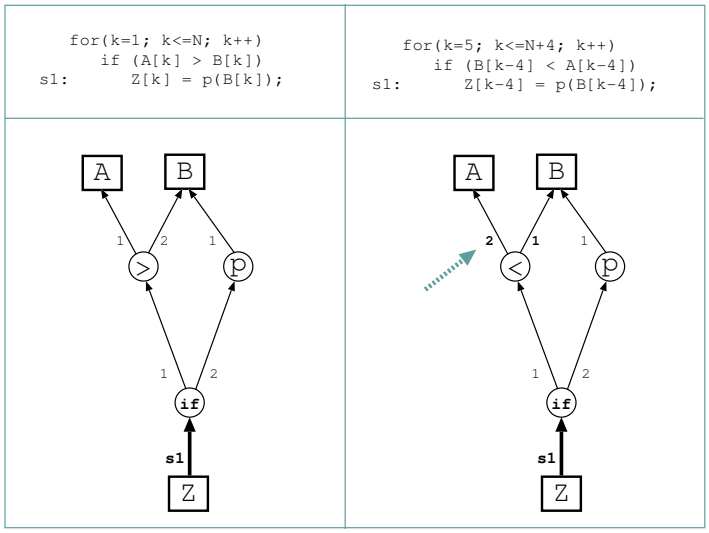


Figure 9.6: An example function pair where the relational operator $<$ in the original has been replaced with its dual in the transformed program. In the ADDGs, the position labels on the operator account for the transformation.

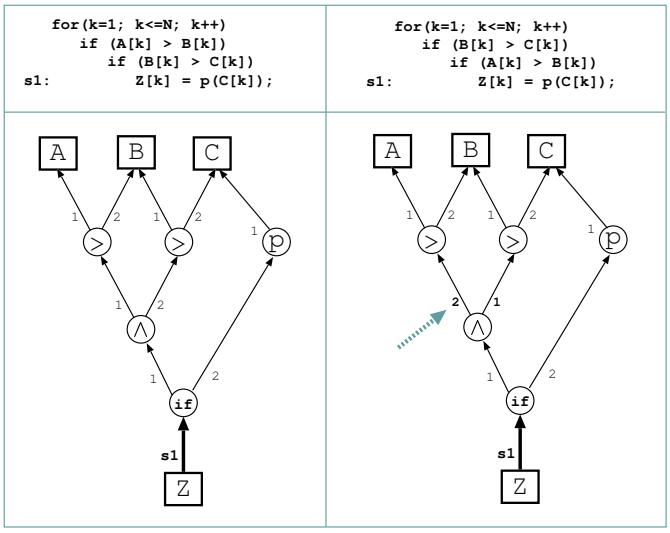


Figure 9.7: An example function pair where the if-conditions have been commuted. The $\wedge$-operator is commutative, therefore, matching-operation has to be invoked.

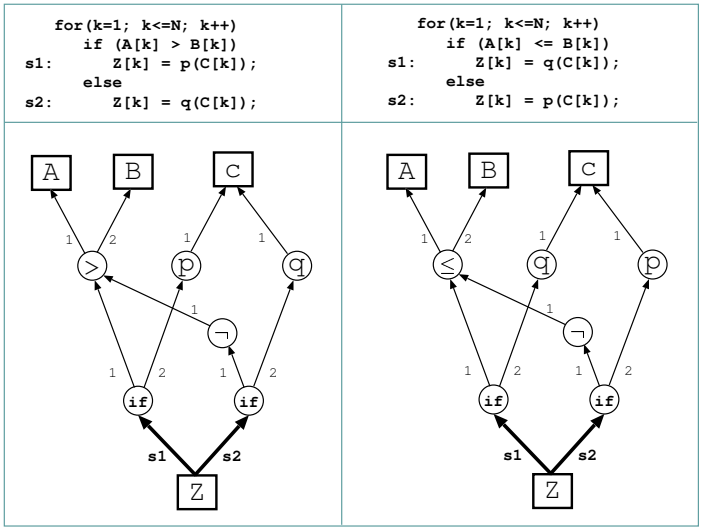


Figure 9.8: An example function pair where the bodies of the `then`-part and `else`-parts of the `if`-statement are swapped, with a corresponding replacement of the relational operator $>$ with its complementary operator $\leq$.

CPU-efficient (scalable to real-sized programs) method that is described in (Vanbroekhoven et al. 2007).

9.2.4 DEF-USE Checking

We have discussed in Chapter 3 (Section 3.2.5, Page 41) that we assume that the input programs have a valid memory access schedule, that is, all reads from a memory location occur only after writing to the location. This helps our method by easing the verification, since commutativity of statements need no longer be checked. Before invoking our equivalence checker, we validate the assumption of a valid schedule, by using an independent DEF-USE checker, that is available in a tool-suite for application of loop transformations (Verdoolaege 2005).

9.3 Case Studies

In this section, we report our experience with the current version of the equivalence checker tool on some kernels taken from actual applications in practice. In Section 9.3.1 we note the limitations of the tool, in Section 9.3.2 we compile the characteristics of the application programs that reflect their code complexity and in Section 9.3.3 we report the results of our experiments with the tool.

9.3.1 Implementation Characteristics

At present, we have implemented our formal equivalence checking methods in proof-of-concept prototype tools. Therefore, their capabilities are rather limited. Moreover, the formalization of the algorithms that we have discussed have evolved since the implementation of the prototype tool. In terms of functionality, it does not handle algebraic data-flow transformations and recurrence handling is limited. Also, it does not include the tabling mechanism and handling of the reconvergent paths. In terms of scalability, the implementation is able to handle modest size programs. Scalability to larger programs is however achievable by extending the techniques with better heuristics, for example, to handle recurrences. Future work will have to address this and this is ongoing work (Fu et al. 2006).

9.3.2 Application Characteristics

The code kernels that we have used from different applications and representative measures of their complexity is as shown in Tables 9.1 and 9.2. Note that in the case of GaussBlur and USVD, we have increased the dimensions of the array variables and created additional versions to check their impact on the time required for verification.

9.3.3 Verification Characteristics

The verification of the original and the transformed versions of the programs required times in the order of a few seconds as show in Tables 9.3 and 9.4. It also includes versions of the transformed code with errors introduced in them. As can be noted, the time required for verification does not degrade in the presence of errors. Also, as shown by verification of versions with higher dimensions of arrays, the impact on the time required for verification is negligibly small.

9.4 Summary

In this chapter, we have discussed the code pre-processing tools that are required in order to use our method in practice. With some modest experiments we have shown the feasibility of the method when applied to code kernels taken from some representative applications. However, further work is required to address some of the issues related to the implementation of our method in order to scale it to larger applications.

Legend:
v = Code version,
l = Number of lines of uncommented code,
s = Number of assignment statements,
p = Number of statement classes,
q = Range of sizes of statement classes,
i = Number of input variables,
t = Number of intermediate variables,
o = Number of output variables,
n = Number of loop nests,
d = Depth of the ADDG,
r = Dimensions of array variables.

Application	v	l	s	p	q	n	r
USVD	1	450	310	24	4-25	46	3
	2	202	67	24	1-7	16	3
	3	450	310	24	4-25	46	6
	4	202	67	24	1-7	16	6

Table 9.1: Measures of complexity in application code fragments for which we have used the statement-level equivalence checking method.

Application	v	l	s	i	t	o	n	d	r
LUD	1	23	2	1	0	1	1	4	2,3
	2	48	16	1	13	1	2	7	2,3
Durbin	1	63	8	4	0	4	7	7	2
	2	56	8	4	0	4	8	7	2
M4ME	1	56	6	4	0	2	3	3	2,4,6
	2	62	8	4	1	2	4	4	2,4,5,6
GaussBlur	1	65	8	2	5	1	5	10	2,3
	2	62	8	2	5	1	5	10	2,3
	3	62	8	2	5	1	5	10	2,3,8

Table 9.2: Measures of complexity in application code fragments for which we have used the operator-level equivalence checking method.

Legend:
o = Version of the original code fragment,
t = Version of the transformed code fragment,
Wall = Wall clock time taken to prove equivalence in seconds,
CPU = CPU clock time taken to prove equivalence in seconds.

Application	o	t	Wall	CPU
USVD	1	2	19.54	0.17
USVD erroneous	1	2	19.11	0.13
USVD	3	4	19.85	0.15

Table 9.3: Measures of verification complexity with statement-level equivalence checking method.

Application	o	t	Wall	CPU
LUD	1	2	4.99	0.27
Durbin	1	2	8.57	0.39
Durbin: erroneous 1	1	2	8.51	0.34
Durbin: erroneous 2	1	2	5.32	0.21
M4ME	1	2	12.75	0.44
M4ME: erroneous	1	2	11.83	0.41
GaussBlur	1	2	12.27	0.69
GaussBlur	1	3	12.46	0.72

Table 9.4: Measures of verification complexity with operator-level equivalence checking method.

Chapter 10

Conclusions and Future Work

This chapter concludes our dissertation. We first provide a summary of our work and its contributions (Section 10.1) and then discuss possible directions for future research on the transformation verification problem (Section 10.2).

10.1 Summary and Contributions

The transformation verification problem in computer science is a long standing one that has been faced in various contexts in hardware and software design. Depending on the particular problem context, different solutions exist in the literature. The goal of our research work is to develop a pragmatic solution for the verification problem faced by designers applying program transformations while designing signal processing applications for programmable embedded systems or scientific applications for high-performance computing. In this dissertation, we have presented a fully automatic method for the verification of loop and data-flow transformations applied on array-intensive programs that are common in these domains by functional equivalence checking of the original and the transformed programs.

We have started from the prior method for verification of loop transformations that has been developed by Samsom (1995) and prior work by Čupák (1998) on the feasibility of full system-level verification through

the integration of the loop verification method and the SFG-tracing technique developed by Claesen et al. (1991). The choice of program representation in Samsom's statement-level equivalence checking method has been such that it cannot be extended for the verification of data-flow transformations. As our first contribution, we have made two refinements to this prior method. Firstly, we have refined the constrained expressions representation of programs with the proposal for the representation of a precondition in the constrained expressions as a tuple of dependency mappings instead of a single domain. Secondly, we have relaxed the notion of a statement class to a weaker form that allows more freedom for matching statements in the original program to their corresponding statements in the transformed program. These refinements have helped us to verify any combination of loop and data-reuse transformations with a single sufficient equivalence condition defined by us.

This initial work has led us to recognize the fundamental limitations to the class of program transformations that can be verified in the restricted setting of statement-level equivalence checking. We have hence departed from it to an operator-level equivalence checking setting with the proposal for the choice of array data dependence graphs (ADDGs) as a suitable program representation for array-intensive programs. This representation is only possible for programs with certain properties that define the class of $\mathscr{D}$ programs. This class is the most commonly occurring class in design practice in signal processing and scientific computing applications and is recognized in the related literature. We have studied this representation and isolated notions of data dependence paths and slices that at once define data-flow in the program for a group of elements of the output variables. This has led us to propose a sufficient equivalence condition over matching data dependence slices in the original and transformed programs that is able to verify any combination of loop transformations and expression propagation transformations. We have developed an operator-level equivalence checking method that implements this sufficient condition through a synchronized traversal of the ADDGs of the two programs. This method is also able to deal with recurrences over computation in the programs.

The operator-level equivalence checking method proposed by us for loop and expression propagations relies on a matching relation between the operators in the two ADDGs. The operators are treated as uninterpreted functions and hence the method, by itself, cannot verify algebraic trans-

formations. This has led us to show how suitable normalization methods can be plugged into the method to establish matching relations under algebraic transformations. This is made possible by our relaxation of the notion of matching slices to a weaker form that allows more freedom to match the corresponding slices in the two programs. Relying on this, we have proposed a sufficient condition for equivalence that is able to verify combinations of loop transformations, expression propagations and algebraic transformations. We have implemented this condition by generalizing our operator-level equivalence checking method to handle the most common category of algebraic transformations, namely, those that rely on associative and commutative properties of the operators.

The method we have developed is based on the graph structure of the data-flow of the program and as such carries over the heuristic performance optimization opportunities that can be exploited by a traversal based scheme. We have identified specific opportunities that can be exploited by our method. We have shown that our method can generate useful diagnostics for error localization, even with no further extensions other than additional book keeping. By implementing our method in a prototype tool, we have demonstrated the usefulness of our verification approach on representative real-life applications. This relies on automatic program preprocessing using a tool-chain that we have helped define. The verification has been shown to be in the order of seconds, and most importantly, in a completely hands-free manner.

10.2 Directions for Future Research

The ever increasing importance for designing highly optimized systems implies an ever increasing stress on the program transformations in application design and this, in turn, implies an ever growing need for solutions to the verification problems that the transformations pose. We believe, for such transformation verification problems, functional equivalence checking approach is better suited to address the pragmatic concerns of the design activity. As far as the context of embedded systems design is concerned, the lessons from hardware synthesis are telling enough to embrace the equivalence checking approach to verification. This seems particularly important in the present thrust toward the so-called Electronic System Level (ESL) design where much of the

transformation-based design activity is on application specifications in languages like UML, MATLAB, SystemC, etc. Equivalence checking for such specifications under design transformations is an open problem. The ultimate challenge in this line of research is to develop an extensible equivalence checking framework that can support different notions of equivalence and in which checkers for different sub-classes of programs and transformations can seamlessly be plugged in. We believe that this is achievable in the foreseeable future given the progress on two fronts that matter most. Firstly, program analysis methods have significantly matured in the recent years, and secondly, there has been tremendous progress made in the development of decision procedures for relevant theories of interest. The huge commercial incentives that exist for practical solutions to the equivalence checking problems is another important factor albeit a non-technical one.

The above general campaign for future research on equivalence checking aside, there are specific avenues for future research through which the work presented in this dissertation can be improved. In general, the method that has been presented is restricted to a class of programs and transformations. It is interesting to see in what ways these classes can be relaxed to make the method even more broadly applicable in practice, not only in the target application domain of direct interest to us during the course of this dissertation work, but beyond. In particular, we believe the following immediate problems as important to pursue.

Transformations across dynamic control-flow

The method we have presented is mainly for programs with static control-flow. It can support only a minimal data dependent control-flow using the if-conversion device. We have relied on the observation that transformations are commonly applied only on the parts of a program with static control-flow. However, there now exist more and more applications for which transformations are being applied across dynamic control-flow. Therefore, it is important to see how far our method can be adapted or extended in order to deal with transformations over data dependent `while`-loops and `if`-conditions.

Algebraic transformations across recurrence

The normalizations that we have discussed for handling associative and commutative transformations assume that the recurrence has not been

involved in implementing the transformations. Our method treats recurrences as black-boxes as far as operators in them are concerned. Obvious instance specific fixes aside, a general approach for handling algebraic transformations across recurrences is important and interesting to explore.

Heuristics for better error diagnostics

The error diagnostics that our method is able to generate at present are limited to spitting out the book keeping information that it has maintained at the point of failure of equivalence. We find that it is useful enough in practice at present, mainly due to the fact that the designers are content to have found an error automatically without having to do laborious simulation that they are prepared to spend the effort required to understand whatever diagnostics output by the method. However, this can reach a threshold of designer's tolerance for large programs. Therefore, there exists much scope for developing heuristics to deduce the location for the cause of failure, relate it to the program text and present the diagnostics intelligibly. We believe this is very important to address in order for the method to gain acceptance in practice and become a part of the designer's toolbox.

Appendix A

Definitions of the Geometric Operations

Operation	Definition
Restrict Domain	$\text{RestrictDomain}(F, S) := \{x \to y \mid x \to y \in F \wedge x \in S\}$
Restrict Range	$\text{RestrictRange}(F, S) := \{x \to y \mid x \to y \in F \wedge y \in S\}$
Join	$F \bowtie G := \{x \to z \mid \exists y \text{ s.t. } x \to y \in F \wedge y \to z \in G\}$
Closure*	$F^+ := \{x \to z \mid x \to z \in F \vee \exists y \text{ s.t. } x \to y \in F \wedge y \to z \in F^+\}$

*positive transitive closure

References

Absar, M. J., P. Marchal, and F. Catthoor (2005). Data-access optimization of embedded systems through selective inlining transformation. In *3rd Workshop on Embedded Systems for Real-time Multimedia (ESTIMedia)*, pp. 75–80. IEEE. [12, 180]

Adve, S. V., D. Burger, R. Eigenmann, A. Rawsthorne, M. D. Smith, C. H. Gebotys, M. T. Kandemir, D. J. Lilja, A. N. Choudhary, J. Z. Fang, and P.-C. Yew (1997). Changing interaction of compiler and architecture. *IEEE Computer 30*(12), 51–58. [4]

Aho, A. V., R. Sethi, and J. D. Ullman (1986). *Compilers: Principles, Techniques, and Tools*. Addison Wesley. [2]

Alias, C. and D. Barthou (2005). Deciding where to call performance libraries. In *Euro-Par*, Volume 3648 of *Lecture Notes in Computer Science*, pp. 336–345. Springer. [32]

Allen, J. R., K. Kennedy, C. Porterfield, and J. D. Warren (1983). Conversion of control dependence to data dependence. In *Symposium on Principles of Programming Languages (POPL)*, pp. 177–189. ACM. [39, 181]

Allen, R. and K. Kennedy (2001). *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers. [10, 42, 43, 48, 55]

Angelo, C. M. (1994). *Formal Hardware Verification in a Silicon Compilation Environment by means of Theorem Proving*. Ph. D. thesis, Departement Elektrotechniek, Katholieke Universiteit Leuven, Belgium. [33]

Bacon, D. F., S. L. Graham, and O. J. Sharp (1994). Compiler transformations for high-performance computing. *ACM Comput-*

ing Surveys 26(4), 345–420. [4, 48]

Balasa, F., F. Catthoor, and H. De Man (1997). Practical solutions for counting scalars and dependences in ATOMIUM – a memory management system for multi-dimensional signal processing. *IEEE Transactions on Computer-aided Design CAD-16*(2), 133–145. [16]

Banerjee, U. (1988). *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers. [48]

Barrett, C. W., Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. D. Zuck (2005). TVOC: A translation validator for optimizing compilers. In *International Conference on Computer Aided Verification*, Volume 3576 of *Lecture Notes in Computer Science*, pp. 291–295. Springer. [30]

Barthou, D., P. Feautrier, and X. Redon (2001). On the equivalence of two systems of affine recurrence equations. Technical Report 4285, INRIA, France. [31]

Barthou, D., P. Feautrier, and X. Redon (2002). On the equivalence of two systems of affine recurrence equations. In B. Monien and R. Feldmann (Eds.), *8th International Euro-Par Conference*, Volume 2400 of *Lecture Notes in Computer Science (LNCS)*, pp. 309–313. Springer. [31]

Blelloch, G. E. (1989). Scans as primitive parallel operations. *IEEE Transactions on Computers 38*(11), 1526–1538. [167]

Boyle, J. M., R. D. Resler, and V. L. Winter (1999). Do you trust your compiler? *IEEE Computer 32*(5), 65–73. [7]

Brandolese, C., W. Fornaciari, F. Salice, and D. Sciuto (2002). The impact of source code transformations on software power and energy consumption. *Journal of Circuits, Systems, and Computers 11*(5), 477–502. [6]

Brockmeyer, E., M. Miranda, H. Corporaal, and F. Catthoor (2003). Layer assignment techniques for low energy in multi-layered memory organizations. In *Design, Automation and Test in Europe (DATE)*, pp. 11070–11075. IEEE Computer Society. [16]

Catthoor, F., M. Janssen, L. Nachtergaele, and H. De Man (1996). System-level data-flow transformations for power reduction in image and video processing. In *International Conference on Electronic Circuits and Systems (ICECS)*, pp. 1025–1028. IEEE. [14]

Catthoor, F., M. Janssen, L. Nachtergaele, and H. De Man (1998a). System-level data-flow transformation exploration and power-area trade-offs demonstrated on video codecs. *Journal of VLSI Signal Processing 18*(1), 39–50. [14]

Catthoor, F., S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle (1998b). *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publishers. [6, 11, 12]

Catthoor, F. and E. Brockmeyer (2000). *Unified low-power design flow for data-dominated multi-media and telecom applications*, Chapter Unified meta-flow summary for low-power data-dominated applications, pp. 7–23. Kluwer Academic Publishers. [11]

Catthoor, F. and A. Vandecappelle (2000). How to write code for high-performance, low-power multimedia applications: Course notes. Technical report, IMEC. [7]

Catthoor, F., K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. Van Achteren, and T. Omnès (2002). *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer Academic Publishers. [6, 11]

Claesen, L. J. M., F. Proesmans, E. Verlind, and H. De Man (1991). SFG-Tracing: A methodology for the automatic verification of MOS transistor level implementations from high level behavioral specifications. In P. A. Subrahmanyam (Ed.), *SIGDA International Workshop on Formal Methods in VLSI Design*. ACM. [32, 190]

Claesen, L. J. M., M. Genoe, E. Verlind, F. Proesmans, and H. De Man (1992). SFG-Tracing: A methodology of design for verifiability. In P. Prinetto and P. Camurati (Eds.), *Correct Hardware Design Methodologies*, pp. 187–202. North Holland. [32]

Collard, J.-F. (2003). *Reasoning About Program Transformations: Imperative Programming and Flow of Data*. Springer. [48]

Čupák, M. (1998). *System Level Functional Validation of Multimedia Applications*. Ph. D. thesis, Fakulta Elektrotechniky a Informatiky, Slovenská Technická Univerzita v Bratislave, Slovensko. [18, 34, 106, 189]

Čupák, M., F. Catthoor, and H. De Man (1998). Efficient functional validation of system-level loop transformations for multi-media applications. In *3rd International Workshop on High Level Design Validation and Test*, pp. 72–79. IEEE Computer Society. [34]

Čupák, M., F. Catthoor, and H. De Man (2003). Efficient system-level functional verification methodology for multimedia applications. *IEEE Design & Test of Computers 20*(2), 56–64. [33, 34, 106]

Currie, D. W., A. J. Hu, S. Rajan, and M. Fujita (2000). Automatic formal verification of DSP software. In *Design Automation Conference (DAC)*, pp. 130–135. ACM. [29]

Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13*(4), 451–490. [41]

Danckaert, K. (2001). *Loop Transformations for Data Transfer and Storage Reduction on Multiprocessor Systems*. Ph. D. thesis, Departement Elektrotechniek, Katholieke Universiteit Leuven, Belgium. [15]

De Greef, E. (1998). *Storage Size Reduction for Multimedia Applications*. Ph. D. thesis, Departement Elektrotechniek, Katholieke Universiteit Leuven, Belgium. [179]

De Greef, E. (1998-2005). `per` - polyhedral extraction routines. [179]

Feautrier, P. (1988). Array expansion. In *International Conference on Supercomputing*, pp. 429–441. ACM. [14, 41]

Feautrier, P. (1991). Dataflow analysis of array and scalar references. *International Journal of Parallel Programming 20*(1), 23–53. [39]

Fisher, A. L. and A. M. Ghuloum (1994). Parallelizing complex scans and reductions. In *SIGPLAN Conference on Programming Language*

Design and Implementation (PLDI), pp. 135–146. [167]

Franke, B. and M. O'Boyle (2003). Array recovery and high-level transformations for DSP applications. *ACM Transactions on Embedded Computing Systems 2*(2), 132–162. [12]

Fu, Q., M. Bruynooghe, G. Janssens, and F. Catthoor (2006). Requirements for constraint solvers in verification of data-intensive embedded system software. In *Proceedings of the 1st Workshop on constraints in Software Testing, Verification and Analysis*, pp. 46–57. URL: http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=42452. [185]

Genoe, M., L. J. M. Claesen, E. Verlind, F. Proesmans, and H. De Man (1991). Illustration of the SFG-tracing multi-level behavioral verification methodology, by the correctness proof of a high to low level synthesis application in cathedral-ii. In *International Conference on Computer Design: VLSI in Computer & Processors (ICCD)*, pp. 338–341. IEEE Computer Society. [32]

Genoe, M., L. J. M. Claesen, and H. De Man (1994). A parallel method for functional verification of medium and high throughput DSP synthesis. In *International Conference on Computer Design: VLSI in Computer & Processors,*, pp. 460–463. IEEE Computer Society. [32]

Goldberg, B., L. Zuck, and C. Barrett (2004). Into the loops: Practical issues in translation validation for optimizing compilers. In *International Workshop on Compiler Optimization Meets Compiler Verification*, ENTCS. Elsevier. [30]

Goos, G. and W. Zimmermann (1999). Verification of compilers. In E.-R. Olderog and B. Steffen (Eds.), *Correct System Design, Recent Insight and Advances*, Volume 1710 of *Lecture Notes in Computer Science*, pp. 201–230. Springer. [27]

Gordon, M. J. C. (1985). HOL: A machine oriented formulation of higher order logic. Technical Report 68, Computer Laboratory, University of Cambridge. [33]

Hilfinger, P. N. (1985). Silage: A high-level language and silicon compiler for digital signal processing. In *Conference on Custom Integrated Circuits*, pp. 213–216. IEEE. [33]

Hu, Y., C. Barrett, B. Goldberg, and A. Pnueli (2005). Validating more loop optimizations. In *International Workshop on Compiler Optimization Meets Compiler Verification*, Electronic Notes in Theoretical Computer Science. Elsevier. [30]

Issenin, I., E. Brockmeyer, M. Miranda, and N. Dutt (2004). Data reuse analysis technique for software-controlled memory hierarchies. In *Design, Automation and Test in Europe (DATE)*, pp. 202–207. [15]

Karp, R. M., R. E. Miller, and S. Winograd (1967). The organization of computations for uniform recurrence equations. *Journal of the ACM 14*(3), 563–590. [40]

Kelly, W., V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott (1996a). *The Omega Calculator and Library,* version 1.1.0. Department of Computer Science, University of Maryland. Available from `http://www.cs.umd.edu/projects/omega`. [175]

Kelly, W., W. Pugh, E. Rosser, and T. Shpeisman (1996b). Transitive closure of infinite graphs and its applications. *International Journal of Parallel Programming 24*(6), 579–598. [40]

Kern, C. and M. R. Greenstreet (1999). Formal verification in hardware design: a survey. *ACM Transactions Design Automation of Electronic Systems 4*(2), 123–193. [28]

Kienhuis, B., E. F. Deprettere, P. van der Wolf, and K. A. Vissers (2002). A methodology to design programmable embedded systems - the y-chart approach. In *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation (SAMOS)*, Volume 2268 of *Lecture Notes in Computer Science (LNCS)*, pp. 18–37. Springer. [5]

Kjeldsberg, P. G. (2001). *Requirement Estimation and Optimization for Data Intensive Applications*. Ph. D. thesis, Norwegian University of Science and Technology. [17]

Kulkarni, C. (2001). *Cache optimization for multimedia applications*. Ph. D. thesis, Katholieke Universiteit Leuven, Leuven, Belgium. [17]

Kurshan, R. P., V. Levin, M. Minea, D. Peled, and H. Yenigün (2002). Combining software and hardware verification techniques. *Formal Methods in System Design 21*(3), 251–280. [28]

Leupers, R. (2002). Compiler design issues for embedded processors. *IEEE Design & Test of Computers 19*(4), 51–58. [6]

Mateev, N., V. Menon, and K. Pingali (2003). Fractal symbolic analysis. *ACM Transactions on Programming Languages and Systems 25*(6), 776–813. [30]

Metzger, R. and Z. Wen (2000). *Autmatic Algorithm Recognition and Replacement: A New Approach to Program Optimization*. The MIT Press. [29]

Miranda, M., F. Catthoor, M. Janssen, and H. De Man (1998). High-level address optimisation and synthesis techniques for data-transfer intensive applications. *IEEE Transactions on VLSI Systems 6*(4), 677–686. [17]

Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers. [4]

Müller-Olm, M. (1997). *Modular Compiler Verification - A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, Volume 1283 of *Lecture Notes in Computer Science*. Springer. [27]

Necula, G. C. (2000). Translation validation for an optimizing compiler. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 83–95. ACM. [30]

Omnès, T. (2001). *Acropolis: un precompilateur de specification pour l'exploration du transfert et du stockage des donnees en conception de systemes embarques a haut debit*. Ph. D. thesis, Ecole des Mines de Paris, France. [16]

Oppen, D. C. (1978). A $2^{2^{2^{pn}}}$ upper bound on the complexity of presburger arithmetic. *Journal of Computer and System Sciences (JCSS) 16*(3), 323–332. [175]

Palkovic, M., E. Brockmeyer, H. Corporaal, F. Catthoor, and J. Vounckx (2004). Hierarchical rewriting and hiding of conditions to enable global loop transformations. In *2nd Workshop on*

Optimizations for DSP and Embedded Systems (ODES), held in conjunction with International Symposium on Code Generation and Optimization (CGO). [12]

Palkovic, M., E. Brockmeyer, P. Vanbroekhoven, H. Corporaal, and F. Catthoor (2005). Systematic preprocessing of data dependent constructs for embedded systems. In *International Workshop on Integrated Circuit and System Design, Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Lecture Notes in Computer Science (LNCS), pp. 89–98. Springer. [14, 181]

Passos, N. L. and E. H.-M. Sha (1996). Achieving full parallelism using multidimensional retiming. *IEEE Transactions on Parallel and Distributed Systems 7*(11), 1150–1163. [167]

Pnueli, A., M. Siegel, and E. Singerman (1998a). Translation validation. In B. Steffen (Ed.), *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Volume 2400 of *Lecture Notes in Computer Science (LNCS)*, pp. 151–166. Springer. [30]

Pnueli, A., O. Strichman, and M. Siegel (1998b). The Code Validation Tool (CVT): Automatic verification of a compilation process. *International Journal on Software Tools for Technology Transfer 2*(2), 192–201. [30]

Prasad, M. R., A. Biere, and A. Gupta (2005). A survey of recent advances in sat-based formal verification. *International Journal on Software Tools for Technology Transfer 7*(2), 156–173. [28]

Presburger, M. (1929). Über die vollstndigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *Comptes Rendus du I congrs de Mathmaticiens des Pays Slaves*, Warszawa, pp. 92–101. [51]

Pugh, W. (1991). Uniform techniques for loop optimization. In *International Conference on Supercomputing*, pp. 341–352. ACM. [40]

Pugh, W. (1992). A practical algorithm for exact array dependence analysis. *Communications of the ACM 35*(8), 102–114. [175]

Samet, H. (1978). Proving the correctness of heuristically optimized code. *Communications of the ACM 21*(7), 570–582. [29]

Samsom, H. (1995). *Formal Verification and Transformation of Video and Image Specifications*. Ph. D. thesis, Departement Elektrotechniek, Katholieke Universiteit Leuven, Belgium. [18, 34, 97, 98, 189]

Samsom, H., L. Claesen, and H. De Man (1993). Synguide: An environment for doing interactive correctness preserving transformations. In *Workshop on VLSI Signal Processing, VI*, pp. 269–277. IEEE. [33]

Samsom, H., F. Franssen, F. Catthoor, and H. De Man (1995). System level verification of video and image processing specifications. In *International Symposium on System Synthesis*, pp. 144–149. [34]

Séméria, L. and G. D. Micheli (1998). SpC: synthesis of pointers in C: application of pointer analysis to the behavioral synthesis from C. In *International Conference on Computer-Aided Design*, pp. 340–346. IEEE. [12]

Shashidhar, K. C., A. Vandecappelle, and F. Catthoor (2001). Low power design of turbo decoder module with exploration of energy-performance trade-offs. In *Workshop on Compilers and Operating Systems for Low Power (COLP) in conjunction with International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Barcelona, Spain, pp. 10.1–10.6. [16, 20, 210]

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2002a). Automatic a posteriori verification of certain global source code restructuring transformations. In *Program Acceleration through Application and Architecture driven Code Transformations (PA3CT)*, Edegem, Belgium. [211]

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2002b). Geometric model checking: An automatic verification technique for loop and data reuse transformations. In *1st International Workshop on Compiler Optimization Meets Compiler Verification (COCV), Electronic Notes in Theoretical Computer Science 65*(2), 71–86. [20, 210]

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2003a). Automatic functional verification of memory oriented global source code transformations. In A. J. Hu (Ed.), *8th International Workshop on High Level Design Validation and Test (HLDVT)*,

pp. 31–36. IEEE Computer Society. [20, 210]

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2003b). An automatic verification technique for loop and data reuse transformations based on geometric modeling of programs. *Journal of Universal Computer Science 9*(3), 248–269. [20, 209]

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2004a). Automatic verification of algebraic transformations. In *Program Acceleration through Application and Architecture driven Code Transformations (PA3CT)*, Edegem, Belgium. [211]

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2004b). Verification of program transformations by program equivalence checking. In *Symposium on Verification and Validation of Software Systems (VVSS)*, Eindhoven, The Netherlands. [210]

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2005a). Functional equivalence checking for verification of algebraic transformations on array-intensive source code. In *Design, Automation and Test in Europe (DATE)*, pp. 1310–1315. IEEE Computer Society. [20, 209]

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2005b). Functional equivalence checking for verification of algebraic transformations on array-intensive source code. In *ACM Sigda Ph.D. Forum at Design Automation Conference ()*, Anaheim, CA, USA. [210]

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2005c). "Look Ma, No Rules, And No Hints Neither!". Seminar on Verifying Optimizing Compilers (No. 05311), Schloss Dagstuhl, Wadern, Germany. [210]

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2005d). Verification of source code transformations by program equivalence checking. In R. Bodík (Ed.), *International Conference on Compiler Construction*, Volume 3443 of *Lecture Notes in Computer Science (LNCS)*, pp. 221–236. Springer. [18, 20, 209]

Subramanian, S. and J. V. Cook (1996). Automatic verification of object code against source code. In *Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS)*, pp. 46–

55. IEEE. [29]

Tip, F. (1995). A survey of program slicing techniques. *Journal of Programming Languages 3*(3), 121–189. [64]

Van Achteren, T. (2004). *Data Reuse Exploration Techniques for Multimedia Applications*. Ph. D. thesis, Departement Elektrotechniek, Katholieke Universiteit Leuven, Belgium. [15]

van Engelen, R. A. and K. A. Gallivan (2001). An efficient algorithm for pointer-to-array access conversion for compiling and optimizing DSP applications. In *International Workshop on Innovative Architectures for Future Generation High-Performance Processors and Systems*, pp. 80–89. IEEE. [12, 42]

van Engelen, R., D. B. Whalley, and X. Yuan (2004). Automatic validation of code-improving transformations on low-level program representations. *Science of Computer Programming 52*, 257–280. [29]

van Swaaij, M. (1992). *Data Flow Geometry: Exploiting Regularity in System-level Synthesis*. Ph. D. thesis, Departement Elektrotechniek, Katholieke Universiteit Leuven, Belgium. [15]

Vanbroekhoven, P., G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor (2003). A step towards a scalable dynamic single assignment conversion. Technical Report CW 360, Department of Computer Science, Katholieke Universiteit Leuven, Belgium. [41]

Vanbroekhoven, P., K. C. Shashidhar, M. Palkovic, G. Janssens, M. Bruynooghe, and F. Catthoor (2005a). Dynamic single assignment in action. Architectures and Compilers for Embedded Systems Symposium (ACES), Edegem, Belgium. [211]

Vanbroekhoven, P., G. Janssens, M. Bruynooghe, and F. Catthoor (2005b). Transformation to dynamic single assignment using a simple data flow analysis. In *Proceedings of The Third Asian Symposium on Programming Languages and Systems (APLAS)*, Lecture Notes in Computer Science (LNCS). Springer. [14, 41]

Vanbroekhoven, P., G. Janssens, M. Bruynooghe, and F. Catthoor (2007). A practical dynamic single assignment transformation.

ACM Trans. Design Autom. Electr. Syst. 12(4), 40:1–40:21. [185]

Vandecappelle, A., B. Bougard, K. C. Shashidhar, and F. Catthoor (2003). *Compilers and Operating Systems for Low Power*, Chapter Low-Power Design of Turbo-Decoder with Exploration of Energy-Throughput Trade-off, pp. 173–191. Kluwer Academic Publishers. [16, 20, 209]

Verdoolaege, S. (2005). *Incremental Loop Transformations and Enumeration of Parametric Sets*. Ph. D. thesis, Departement Computerwetenschappen, Katholieke Universiteit Leuven, Belgium. [15, 185]

Voeten, J. (2001). On the fundamental limitations of transformational design. *ACM Trans. Design Autom. Electr. Syst. 6*(4), 533–552. [4]

Weiser, M. (1981). Program slicing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 439–449. [64]

Wolf, W. and M. Kandemir (2003). Memory system optimization of embedded software. *Proceedings of the IEEE 91*(1), 165–182. [6]

Wolfe, M. (1996). *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company. [42, 43, 48, 55]

Wuytack, S. (1998). *System-level power optimization of data storage and transfer*. Ph. D. thesis, Departement Elektrotechniek, Katholieke Universiteit Leuven. [15, 16]

Yang, W., S. Horwitz, and T. Reps (1989). Detecting program components with equivalent behaviors. Technical Report 840, Department of Computer Science, University of Wisconsin at Madison, USA. [29]

Zuck, L., A. Pnueli, Y. Fang, and B. Goldberg (2003). VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science 9*(3), 223–247. [30]

List of Publications

In an international, refereed journal

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2003b). An automatic verification technique for loop and data reuse transformations based on geometric modeling of programs. *Journal of Universal Computer Science 9*(3), 248–269.

In a refereed book, published as a chapter

Vandecappelle, A., B. Bougard, K. C. Shashidhar, and F. Catthoor (2003). *Compilers and Operating Systems for Low Power*, L. Benini, M. Kandemir, and J. Ramanujam (Eds.), Chapter – Low-Power Design of Turbo-Decoder with Exploration of Energy-Throughput Trade-off, pp. 173–191. Kluwer Academic Publishers.

In international, refereed conferences, published in proceedings

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2005e). Verification of source code transformations by program equivalence checking. In R. Bodík (Ed.), *International Conference on Compiler Construction (CC)*, Volume 3443 of *Lecture Notes in Computer Science (LNCS)*, pp. 221–236. Springer.

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2005b). Functional equivalence checking for verification of algebraic transformations on array-intensive source code. In L. Benini and N. Wehn (Eds.), *Design, Automation and Test in Europe (DATE)*, pp. 1310–1315. IEEE Computer Society.

In international, refereed workshops, published in proceedings

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2003a). Automatic functional verification of memory oriented global source code transformations. In A. J. Hu (Ed.), *8th International Workshop on High Level Design Validation and Test (HLDVT)*, pp. 31–36. IEEE Computer Society.

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2002b). Geometric model checking: An automatic verification technique for loop and data reuse transformations. In J. Knoop and W. Zimmermann (Eds.), *1st International Workshop on Compiler Optimization Meets Compiler Verification (COCV)*, Volume 65 of *Electronic Notes in Theoretical Computer Science*. Elsevier.

Shashidhar, K. C., A. Vandecappelle, and F. Catthoor (2001). Low power design of turbo decoder module with exploration of energy-performance trade-offs. In L. Benini, M. Kandemir, and J. Ramanujam (Eds.), *2nd Workshop on Compilers and Operating Systems for Low Power (COLP) in conjunction with International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Barcelona, Spain, pp. 10.1–10.6.

In international, refereed conferences and seminars, not published or only as an abstract

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2005d). 'Look Ma, No Rules, And No Hints Neither!'. *Schloss Dagstuhl Seminar on Verifying Optimizing Compilers (No. 05311)*, Wadern, Germany.

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2005c). Functional equivalence checking for verification of algebraic transformations on array-intensive source code. *ACM Sigda Ph.D. Forum at Design Automation Conference*, Anaheim, CA, USA.

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2004b). Verification of program transformations by program equivalence checking. *Symposium on Verification and Validation of Software Systems (VVSS)*, Eindhoven, The Netherlands.

In regional, non-refereed symposiums, published in proceedings

Vanbroekhoven, P., K. C. Shashidhar, M. Palkovic, G. Janssens, M. Bruynooghe, and F. Catthoor (2005b). Dynamic single assignment in action. *Symposium on Architectures and Compilers for Embedded Systems (ACES)*, Edegem, Belgium.

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2004a). Automatic verification of algebraic transformations. *Symposium on Program Acceleration through Application and Architecture driven Code Transformations (PA3CT)*, Edegem, Belgium.

Shashidhar, K. C., M. Bruynooghe, F. Catthoor, and G. Janssens (2002a). Automatic a posteriori verification of certain global source code restructuring transformations. *Symposium on Program Acceleration through Application and Architecture driven Code Transformations (PA3CT)*, Edegem, Belgium.

Curriculum Vitæ

K. C. Shashidhar

s/o Prof. K. S. Chikkaputtaiah & Smt. U. M. Rajeevi,
No.1932, *Shubha*, 6th Cross, AIISH Colony,
Bogadi II Stage South, Mysore 570 026,
Karnataka, INDIA.
Email: kc.shashidhar@gmail.com
Webpage: http://kc.shashidhar.googlepages.com

Born on 5th February, 1976 in Mysore, India.

1993–1997 Bachelor of Engineering degree from Department of Computer Science and Engineering, Sri Jayachamarajendra College of Engineering, University of Mysore, India.

1997–1998 Software Engineer at Infosys Technologies Pvt. Ltd., Mangalore, India, and subsequently, Software Engineer at Philips Software Center Pvt. Ltd., Bangalore, India.

1998–2000 Master of Technology degree from Department of Computer Science and Engineering, Indian Institute of Technology (IIT), New Delhi, India.

2000–2001 Pre-doctoral studies at Interuniversitair Micro-Electronica Centrum (IMEC) and Katholieke Universiteit Leuven, Belgium.

2001–2005 Doctoral studies at IMEC and Departement Computerwetenschappen, Katholieke Universiteit Leuven, Belgium.

2006– Researcher in the Control Software Engineering Methods and Tools Group at India Science Laboratory, General Motors Corp., Bangalore, India.