Turchin's Supercompiler Revisited

An operational theory of positive information propagation

Master's Thesis

Revised Edition

Morten Heine Sørensen

DIKU, Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen Ø, Denmark
Electronic mail: rambo@diku.dk

January 15, 1996

Abstract

Turchin's supercompiler is a program transformer that includes both partial evaluation and deforestation. Although known in the West since 1979, the essence of its techniques, its more precise relations to other transformers, and the properties of the programs that it produces are only now becomming apparent in the Western functional programming community.

This thesis gives a new formulation of the supercompiler in familiar terms; we study the essence of it, how it achieves its effects, and its relations to related transformers; and we develop results dealing with the problems of preserving semantics, assessing the efficiency of transformed programs, and ensuring termination.

Preface

This report constitutes a Master's Thesis ("speciale") forming part of the credit towards the Master's degree ("Cand. Scient.") in Computer Science ("Datalogi") at DIKU, the Department of Computer Science at the University of Copenhagen. It reports work done between April 1993 and January 1994. My advisor was Neil D. Jones, DIKU. External examiner was Peter Sestoft, DTH.

The thesis consists of 15 chapters. Chapters 3, 5, 8, and parts of Chapter 9 are based on [Sor94b] which is joint work with Neil D. Jones and Robert Glück. I have *not* made explicit references to this effect in these chapters. The paper was written concurrently with the present thesis.

Of the remaining parts, many are synthesis of material already known in the literature. Some sections and chapters draw very directly on papers by other authors; this is mentioned explicitly in the chapters and sections in question. Section 15.1 tries to assess the original contributions of this thesis.

I have adopted the childish convention of supplying each chapter with a quotation. At least some of the quotations make a point which is directly related to the contents of the chapter in question and would have been mentioned somewhere else in the text, if not in a quotation. I hope that persons with little taste for quotations will consider this as an extenuating circumstance.

All examples, theorems, *etc.* are numbered within sections and share the same numbering sequence. This should make it easy to locate examples, *etc.* referred to in the text.

For a number of reasons the thesis is in English. As will be apparent from the text, English is not my native language. I ask the reader to forgive me for mistakes which are perhaps obvious to him.

The thesis is purely theoretical; no implementation of the algorithms described has been undertaken. See section 1.4 for further elaboration of this point.

Acknowledgements

I would like to thank Jakob Rehof with whom I share many interests in theoretical computer science, some of which we practised in a previous student project, and Fritz Henglein who was an inspiring supervisor on that project. Thanks also to Peter Harry Eidorff with whom I have spent great undergraduate days and to my girl-friend Mette Bjørnlund for being tolerant and interested in my professional problems.

The Computer Department deserves a special thanks for never making the computing environment at DIKU boring or monotonous. I would like to thank Annelise Axen and Neil D. Jones for allowing me special privileges that I was not entitled to, at a time when there were particularly severe problems. I would also like to thank Lisa Wiese and Annelise Axen for helping out with arrangements when I attended WSA '93 and a HOL course in Aarhus in december 1993, and Neil for arranging funds in those connections.

I would like to thank a number of people for quick e-mail responses to my questions concerning their work: Olivier Danvy, Carsten Kehler Holst, Bern Martens, Danny De Schreye, Stefaan Decorte, John Gallagher, Donald Smith, Phil Wadler, Aki Takano, Geoff Hamilton. Of course, any misperceptions left in the exposition are entirely my responsibility. This also holds for material discussed with people mentioned below.

I would like to thank Nils Andersen for discussions on Chin's work and on the ideas in Section 12.4. Thanks to David Sands for a bit of help on operational semantics that made me change the proof in Chapter 10.

I would like to thank Wei-Ngan Chin who patiently answered many questions concerning his work. I would also like to thank Chin for being external examiner on the student project [Sor93a] and providing many comments on that project, thereby improving the subsequent papers [Sor93a,Sor94b].

I would like to thank Robert Glück who explained many details of his own works and made the significance of details in Turchin's works so much clearer to me during his stay on DIKU. That I came to understand the significance of positive and negative information propagation is solely due to Robert. I think that Robert and Andrei Klimov have done the functional programming community a great favour by writing [Glu93a] where the essentials of *driving* are explained clearer than anywhere else in the literature. I am also grateful for the many comments to drafts of the present thesis.

I owe a special thanks to Neil D. Jones, my supervisor. It is not for me to assess the significance of the present thesis; but *if* it is worth anything, then it is more than anything due to Neil. I think that many problems in Computer Science are harder to discover than to solve; the role of a good supervisor is, I think, to provide the student with interesting problems. As such I think that Neil has done, and continues to do, a perfect job. The idea of using a grammar analysis to ensure termination of deforestation that I pursued in [Sor93a,Sor93b,Sor94a] is due to Neil. The question of the relation between deforestation and supercompilation, which in fact motivated the very formulation of the driving algorithm studied in this thesis, and many of the subsequent developments, came from Neil. Also, comments from Neil beyond a degree of detail that one could reasonably expect, improved the present thesis in many ways.

Last of all I would like to thank my parents for helping me out in many ways and for always being a great support.

Preface to the Revised Edition

In this revised edition appearing as a DIKU-report, I have made a number of corrections, most of which was discovered by the external examiner Peter Sestoft.

Apart from this a few comments also seem appropritate.

First, the ideas outlined in Chapter 7 have changed slightly. The research paper suggested in the Conclusion has been written by Robert Glück and myself, but the emphasis has shifted from the correspondence between logic programming and driving towards a correspondence between partial deduction (partial evaluation of logic programs) and driving (or supercompilation). The paper has been submitted to PLILP '94 with the title *Partial Deduction and Driving are Equivalent.*

The ideas in Section 9.2 which is also suggested as a research paper in the Conclusion still seem promising. I have for some time worked on the idea with Kristian Nielsen, DIKU, and I expect a research paper to be ready around the end of April 1994.

The problem of self-application is dealt with in very little detail, and implementation of a self-applicable positive supercompiler with an automatic technique for ensuring termination should have been mentioned as future work in Section 15.3. The significance of negative information is also dealt with in very little detail and should also have been mentioned in Section 15.3.

Acknowledgements

I am grateful for the many comments and corrections from Peter Sestoft, and the interest already shown in my thesis by researchers working on related problems.

March 10,

Morten Heine Sørensen.

Contents

Abstract **i**

Preface **iii**
Acknowledgements . . . iii

Preface to the Revised Edition **v**
Acknowledgements . . . v

1 Introduction **1**
1.1 The philosophy of the supercompiler project . . . 1
1.2 The history of the supercompiler project . . . 3
1.3 The historical relations to other transformers . . . 3
1.4 Purpose of the thesis . . . 5
1.5 Overview of the thesis . . . 5

I Positive Supercompilation **7**

2 A Simple, Lazy, First-order, Pattern-matching Language **9**
2.1 Why not Refal? . . . 9
2.2 Syntax . . . 11
2.3 Some notational conventions . . . 12
2.4 Rewrite semantics . . . 13
2.5 Correct object programs . . . 14

3 The Positive Supercompiler **17**
3.1 Driving . . . 17
3.2 Folding and postunfolding . . . 20
3.3 Generalizing . . . 23
3.4 A Burstall-Darlington explanation . . . 23
3.5 Postive supercompilation as generalized interpretation . . . 24
3.6 The essence of driving . . . 24
3.7 Overview of correctness issues . . . 26

4 Trees and Graphs as Interpretation and Transformation **27**
4.1 Interpretation trees . . . 27
4.2 Transformation trees and graphs . . . 29
4.3 Walks . . . 31
4.4 Residual programs from finite transformation graphs . . . 33
4.5 Perfect transformation trees and graphs . . . 34

II Applications **37**

5 Effects of Positive Supercompilation **39**
5.1 Elimination of intermediate data structures . . . 39
5.2 Program specialization . . . 41
5.3 Theorem proving . . . 43

6 Programming Systems and Compilers **45**
6.1 Programs=specifications . . . 46
6.2 The Futamura Projections in Partial Evaluation . . . 46
6.3 A simple intuition on the Futamura projections . . . 47
6.4 Metasystem transition . . . 48
6.5 Compiler and interpreter extraction . . . 49
6.6 Specializer extraction . . . 50
6.7 Futamura projections by metasystem transition . . . 50
6.8 Specializer projections . . . 50
6.9 Can $\mathcal{W}$ generate compiler-generators? . . . 51

7 Logic Programming by Positive Supercompilation **53**
7.1 Logic programming . . . 53
7.2 Logic programming by driving . . . 54
7.3 Transformation trees and SLD-trees . . . 55
7.4 Previous relations between supercompilation and problem solving . . . 59

8 Positive Supercompilation of Pattern Matching **61**
8.1 A test for program transformers . . . 61
8.2 A comment on measuring complexity . . . 63
8.3 Previous results on the KMP test . . . 63
8.4 Supercompilation of a pattern matching program . . . 63
8.5 Transformation graph schemes . . . 65
8.6 Theorem on complexity of specialized matchers . . . 67

9 Positive Supercompilation and Other Transformers **73**
9.1 Deforestation . . . 73
9.2 Partial Evaluation of Functional Programs . . . 80
9.3 Generalized partial computation (GPC) . . . 86
9.4 Interpretation and partial evaluation of Prolog Programs . . . 87
9.5 Relation to Turchin's supercompiler . . . 87
9.6 Conclusion . . . 89

III Correctness **91**

10 Preservation of operational semantics **93**
10.1 A discussion on equal terms . . . 93
10.2 Driving and folding preserves operational semantics . . . 95
10.3 Generalizing and postunfolding preserves semantics . . . 97
10.4 Related work . . . 98

11 On Efficiency and Size of Residual Programs **99**
11.1 Linearity of speedups . . . 99
11.2 The problem of duplicated computation . . . 101
11.3 The problem of excessive residual definitions . . . 104
11.4 The problem of duplicated code . . . 105

12 Introduction to the Problem of Ensuring Termination of $\mathcal{W}$ **107**
12.1 The canonical non-$\mathcal{W}$-termination patterns . . . 108
12.2 $\mathcal{W}$-Termination patterns . . . 110
12.3 Quasi-finiteness . . . 111
12.4 A general characterization of Non-$\mathcal{W}$-termination . . . 111
12.5 Recursive Unsolvability of $\mathcal{W}$-termination . . . 112
12.6 On the need for generalizations in principle . . . 113

13 Termination of Related Transformers **115**
13.1 Deforestation . . . 115
13.2 Partial evaluation of functional programs . . . 118
13.3 Supercompilation . . . 119
13.4 Termination of logic programs . . . 119
13.5 Transformation of logic programs . . . 121

14 Stopping the positive supercompiler **123**
14.1 A simple off-line strategy . . . 123
14.2 Why $\mathcal{W}$ terminates on the pattern matcher . . . 125
14.3 A simple on-line strategy . . . 128
14.4 On-line or off-line? . . . 130

15 Conclusion **131**
15.1 What have we achieved . . . 131
15.2 Related work . . . 132
15.3 Future work . . . 133

Bibliography **135**

Chapter 1

Introduction

It is unwise to put a hand into machinery.

Valentin F. Turchin.[1]

The supercompiler project was started in the 1960's by Turchin and his co-workers in Moscow, USSR.

Section 1.1 describes the supercompiler project and its underlying philosophy. Apart from footnotes in later chapters, the present chapter is the only place where we shall elaborate on non-technical aspects of the supercompiler project. The formulations in this section are deliberately chosen very close to formulations in Turchin's papers. Section 1.2 describes the history of the supercompiler project. Section 1.3 describes the history of partial evaluation, generalized partial computation, and deforestation, and its relation to the history of supercompilation. The motivations and applications of supercompilation and related transformers mentioned in these first three sections are treated in depth in Chapters 5, 6, 7, and 8. Section 1.4 explains the purpose of the thesis. Finally, Section 1.5 gives an overview of the thesis.

1.1 The philosophy of the supercompiler project

A *Metasystem transition* is a jump from a system S to a metasystem S', which somehow integrates a number of S-systems, modifies, and controls them as its subsystems [Tur86a, page 1]. Turchin considers metasystem transition as one of the three main instruments of creative, human thinking [Tur80c, page 651]. The three instruments are [Tur86c, page 7]:

1. Computation, or deduction.
2. Generalization, or abstraction, or induction.
3. Metasystem transition.

To solve a problem, we first try to use some standard system of rules, *deducing*. Along the way we observe the results, *generalizing* repeating patterns. If we fail to obtain a solution to our problem, we start to analyze why we failed, and for this purpose we examine the process of applying our rules; we perform a *metasystem transition* with respect to the ground-level system of rules. This could give us new, more elaborate rules to solve the problem. If we fail once more, we make another metasystem transition and analyze our means of finding new rules, *i.e.* we analyze our techniques for examining the ground-level rules. The transitions may proceed infinitely (and fruitlessly so, *e.g.* if the problem is unsolvable).

The Refal project, or supercompiler project, is an effort to implement unlimited metasystem transition, as well as the two other main instruments of thinking, on a computer [Tur86a, page 5]. There are mainly two different motivations for this endeavour.

[1] In [Tur86a].

1. The first reason is related to research in Artificial Intelligence [Tur86c, pages 1-11]. If unlimited metasystem transition is taken to be one of the main sources for creative thinking, an implementation on a computer of the concept would indeed seem an interesting approach to artificial intelligence.

   The requirement that the metasystem transtion be capable of happening an unlimited number of times has some implications on the implementation. The approach must be constructive and should not apply complicated Set Theory or Logic, because this would make it very hard for the implementation to automatically analyze itself by metasystem transition. Thus, the mathematician is allowed to observe the implementation of the concepts running, to observe the metasystem transitions, and he is allowed to reason about them, but not to build sophisticated mathematics into the implementation.[2]

   One can perceive Refal as a language suitable for implementing a system (a program) that analyzes programs, including itself, in this way, and the supercompiler as such a system that computes, generalizes, and—at the will of the computer scientist—performs metasystem transitions (through self-applications).

   We shall not be concerned with this motivation in the present thesis.

2. The second reason is related to a particular instance of repeated metasystem transition [Tur86c, pages 11-45]. In the sixties, as now, new programming languages kept appearing. This motivated the idea of a system programming environment where one could introduce new programming languages, merely by giving a *specification*. In this context Refal appears as a specification language, the specification of a new language L being an L-interpreter written in Refal.

   Moreover, one could imagine the introduction of a whole hierarchy of languages, suited for specific applications, just as one introduces hierarchies of procedures in programs. In such a setting one needs a means of turning specifications into efficient programs, preferably in the ground-level language. This task is undertaken by the supercompiler written for Refal programs, and implemented in Refal; Turchin realized independently of other researchers all three Futamura projections stated in terms of metasystem transition [Tur86c, page 258], see Chapter 6.

From these two motivations a number of other applications evolved.

1. The supercompiler performs a number of *optimizations*, for instance program specialization and elimination of intermediate data structures [Tur82], see Chapter 5.

2. The supercompiler works as a *theorem prover* [Tur80c]. For instance, to prove an equality $t = t'$ in Peano arithmetic, one formulates certain function definitions capturing the axiomatics of the equality theory, and transforms the function terms representing t and t', respectively, to identical terms, see Chapter 5.

3. The supercompiler is applied to *problem solving*. The computation of inverse functions was described in [Tur82] and developed by A. Y. Romanenko in [Rom91]. Glück applies metasystem transition to knowledge based systems and notes the similarity with Prolog interpretation [Glu92a], see Chapter 7.

4. The *generation of specializers* from weaker specializers is an application that has been studied by Glück and Jørgensen [Glu91b,Glu94a], see Chapter 6.

5. The principle of metasystem transition inspired a constructive approach to the *foundations of mathematics* [Tur87]. We shall not be concerned with this application of metasystem transition in the present thesis.

The philosophy underlying the whole project has been developed in [Tur77b], and is summarized in [Tur86a].

[2] Similarly, one should be allowed to use one's hands while building a machine, but one should not put the hands into the running machinery; hence the quotation.

1.2 The history of the supercompiler project

The technical development of the supercompiler project has been described in a number of papers in Russian and English. The technical development of *Refal* happened in the context of the idea of a system programming environment as described above. This started in the mid 1960's, and resulted in running Refal interpreters by the end of the 1960's. These early results were published in Russian.

In 1972, the transformations of Refal programs underlying the later supercompiler, the so-called *equivalent transformations*, were published in Russian. The author is only aware of one English publication containing these equivalences, *viz.* [Tur80a, Chapter 3]. The most important rule concerns *driving*: the forced instantiation of a function call for all possible value cases of the arguments followed by unfolding.

The *theory of compilation*, *i.e.* the results concerning compilation, compiler generation, and compiler generator generation by means of the supercompiler, was conceived in 1973, but not published until 1977 [Tur77a]. In 1979, the first paper in English, stating the idea and current state of the supercompiler project, appeared [Tur79]. The next year, the report [Tur80a] gave the result of the whole project: description of Refal, the equivalence transformations, the theory of compilation and metasystem transition including the Futamura projections. Also in 1980, some papers describing material in parts of [Tur80a] were published: [Tur80b] on the theory of compilation and metasystem transition, and [Tur80c] on the supercompiler as a theorem prover.

In 1982, experiments with an implementation of the supercompiler were reported by Turchin and co-workers [Tur82]

In 1986 three papers appeared: [Tur86a] describing Refal and containing a brief discussion on the philosophy underlying the project, and [Tur86b,Tur86c] describing the driving part of the supercomiler in detail.

In 1987 Turchin published the paper [Tur87] which gives a foundation of mathematics in terms of metasystem transition.

In 1988 the paper [Tur88] described the means of automatically ensuring termination of supercompilation. As far as the author knows, it is still the only paper on automatic termination of the supercompiler.

In 1989 the first implementation of a self-applicable supercompiler was reported by Glück and Turchin [Glu89]. However, only single self-application was considered, and the supercompiler needed information from the user to ensure termination. As far as the author knows, this situation has not changed since then. Glück published a paper describing metasystem transition in the context of partial evaluation [Glu91a]; the Futamura projections are special cases of metasystem transition. Glück also published a paper on the *specializer projections* concerning specialization, specializer generation, *etc.* [Glu91b]; these are also special cases of metasystem transition.

In 1990 Glück and Turchin published a paper showing that the supercompiler could automatically generate efficient pattern matchers similar to Knutt-Morris-Pratt style matchers (see Chapter 8) [Glu90]. Another application was reported in [Rom91], improving on some earlier work.

In 1992 Glück applied metasystem transition to knowledge based systems [Glu92a], and reflected on the difficulties in principle of metasystem transition and self-application [Glu92b].

Recently Glück and Klimov published a paper on the essence of driving [Glu93a], and Turchin published the paper [Tur93] which emphasizes the significance of specializing a meta-circular interpreter with respect to a program and partially known data rather than just specializing the program with respect to the partially known data. This is also emphasized in the paper by Glück and Jørgensen [Glu94a] which takes the ideas in [Glu91b] further.

The latest paper known to the author is [Glu94b] which elaborates on the philosophical significance of metasystem transition.

Yet other papers have been published by people that worked with Turchin in Moscow, S. A. Romanenko, A. Y. Romanenko, And. V .Klimov, Ark. V. Klimov, Abramov and others.

1.3 The historical relations to other transformers

The possibility of program specialization in principle is contained in Kleene's $s-m-n$ Theorem [Kle52] which states that (i) given a recursive function f of $n+m$ variables, the function $f_{d_1,\dots d_n}$ of m arguments

mapping $e_1 \dots e_m$ to $f\ d_1 \dots d_n\ e_1 \dots e_m$ is also recursive; and (*ii*) the function mapping f and $d_1 \dots d_n$ to $f_{d_1 \dots d_n}$ is recursive.

The history of *partial evaluation* goes back to work by Lombardi and Raphael in 1964 and 1967 on partial evaluation of LISP programs [Lom64,Lom67].

Futamura in Japan 1971, aware of the work of Lombardi and Raphael, discovered the possibility of compilation and compiler generation by partial evaluation [Fut71]. Ershov says: "The journal was really obscure and hardly read more than by a dozen readers."[3]

In Sweden a group around Sandewall had been working on Partial Evaluation of LISP programs, implementing a partial evaluator *REDFUN* around the mid 1970's [Bec75]. They were aware of the Lombardi-Raphael work as well as Futamura's paper. They also considered the possibility of compiler generator generation, but had not attempted self-application in practice.

As mentioned, Turchin in Moscow had formulated the transformation rules for *driving* in 1972 [Tur72,Tur74]. The main results of the theory of compilation were formulated in 1973. Due to political circumstances, Turchin was dismissed from his job from 1974 and his results were not published at this point.

Ershov in Novosibirsk, USSR, discovered around 1976 the first Futamura projection independently, and saw that a compiler could be obtained as the generating extension of an interpreter, but did not realize that this could be obtained by self-application of the specializer.

In 1976 Ershov went to Moscow, and was during a seminar approached by a co-worker of Turchin, S. A. Romanenko, attending the seminar, who told Ershow about the work in Turchin's group. All three met in a hotel room next day, where Turchin presented the idea of self-application to Ershov, and they formulated all three Futamura projections.

In Leningrad in 1977, Ershov met Sandewall. Sandewall told Ershov about Futamura's 1971 paper, and later Ershov received a copy. Ershov then coined the term *Futamura projections* for Futamura's equations, *run-through* for the Russian word ПРОГОНКА, which Turchin translated as *driving*, and *mixed computation* for what is also known as partial evaluation [Ers78].

In 1977 Turchin had to emigrate to the West. After this, the book [Tur77b] was published anonymously. This book mentions all the Futamura projections in terms of metasystem transition. The first paper published by Turchin in the West was [Tur79].

Surveys of the field of partial evaluation and supercompilation were given by Ershov [Ers82] and Futamura [Fut83].

Knowing of Ershov's papers and after meeting Ershov in Paris in 1983, Jones in Copenhagen worked towards an implementation of a self-applicable partial evaluator and succeded with his group in 1984 in self-applying the partial evaluator *mix*, named according to Ershov's terminology, to obtain both a compiler and a compiler generator [Jon85]. This project launched a number of other projects on self-applicable partial evaluation in Copenhagen. We call this line of research *the Copenhagen tradition.*

This settles the historical connections between partial evaluation and supercompilation. For more historical details, see [Ers78,Ers88,Fut83,Tur79,Jon93]. A brief survey (adopted from [Bec75]) on other work on partial evaluation in the early 1970's is contained in [Fut83]. More information on the overwhelming body of literature on partial evaluation in the 1980's and 1990's can be found in [Jon93].

We briefly describe the relation to other transformers with which we shall also be concerned.

Partial evaluation of logic programs was introduced by Komorowski in his Ph.D. thesis from 1981, see [Kom82]. Since then, partial evaluation of logic programs has been studied extensively, see [Jon93].

Generalized partial computation was described by Futamura and Nogi in 1988 [Fut88], and later developed in some detail by Takano [Tak91].

In his Ph.D. thesis Wadler in 1984 described the *listless transformer* [Wad84,Wad85] and later the *deforestation algorithm* [Wad88,Fer88]. Wadler was aware of Turchins's work from 1982 but made his discoveries independently.[4]

[3] See [Ers78].

[4] The author must apologize for erroneously stating in the student project [Sor93a] that the development of the listless transformer was inspired by Turchin's work.

1.4 Purpose of the thesis

As mentioned in Section 1.2, the supercompiler has been known in the West since 1979. Yet, the essence of its techniques and its more precise relations to other transformers are not apparent from the papers about it.

All Turchin's papers describe the supercompiler for the language Refal, which has a complicated notion of patterns. This means that algorithms for pattern matching in Refal, and in general all metaalgorithms taking Refal programs as input, are complicated to formulate and explain.

Unfortunately, it is exactly in the mechanism of instantiation and unfolding that the essence of driving lies. Explaining this in terms of Refal is complicated because Refal is complicated. Further, the author cannot recall having seen a complete formulation of the supercompiler in any paper. The problem is that the complications of Refal make it infeasible to write down the complete supercompiler on paper for humans to read.

Also, since Refal is rather different from the style of language usually studied in connection with program transformation, *e.g.* used in [Bur77], it is hard to understand what are the essentials, and it is hard to compare the supercompiler to related transformers.

The purpose of this thesis is to recast the whole theory of supercompilation in a familiar language; to formulate a supercompiler in familiar terms; to understand the essence of driving, how it achieves its effects, and its relations to related transformers; and to develop results dealing with the problems of preserving semantics, evaluating efficiency of transformed programs, and ensuring termination.

As mentioned in the preface, no implementation of the principles of the present thesis has been undertaken. The author believes that the purpose of the thesis can be served well by a purely theoretical investigation.[5] Indeed, one of the major achievements of the thesis is that the supercompiler studied is so simple that a number of theoretical results can be shown for it.

One of the natural directions for further research is to implement a self-applicable fully automatic positive supercompiler to investigate whether fully automatic self-application of supercompilation, which will be seen to be more powerful than *e.g.* partial evaluation, is feasible in practice. There are two reasons why the author has not already tried this. First, the amount of work that such a project entails exceeds what is reasonable to expect from a Master's thesis. And second, such work does not, by itself, serve the purposes for the thesis that was mentioned above.

Several implementations of the supercompiler exist. Experiments with an implementation developed at the City University of New York by Turchin and co-workers were reported in [Tur82]. Another implementation of the simple supercompiler described in [Glu93a] has been undertaken by a student of Glück.

1.5 Overview of the thesis

The paper falls in five parts. The first part consists of everything up to, and including, Chapter 1. The last part consists of the Chapter 15 and the bibliography. The remainder of the report falls in three parts, henceforth called part I, II, and III.

Part I describes the *positive supercompiler*, a reformulation of Turchin's supercompiler. Chapter 2 describes the language we shall be concerned with throughout the thesis. Chapter 3 describes the positive supercompiler. Chapter 4 describes interpretation trees and transformation trees and graphs. Interpretation trees can be considered as a call-by-name semantics more precise than that in Chapter 2. Similarly, transformation trees and graphs give a more precise formulation of the positive supercompiler, useful in many subsequent proofs.

Part II describes *applications* of the positive supercompiler and its relations to other program transformers. Chapter 5 shows three applications of the positive supercompiler, which are usually given as applications of Turchin's supercompiler. Chapter 6 shows the role of positive supercompilation in programming systems and in the automatic production of compilers. Chapter 7 shows certain connections

[5] See, however, the preface to the revised edition

between logic programming and positive supercompilation. Chapter 8 considers positive supercompilation of a naive pattern matcher in detail. Chapter 9 relates the positive supercompiler and the effects it achieves to the other transformers mentioned in Section 1.3 and to Turchin's supercompiler.

Part III is concerned with *correctness*, in particular with *termination*. Chapter 10 shows that the positive supercompiler produces programs as output that are equivalent, in a certain sense, with the input. Chapter 11 is concerned with the problem of ensuring that the output of the positive supercompiler is at least as efficient as the input. Chapter 12 contains an analysis of the problem of ensuring termination of the positive supercompiler. Chapter 13 reviews techniques for ensuring termination of related transformers, and Chapter 14 develops a simple means of ensuring termination of the positive supercompiler.

Part I

Positive Supercompilation

Chapter 2

A Simple, Lazy, First-order, Pattern-matching Language

> A language which deals with itself must be
> neither too sophisitcated nor too elementary,
> a situation reminiscent of maximizing
> the product of two factors with a given sum.
>
> *Valentin F. Turchin.*[1]

This chapter describes the language, M_1, that our metaalgorithms will be concerned with.

Section 2.1 reflects on the significance of the particular choice of language for the purposes of the present thesis. Section 2.2 introduces the syntax of M_1. Section 2.3 reviews some notational conventions that we shall employ throughout the paper. Section 2.4 describes the operational semantics of M_1 by a simple rewrite interpreter. Section 2.5 describes certain conventions about the correctness of object programs.

2.1 Why not Refal?

As described in Chapter 1, the supercompiler project has always been tied to a particular language, Refal. A brief explanation of Refal's syntax can be found in *e.g.* [Tur86c]. The semantics of Refal is defined by the *Refal Machine*, essentially a rewrite interpreter. This is also described briefly in *e.g.* [Tur86c]. A detailed exposition of both syntax and semantics appears in [Tur80a].

Refal is similar to Markov Algorithms, see *e.g.* [Men87], except that variables are allowed in patterns, and substrings can be grouped together by brackets. In more familiar programming terms, Refal is a call-by-value, first-order, purely functional, untyped programming language with a sophisticated notion of patterns. Instead of tree-structured patterns like *Cons x xs* as in ML [Mil90], Haskell [Hud92], or Miranda[2] [Tur85,Tur90], Refal deals with *strings* of atomic *symbols*. This means that patterns can collect pieces from both ends of a piece of data.

Example **2.1.1** The *reverse* function is programmed as follows in Refal:

$$\begin{array}{llcll} reverse & \{sC\,eS & = & < reverse\; eS > \; sC; & \\ & & = & & \} \end{array}$$

The name outside of curly braces shows that we are defining a function called *reverse*. The first clause then states that to reverse a string consisting of one symbol sC followed by a possibly non-empty string

[1]In [Tur79].
[2]Miranda is a trademark of Research Software Ltd.

eS, we must reverse the remaining string and put the symbol at the end; function calls are put in so-called *activation brackets* $<$ and $>$, and concatenation is denoted by juxtaposition. The second clause states that the reverse of the empty string is the empty string; the empty string is denoted by writing nothing!

An equivalent formulation is:

$$\begin{array}{lllll} reverse & \{eS\, sC & = & sC\ < reverse\ eS >; & \\ & & = & & \} \end{array}$$

□

Considering its intended use, Turchin set forth three requirements that the language for his project was to satisfy [Tur79,Tur80a,Tur86a]:

1. It must be *universal*, not only in the sense that it be possible to represent any partial recursive function, but also that it not be aimed at any special kind of programming problems or concept systems.

2. It must be *convenient* to use for programming tasks.

3. It must be *minimal*, so that the task of writing metaalgorithms with Refal programs as input does not become excessively laborious.

These requirements seem reasonable, and one can definitely argue that Refal satisfies *1* and *2*, and to some extent *3*. For our purposes there are, however, three problems with the complicated notion of pattern matching in Refal.

(*i*) It may be questioned whether the language is sufficiently minimal; metaalgorithms for expressing pattern matching, as are needed in an interpreter and in the supercompiler itself, are fairly complicated, see [Tur86b,Tur86c]. This means that it becomes complicated to reason about interpreters, supercompilers and other metaalgorithms.

(*ii*) It makes it hard to focus on the essence of supercompilation or, more specifically, the essence of driving.

(*iii*) It makes it hard to compare supercompilation with related program transformation methodologies known in the functional programming community.

In the present thesis the focus is more on (*i*)-(*iii*) than on 1 and 2 in the above requirements. Therefore we shall resort to a slightly simpler first-order language with tree-structured patterns, which will be known as M_1. The language is essentially the first-order fragment of Miranda[3] and is well-known in the functional programming community.

In M_1 we adopt call-by-name evaluation rather than call-by-value evaluation. This choice is motivated mainly by the fact that positive supercompilation, the subject of the entire thesis, basically simulates call-by-name evaluation, see Section 3.5. If we adopted a call-by-value semantics for the language, significant changes would have to be made to the positive supercompilation algorithm to ensure that its output programs were semantically equivalent to its input programs. Then the relation to deforestation, which in the present setting is extremely close, see Section 9.1, would not be so clear. Of course, the disadvantage of adopting call-by-name is that the relation to transformers simulating call-by-value, *e.g.* most partial evaluators, become less clear, see Section 9.2.

There is another reason for avoiding Refal, at least at it has been presented in the early papers, *viz.* the unusual style of syntax. In the example above, function calls are enclosed in the activation brackets $<$ and $>$. In the earlier papers, k and $\perp$ were used as opening and closing bracket, respectively. Of course, compared to what has already been mentioned, this motivation is superficial.

Previous transformers stated for M_1 include *deforestation* [Fer88], *generalized partial computation* [Tak91], and *partial evaluation* [Jon93].[4] We shall return to these transformers in Chapter 9.

[3] Hence the name M_1.

[4] Although in the case of partial evaluation a call-by-value semantics is assumed, making it harder to compare the partial evaluator to the positive supercompiler.

REMARK **2.1.2** The reader may have gotten the impression that the author has a grudge against Refal. In fact, this is not the case. Disregarding the three points (*i*)-(*iii*) above, the author has no preference for *e.g.* first-order Miranda over Refal. In this comparison it should also be kept in mind that Refal was conceived in the early 1960's where a pure functional programming language with pattern matching was something new and sophisticated. □

2.2 Syntax

We first give the syntax of M_1, then explain various points.

DEFINITION **2.2.1** (Object language M_1.) We assume denumerable, disjoint sets of variable names, constructor names, f-function names, and g-function names ranged over by v,c,f, and g, respectively. Let t range over *terms*, d over *definitions*, p over *patterns* as defined by the following grammar.

$$
\begin{array}{lcll}
t & ::= & v & \text{(variable)} \\
 & | & c\, t_1 \ldots t_n & \text{(constructor)} \\
 & | & f\, t_1 \ldots t_n & \text{(f-function call)} \\
 & | & g\, t_0\, t_1 \ldots t_n & \text{(g-function call)} \\
 & | & t_1 = t_2 \ \rightarrow\ t_3 \square t_4 & \text{(conditional)} \\
 & | & \text{let } v = t \text{ in } t' & \text{(local definition)} \\
 \\
d & ::= & f\, v_1 \ldots v_n \leftarrow t & (f\text{-function definition, no patterns}) \\
 & | & g\, p_1\, v_1 \ldots v_n \leftarrow t_1 & \\
 & & \vdots & (g\text{-function definition with patterns}) \\
 & & g\, p_m\, v_1 \ldots v_n \leftarrow t_m & \\
 \\
p & ::= & c\, v_1 \ldots v_n & \text{(patterns with one constructor)}
\end{array}
$$

The symbol $\equiv$ denotes syntactic identity on terms.

In a call $g\, t_0\, t_1 \ldots t_n$, t_0 is called the *pattern argument.* A term with no variables is called *ground.* A term with only variables and constructors is called *passive*. Non-passive terms are called *active* or *computations.* A term with an outermost constructor is call an *observable*; if the term in addition is ground, it is called a *ground observable.* A ground, passive term is called a *constant.*[5]

A term in which no variable occurs more than once is called *linear.* A program is linear if all its right hand sides are linear. A left hand side of a definition in which no variable occurs more than once is also called linear.

As usual we require that left hand sides of definitions be linear.[6] We also require that all variables in a definition's right side be present in its left side. To ensure uniqueness of reduction, we require that each function in a program have at most one definition and, in the case of a g-definition, that no two patterns p_i and p_j contain the same constructor. Finally, we adopt the hygiene convention of assuming in all contexts that two local definitions do not bind the same variable and that no function definition binds the same variable as a local definition. □

Apart from the fact that M_1 is first-order, like Refal, there are two obvious restrictions: function definitions may have *at most one pattern matching argument*, and only *non-nested patterns.* These restrictions are quite customary in the literature and are adopted simply for a concise statement of our metaalgorithms; methods exist for translating arbitrary patterns into the restricted form [Aug85,Wad87b]. In some examples we assume for simplicity that functions may be defined by several pattern matching arguments and that our metaalgorithms have been extended to handle this.

[5] Some of these notions are defined by certain grammars in later sections. The redefinitions are in all cases equivalent to the above definitions.

[6] Instead of adding a conditional with equality test, one could allow non-linear patterns. This would, however, encumber the formulation of our metaalgorithms.

The syntax above extends that of [Fer88,Tak91,Jon93] by inclusion of conditionals and local definitions. The presence of conditionals allows us to express easily a certain program for which supercompilation is particularly interesting in Chapter 8. It also allows us to discuss some specific aspects of supercompilation more concisely in Section 3.6.

Local definitions are not intended for the user; in fact, programs can always be turned into a form not using local definitions [Joh85]. Local definitions are indended as a special form to make certain terms stand out clearly for metaalgorithms. This will be much clearer in Sections 3.3 and 12.1.

We end this section with a measure on terms.

DEFINITION **2.2.2** Define for a term t the *size* of t, $|t|$, as follows.

$$
\begin{array}{lcl}
|v| & = & 1 \\
|c\, t_1 \ldots t_n| & = & 1 + \sum_{i=1}^{n} |t_i| \\
|f\, t_1 \ldots t_n| & = & 1 + \sum_{i=1}^{n} |t_i| \\
|g\, t_0 \ldots t_n| & = & 1 + \sum_{i=0}^{n} |t_i| \\
|t_1 = t_2 \rightarrow t_3 \,\square\, t_4| & = & 1 + \sum_{i=1}^{4} |t_i| \\
|\text{let } v = t_1 \text{ in } t_2| & = & 1 + \sum_{i=1}^{2} |t_i|
\end{array}
$$

□

2.3 Some notational conventions

Our metaalgorithms will be stated as functions of two arguments: a program and a term. For instance, an *interpreter* metaalgorithm will take a ground term and a program and evaluate the term using the definitions of the program. The metaalgorithms will generally contain recursive calls, but always with the original program argument. Therefore the program argument is not written explicitly; the metaalgorithm is to be understood in the context of some program.

The following abbreviations are rather tedious to get acquainted with, but they save a lot of explanations later on.

DEFINITION **2.3.1** (Notational conventions.) If a program s contains a function definition

$$f\, v_1 \ldots v_n \leftarrow t$$

then in the context of s, t^f denotes t and $v_1^f \ldots v_n^f$ denote $v_1 \ldots v_n$. Similarly, if a program s contains a function definition

$$
\begin{array}{lcl}
g\, p_1\, v_1^1 \ldots v_n^1 & \leftarrow & t_1 \\
 & \vdots & \\
g\, p_m\, v_1^m \ldots v_n^m & \leftarrow & t_m
\end{array}
$$

where $p_i = c_i\, v_{n+1}^i \ldots v_{n+k_i}^i$, then in the context of s, t^{g,c_i} denotes t_i, and $v_1^{g,c_i} \ldots v_n^{g,c_i}$ denote $v_1^i \ldots v_n^i$. Finally, $v_{n+1}^{g,c_i} \ldots v_{n+k_i}^{g,c_i}$ denote $v_{n+1}^i \ldots v_{n+k_i}^i$, and $p_1^g \ldots p_m^g$ denote $p_1 \ldots p_m$. □

EXAMPLE **2.3.2** In the context of the program:

$$
\begin{array}{lcl}
a\, Nil\, ys & \leftarrow & ys \\
a\, (Cons\, x\, xs)\, zs & \leftarrow & Cons\, x\, (a\, xs\, zs)
\end{array}
$$

$t^{a,Nil}$ denotes ys and $t^{a,Cons}$ denotes $Cons\, x\, (a\, xs\, zs)$. Further, $v_1^{a,Nil}$ denotes ys and $v_1^{a,Cons}$ denotes zs. There is no $v_2^{a,Nil}$, while $v_2^{a,Cons}, v_3^{a,Cons}$ denote x, xs. Finally, p_1^a, p_2^a denote $Nil, (Cons\, x\, xs)$. □

2.4 Rewrite semantics

We now define the semantics of M_1.

For every term two possibilities exist during evaluation. (i) Either the term has a known outermost constructor, and then interpretation proceeds to the arguments, provided that there is a function call, a local definition, or a conditional in at least one of the constructor arguments.[7] (ii) Otherwise call-by-name evaluation forces a unique call, conditional, or local definition be unfolded.

EXAMPLE **2.4.1** In the term $g\ (f\ t_1 \ldots t_n)\ t'_1 \ldots t'_m$ we are forced to unfold the call to f to be able to decide which clause of g's definition to choose. As another example, in the term $g\,(f\,t_1 = Nil\ \rightarrow\ t_2 \,\Box\, t_3)$ we are forced to unfold the call to f to be able to decide between the branches of the conditional. This, in turn, is forced by the need to decide which clause of g's definition to apply. □

In case (i) above the term is a ground observable. In case (ii) the term will be written: $t \equiv e[r]$ where r identifies the next function call, conditional, or local definition to unfold, and e is the surrounding part of the term. Traditionally, these are the *redex* and the *evaluation context*, respectively. The intention is that r is a call which is ready to be unfolded (no further evaluation of the arguments is necessary), a conditional in which a branch can be chosen (the terms in the equality test are completely evaluated), or a local definition.

We now define these notions more precisely.

DEFINITION **2.4.2** (Ground context, ground redex, ground observable, constant.) Let e range over *contexts*, r over *redexes*, o over *observables*, and b over *constants*, as defined by the grammar below.

e	$::=$	$[]$	
	$\mid$	$g\ e\ t_1 \ldots t_n$	
	$\mid$	$e' = t_2\ \rightarrow\ t_3 \,\Box\, t_4$	(First reduce left of =)
	$\mid$	$b = e'\ \rightarrow\ t_3 \,\Box\, t_4$	(Then reduce right of =)
e'	$::=$	$e \mid c\ b_1 \ldots b_{i-1}\ e'\ t_{i+1} \ldots t_n$	(Reduce left to right under constructor in test =)
r	$::=$	$f\ t_1 \ldots t_n$	
	$\mid$	$g\ o\ t_1 \ldots t_n$	(Reduce g-function if pattern-argument has outermost constructor)
	$\mid$	$b_1 = b_2\ \rightarrow\ t \,\Box\, t'$	(Reduce conditional if test arguments are constants)
	$\mid$	let $v = t$ in t'	
o	$::=$	$c\ t_1 \ldots t_n$	(Outermost constructor known)
b	$::=$	$c\ b_1 \ldots b_n$	(Complete value known)

The expression $e[t]$ denotes the result of replacing the occurrence of $[]$ in e by t. □

The following proposition states that for all t, either t decomposes uniquely into context e and redex r, or t is a ground observable. This provides the desired way of finding the next function call, conditional or local definition to unfold.

PROPOSITION **2.4.3** *(The unique decomposition property.) For all t, either there exists a unique context-redex pair e, r such that $t \equiv e[r]$, or $t \equiv o$.* □

PROOF: Easy. □

As the last preparation for the formulation of the interpreter we introduce some notation concerning substitutions.

[7] This action is based on the assumption that the user will demand that the whole term's value be printed out. In this respect the rewrite interpreter is more like the composition of the interpreter and the print loop in Miranda.

DEFINITION **2.4.4** (Substitutions.) $FV(a)$ denotes the set of *free variables*[8] of a, where a can be a term, context, or pattern. A *substitution* θ is a mapping from variables to terms. If $x_i\theta \equiv t_i$ for $i = 1 \ldots n$, and $y\theta \equiv y$ for $y \not\equiv x_i$ for all $i = 1 \ldots n$, we write θ as $\{x_i := t_i\}_{i=1}^n$. Substitutions are applied to terms in the usual pointwise manner. Application of θ to t is denoted $t\theta$. (Substitutions always bind weaker than everything else.) A *ground substitution, constant substitution, passive substitution* θ for a term t is a mapping of at least all the free variables of t to terms which are ground, constants, and passive, respectively. $t\theta$ is call a ground, constant, and passive instance of t, respectively. For an arbitrary substitution θ, $t\theta$ is called an *instance* of t. □

Now we define the interpreter $\mathcal{I}$. It follows from Proposition 2.4.3 that the clauses of $\mathcal{I}$ are exhaustive over all ground terms.

DEFINITION **2.4.5** (Rewrite Interpreter.) Let t_i range over ground terms.

$(1a)$	$\mathcal{I}[\![\, c\, t_1 \ldots t_n \,]\!]$	$\Rightarrow$	$c\, t_1 \ldots t_n$	**if** all t_i are passive
$(1b)$	$\mathcal{I}[\![\, c\, t_1 \ldots t_n \,]\!]$	$\Rightarrow$	$c\, (\mathcal{I}[\![\, t_1 \,]\!]) \ldots (\mathcal{I}[\![\, t_n \,]\!])$	**if** not all t_i are passive
(2)	$\mathcal{I}[\![\, e[f\, t_1 \ldots t_n] \,]\!]$	$\Rightarrow$	$\mathcal{I}[\![\, e[t^f\{v_i^f := t_i\}_{i=1}^n] \,]\!]$	
(3)	$\mathcal{I}[\![\, e[g\, (c\, t_{n+1} \ldots t_{n+m})\, t_1 \ldots t_n] \,]\!]$	$\Rightarrow$	$\mathcal{I}[\![\, e[t^{g,c}\{v_i^{g,c} := t_i\}_{i=1}^{n+m}] \,]\!]$	
$(4a)$	$\mathcal{I}[\![\, e[b = b' \,\rightarrow\, t \,\Box\, t'] \,]\!]$	$\Rightarrow$	$\mathcal{I}[\![\, t \,]\!]$,	**if** $b \equiv b'$
$(4b)$	$\mathcal{I}[\![\, e[b = b' \,\rightarrow\, t \,\Box\, t'] \,]\!]$	$\Rightarrow$	$\mathcal{I}[\![\, t' \,]\!]$,	**if** $b \not\equiv b'$
(5)	$\mathcal{I}[\![\, e[\text{let } v = t \text{ in } t'] \,]\!]$	$\Rightarrow$	$\mathcal{I}[\![\, e[t'\{v := t\}] \,]\!]$	

For a term t, when there is no constant b such that $\mathcal{I}[\![\, t \,]\!] \Rightarrow b$, we write $\mathcal{I}[\![\, t \,]\!] \Rightarrow \bot$. □

The operation of rewriting a term $e[f\, t_1 \ldots t_n]$ into $e[t^f\{v_i^f := t_i\}_{i=1}^n]$ conceptually proceeds in two steps: from $e[f\, t_1 \ldots t_n]$ to $e[t^f]$, and from $e[t^f]$ to $e[t^f\{t_i/v_i^f\}_{i=1}^n]$. We call the first step *unfolding* of the call to f, and the second step *binding* of the arguments in the call to f. Similarly with g-functions. In this terminology, the meaning $\mathcal{I}[\![\, t \,]\!]$ of a ground term t in some program is the result obtained by repeatedly decomposing the term into the unique context and redex and then unfolding the redex and binding its arguments. When an observable is obtained, we proceed to the constructor arguments, if any, provided that there are more conditionals, local definitions, or function calls in at least one of the constructor arguments. It should be noted that this is a *call-by-name* semantics, not a *lazy semantics.*[9]

It should also be noted that conditionals are *hyperstrict*: in $Zero = Succ\, t \,\rightarrow\, t' \,\Box\, t''$ the term t must be evaluated to a constant before evaluation proceeds with t''. This has important consequences for the design of a transformer of M_1 programs, see the end of Section 11.2 and the end of Section 5.2.

We shall sometimes consider certain sublanguages of M_1. These are introduced in the following definition.

DEFINITION **2.4.6** (Sublanguages M_0, $M_{1/2}$.) Let M_0 be the sublanguage obtained from M_1 by excluding if-expressions and let-expressions. Let $M_{1/2}$ be the language be obtained from M_1 by excluding only *let*-expressions. □

As will be emphasized frequently, the language $M_{1/2}$ is the language that we are really concerned with. The adoption of *let*-expressions is motivated in Section 3.3.

2.5 Correct object programs

We assume that the only way a program can fail to return a constant b, is through non-termination. So we assume that all functions ever called are present in the program, and with appropriate patterns in the case of g-functions, and we assume that the number of arguments in the call and in the definitions are the same. Thus, $\mathcal{I}[\![\, t \,]\!]$ is a metaalgorithm defined for all terms, which always return either a constant or $\bot$.

[8]We take the notion of free and bound cariables in a term for granted. The only way a variable can be bound is by a let-expression.

[9]We stick here to the terminology in [Bir88, Chapter 8] with which we assume the reader is familiar.

We also always implicitly assume that that whenever some t is substituted for some v then t has the appropriate form. For instance, if a term has a call $g\ v$ where g is defined by patterns on lists and we substitute a ground term t' for v, then t' must evaluate to a list, if anything.

Chapter 3

The Positive Supercompiler

A supercompiler does not transform M_1 by steps.
It runs P_1, observes (SUPERvises) its operation,
and constructs (COMPILES) a model [...] P_2.

Valentin F. Turchin.[1]

This chapter presents a program transformer, which is henceforth called *the positive supercompiler* for reasons that will become apparent in Section 3.6.

The positive supercompiler consist of four elements, divided into three phases. Phase 1, the *prephase* performs something called *generalization*. Phase 2, the *transformation phase*, performs *driving* and *folding*. And finally, phase 3, the *postphase* performs *postunfolding*.

Section 3.1 describes driving. Section 3.2 describes folding and postunfolding and gives a first example of what the positive supercompiler can do. A more systematic exposition of the effects the positive supercompiler can achieve is given in Chapters 5,6,7, and 8.

In part III the algorithm will be accompanied by techniques to ensure correctness in various respects. These techniques consist in application of generalization to parts of the object program. Section 3.3 describes generalization. In part III we then develop techniques to figure out which parts of the object program to apply generalization to.

Section 3.4 gives an explanation of positive supercompilation in terms of the Burstall-Darlington framework. Section 3.5 explains the positive supercompiler as the generalization of an interpreter. Section 3.6 explains what kind of information the positive supercompiler maintains during transformation. Section 3.7 gives an overview of the correctness issues for the positive supercompiler.

3.1 Driving

We shall express the positive supercompiler by rules for rewriting terms. It is important to realize that these terms may contain variables; otherwise we could simply evaluate the term using our interpreter. The rewrite rules can be understood intuitively as mimicking the actions of a call-by-name evaluator—but extended to continue the transformation whenever a value is not sufficiently defined at transformation time to know *exactly which* program rule should be applied. If the applicable rule is not unique, then sufficient code will be generated to account for every run-time possibility.

As was the case during interpretation in Section 2.4, for every term t two possibilities exist during transformation. (*i*) In the first case, the term is a variable or has a known outermost constructor. If it has a known outermost constructor, transformation proceeds to the arguments, provided that at least one of the arguments contain a function call, a conditional, or a local definition. If it is a variable, then transformation stops. (*ii*) In the second case, the term is not a variable and does not have a known

[1] In *e.g.* [Tur86b].

outermost constructor. Then call-by-name evaluation forces a unique call, conditional, or local definition to be unfolded. Compared to the situation where we were interpreting, we must deal with the fact that the unique call does not necessarily determine a unique clause of the function, and the unique conditional does not necessarily uniquely determine which branch to choose.

EXAMPLE **3.1.1** In the term $g_1\,(g_2\,v)\,c_1\ldots c_m$ we are forced to unfold the call to g_2 to decide which clause of g's definition to choose, but the argument is a variable so we do not know which clause to choose.

As another example, in the term $g\;(v = Nil\;\rightarrow\;t_2\;\Box\;t_3)\,t_4$ we are forced to unfold the conditional, forced by the need to decide which clause of g's definition to apply. However, a variable occurs in the test, so we do not know which branch to choose. □

For the present situation where terms contain variables, we have notions of redex and context, similar to the notions used by the interpreter in Section 2.4. Specifically, in case (i) above the term is an *observable*, and in case (ii) the term will be written: $t \equiv e[r]$ where r, the *redex*, identifies the next function call, conditional, or local definition to unfold, and e, the *context*, is the surrounding part of the term. The intention is, once again, that r is a call which is ready to be unfolded (no further evaluation of the arguments is necessary, but they may contain variables), a conditional in which a branch can be chosen (the terms in the equality test are completely evaluated, but they may contain variables), or a local definition.

We now define these notions more precisely. The only difference from the similar definition is Section 2.4 is that the categories o, b may contain variables.

DEFINITION **3.1.2** (Evaluation context, redex, observable, constant.) Let e range over *evaluation contexts*, r over *redexes*, o over *observables*, and b over *passive* terms, as defined by the grammar below.

$$
\begin{array}{lcl}
e & ::= & [] \\
 & \mid & g\,e\,t_1\ldots t_n \\
 & \mid & e' = t_2\;\rightarrow\;t_3\;\Box\;t_4 \\
 & \mid & b = e'\;\rightarrow\;t_3\;\Box\;t_4 \\
e' & ::= & e \mid c\,b_1\ldots b_{i-1}\,e'\,t_{i+1}\ldots t_n \\
 & & \\
r & ::= & f\,t_1\ldots t_n \\
 & \mid & g\,o\,t_1\ldots t_n \\
 & \mid & b_1 = b_2\;\rightarrow\;t\;\Box\;t' \\
 & \mid & \text{let } v = t \text{ in } t' \\
 & & \\
o & ::= & c\,t_1\ldots t_n \mid v \\
b & ::= & c\,b_1\ldots b_n \mid v
\end{array}
$$

The expression $e[t]$ again denotes the result of replacing the occurrence of [] in e by t. □

The following Proposition states that the Unique Decomposition Property remains true for the new notions of context, *etc.* providing the desired way of finding the next function call, conditional or local definition to unfold.

PROPOSITION **3.1.3** *(The unique decomposition property.) For all t, either there exists a unique context-redex pair e, r such that $t \equiv e[r]$, or $t \equiv o$.* □

PROOF: Easy. □

DEFINITION **3.1.4** (Unifiers.) For two terms t, t' a *unifier* is a substitution σ such that $t\sigma \equiv t'\sigma$. A *most general* unifier σ for t, t' is a unifier σ such that if σ' is also a unifier for t, t', then there exists a θ such that for all t, $t\sigma' \equiv (t\sigma)\theta$. An *idempotent* substitution is a θ such that for all t, $(t\theta)\theta \equiv t\theta$. An injective substitution mapping all variables to variables is called a *renaming*. □

PROPOSITION **3.1.5** (*i*) *if there exists a unifier for* t, t', *then there exists a most general unifier for* t, t'. (*ii*) *if* σ *and* σ' *are two most general unifiers, then there exists a renaming* θ *such that for all* t, $t\sigma \equiv (t\sigma')\theta$. (*iii*) *if there exists a most general unifier, then there exists an idempotent, most general unifier.* □

PROOF: (*i*) and (*ii*) are well-known, see *e.g.* [Ede85]. (*iii*) is less frequently cited, but see [Ede85, Remark 4.2]. □

DEFINITION **3.1.6** For two terms t, t', $MGU(t, t')$ denotes an idempotent, most general unifier if one exists, and $fail$ otherwise. It is convenient to define $t\,fail \equiv t$, *i.e.* $fail$ works as the identity substitution. □

We can now define the driving part of the positive supercompiler; the significance of "positive" will be evident in Section 3.6. The algorithm should be understood in the context of a given object program p. By the unique decomposition property, the algorithm below is exhaustive over all terms. The algorithm is followed by a number of explanations.

DEFINITION **3.1.7** (Driving.)

(0) $\mathcal{W}[\![\, v \,]\!] \Rightarrow v$

(1*a*) $\mathcal{W}[\![\, c\, t_1 \ldots t_n \,]\!] \Rightarrow c\, t_1 \ldots t_n$
if all t_i are passive

(1*b*) $\mathcal{W}[\![\, c\, t_1 \ldots t_n \,]\!] \Rightarrow c\, (\mathcal{W}[\![\, t_1 \,]\!]) \ldots (\mathcal{W}[\![\, t_n \,]\!])$
if not all t_i are passive

(2) $\mathcal{W}[\![\, e[f\, t_1 \ldots t_n] \,]\!] \Rightarrow f^{\square}\, u_1 \ldots u_k$
where
$f^{\square}\, u_1 \ldots u_k \leftarrow \mathcal{W}[\![\, e[t^f\{v_i^f := t_i\}_{i=1}^n] \,]\!]$

(3*a*) $\mathcal{W}[\![\, e[g\, (c\, t_{n+1} \ldots t_{n+m})\, t_1 \ldots t_n] \,]\!] \Rightarrow f^{\square}\, u_1 \ldots u_k$
where
$f^{\square}\, u_1 \ldots u_k \leftarrow \mathcal{W}[\![\, e[t^{g,c}\{v_i^{g,c} := t_i\}_{i=1}^{n+m}] \,]\!]$

(3*b*) $\mathcal{W}[\![\, e[g\, v\, t_1 \ldots t_n] \,]\!] \Rightarrow g^{\square}\, v\, u_1 \ldots u_k$
where
$g^{\square}\, p_1\, u_1 \ldots u_k \leftarrow \mathcal{W}[\![\, e[t^{g,c_1}\{v_i^{c_1} := t_i\}_{i=1}^n]\{v := p_1\} \,]\!]$
$\vdots$
$g^{\square}\, p_m\, u_1 \ldots u_k \leftarrow \mathcal{W}[\![\, e[t^{g,c_m}\{v_i^{c_m} := t_i\}_{i=1}^n]\{v := p_m\} \,]\!]$

(4*a*) $\mathcal{W}[\![\, e[b = b' \;\rightarrow\; t \,\square\, t'] \,]\!] \Rightarrow \mathcal{W}[\![\, e[t] \,]\!]$
if b, b' are ground and $b \equiv b'$

(4*b*) $\mathcal{W}[\![\, e[b = b' \;\rightarrow\; t \,\square\, t'] \,]\!] \Rightarrow \mathcal{W}[\![\, e[t'] \,]\!]$
if b, b' are ground and $b \not\equiv b'$

(4*c*) $\mathcal{W}[\![\, e[b = b' \;\rightarrow\; t \,\square\, t'] \,]\!] \Rightarrow b = b' \;\rightarrow\; \mathcal{W}[\![\, (e[t])MGU(b, b') \,]\!] \,\square\, \mathcal{W}[\![\, e[t'] \,]\!]$
if not both b, b' are ground

(5) $\mathcal{W}[\![\, e[\text{let } v = t \text{ in } t'] \,]\!] \Rightarrow \text{let } v = \mathcal{W}[\![\, t \,]\!] \text{ in } \mathcal{W}[\![\, e[t'] \,]\!]$

□

The notation is similar to that employed for the interpreter in Section 2.4; it is explained in Section 2.3. Note that in clause (4c), the substitution applies to the entire term $e[t]$.

Recall that the symbol $\Rightarrow$ denotes evaluation in the metalanguage, *i.e.* transformation. For instance, in clause (2) the result of transforming $e[f\, t_1 \ldots t_n]$ is a call to a new function $f^{\Box}$. This function is then defined with right hand side the result of transforming $t^f\{v_i^f := t_i\}_{i=1}^n$. The symbol "$\leftarrow$" refers to a definition in the object language (the language of definition 2.2.1.)

The patterns p_i in clause (3b) must be chosen as fresh renamings of the corresponding patterns p_i^g, *i.e.* they must be chosen so that $FV(p_i) \cap FV(e[g\; v\; t_1 \ldots t_n]) = \emptyset$. The terms t^{c_i} must be chosen as corresponding renamings. That is, let θ be such that $p_i^g\theta \equiv p_i$, then t^{c_i} must be such that $t^{g,c_i}\theta \equiv t^{c_i}$.

The **where** should be read as a code generation command. As mentioned, a term $e[r]$ is transformed into a call to a new function $f^{\Box}$; these new functions are collected somehow in a new program.[2] The free variables $u_1 \ldots u_k$ in these calls are simply all the variables of $e[r]$. In clause (3b) the variable v must come first and should not be included in $u_1 \ldots u_k$.

In clause (1b), (3b), (4c), and (5) we say that each of the recursive calls to $\mathcal{W}$ determine a *branch* of transformation.

This algorithm is very similar to the formulation of the deforestation algorithm in [Fer88], as will be investigated in more depth in part II. The present algorithm was "discovered" as the weakest extension of deforestation strong enough to yield efficient pattern matchers when applied to a general matcher and fixed pattern, see Chapter 8.

3.2 Folding and postunfolding

The following definition will be convenient.

DEFINITION **3.2.1** Two terms t, t' are *identical MVR* (modulo variable renaming) if there exists a renaming θ so that $t \equiv t'\theta$. In that case, we write $t \equiv t'$ (MVR). $\Box$

EXAMPLE **3.2.2** Consider the following term and program:

$$
\begin{array}{lcl}
 & & a\;(a\;zs\;ws)\;ts \\
a\;Nil\;ys & = & ys \\
a\;(Cons\;x\;xs)\;ys & = & Cons\;x\;(a\;xs\;ys)
\end{array}
$$

Here a is the append function, and the term is double append; it appends three lists. The inefficiency is apparent: the cons-cells of zs will be deallocated by the inner append which will allocate new cons-cells; these cells are deallocated by the outer append, which in turn allocates new ones.

Here is what happens when we run $\mathcal{W}$ on the term and program above:

$$
\begin{array}{lcl}
\mathcal{W}[\![\, a\;(a\;zs\;ws)\;ts \,]\!] & \Rightarrow & g_1\;zs\;ws\;ts \\
\quad \textbf{where} & & \\
\quad g_1\;Nil\;ws\;ts & \leftarrow & \mathcal{W}[\![\, a\;ws\;ts \,]\!] \\
 & \Rightarrow & g_2\;ws\;ts \\
\qquad \textbf{where} & & \\
\qquad g_2\;Nil\;ts & \leftarrow & \mathcal{W}[\![\, ts \,]\!] \\
 & \Rightarrow & ts \\
\qquad g_2\;(Cons\;x\;xs)\;ts & \leftarrow & \mathcal{W}[\![\, Cons\;x\;(a\;xs\;ts) \,]\!] \\
 & \Rightarrow & Cons\;\mathcal{W}[\![\, x \,]\!]\;\mathcal{W}[\![\, a\;xs\;ts \,]\!] \\
 & \Rightarrow & \ldots \\
\quad g_1\;(Cons\;x\;xs)\;ws\;ts & \leftarrow & \mathcal{W}[\![\, a\;(Cons\;x\;(a\;xs\;ws))\;ts \,]\!] \\
 & \Rightarrow & f\;x\;xs\;ws\;ts \\
\qquad \textbf{where} & & \\
\qquad f\;x\;xs\;ws\;ts & \leftarrow & \mathcal{W}[\![\, Cons\;x\;(a\;(a\;xs\;ws)\;ts) \,]\!] \\
 & \Rightarrow & Cons\;\mathcal{W}[\![\, x \,]\!]\;\mathcal{W}[\![\, a\;(a\;xs\;ws)\;ts \,]\!] \\
 & \Rightarrow & \ldots
\end{array}
$$

[2] We take the liberty of being imprecise on this point.

The transformation of the original term, $a\,(a\,zs\,ws)\,ts$, gives rise to a function g_1, which gives rise to new terms to be transformed, *etc.* In the last step above, the transformation has gone from the original term to the same term, $a\,(a\,xs\,ws)\,ts$ (modulo variable renaming.) This will happen over and over again; the transformation process proceeds infinitely. Also, the term $a\,xs\,ts$ will be encountered over and over again.

As the example perhaps suggests, this will happen quite often when we are transforming recursive functions. □

It is unsatisfactory that the algorithm does not terminate on a wide class of programs. A partial solution is to keep a record of all the terms the algorithm encounters. Before each transformation step of a term, say t, the algorithm checks whether t has previously been encountered. If so, the algorithm should let the result of transforming t be the same as the result obtained the first time t was encountered; and otherwise the algorithm should proceed as usual.

It will turn out in chapter 12 that we need not record *all* terms. We shall record only terms that give rise to new definitions, *i.e.* terms with a function call in the redex. Also, when transforming the arguments of a constructor term, $c\,t_1 \ldots t_n$, the transformation of each term t_i will not be informed about the terms that have been encountered during transformation of the other t_j's. Similarly with each branch in clause (3b). This makes the result of transformation independent of the order in which constructor arguments and clauses of g-definitions are transformed.

The folding scheme is an important part of the positive supercompiler and so deserves to be stated in a definition of its own. The definition is followed by an example of the effect of folding.

DEFINITION **3.2.3** (Folding scheme.) We assume that the residual calls $f^{\square}\ u_1 \ldots u_k$ in clauses (2) and (3a) and the residual calls $g\ v\ u_1 \ldots u_k$ in clause (3b) are uniquely determined by the term which is argument to $\mathcal{W}$, and we assume that for two terms which are identical MVR, the corresponding residual calls are identical MVR too.

We assume that $\mathcal{W}$ somehow makes sure that if a some term $e[r]$ giving rise to a new definition has been encountered, then upon encountering the same term MVR a second time in the same branch of transformation, that branch of transformation generates the residual call and then terminates. No new definition is introduced the second time. □

REMARK **3.2.4** There are a number of ways that one could make this mechanism of folding back explicit. These mechanisms vary in comprehensiblity and efficiency. A variety of proposals have been made in the literature for various transformers. We shall return briefly to the topic in part III. In the present context we prefer to avoid introducing any extra syntax into the algorithm. □

Let us see how this works on the example.[3]

[3]In the below example we are using **where** in two distinct senses. First, as a code generation construct, as explained earlier; and second, as a shorthand notation showing the result when $\mathcal{W}$ goes to the arguments of a term with an outermost constructor (the 3rd and 5th **where** below.

EXAMPLE **3.2.5**

$$
\begin{array}{lll}
\mathcal{W}[\![\, a\ (a\ zs\ ws)\ ts \,]\!] & \Rightarrow & g_1\ zs\ ws\ ts \\
\quad \textbf{where} & & \\
\quad g_1\ Nil\ ws\ ts & \leftarrow & \mathcal{W}[\![\, a\ ws\ ts \,]\!] \\
 & \Rightarrow & g_2\ ws\ ts \\
\qquad \textbf{where} & & \\
\qquad g_2\ Nil\ ts & \leftarrow & \mathcal{W}[\![\, ts \,]\!] \\
 & \Rightarrow & ts \\
\qquad g_2\ (Cons\ x\ xs)\ ts & \leftarrow & \mathcal{W}[\![\, Cons\ x\ (a\ xs\ ts) \,]\!] \\
 & \Rightarrow & Cons\ \mathcal{W}[\![\, x \,]\!]\ \mathcal{W}[\![\, a\ xs\ ts \,]\!] \\
\qquad\quad \textbf{where} & & \\
\qquad\quad \mathcal{W}[\![\, x \,]\!] & \Rightarrow & x \\
\qquad\quad \mathcal{W}[\![\, a\ xs\ ts \,]\!] & \Rightarrow & g_2\ xs\ ts \\
\quad g_1\ (Cons\ x\ xs)\ ws\ ts & \leftarrow & \mathcal{W}[\![\, a\ (Cons\ x\ (a\ xs\ ws))\ ts \,]\!] \\
 & \Rightarrow & f\ x\ xs\ ws\ ts \\
\qquad \textbf{where} & & \\
\qquad f\ x\ xs\ ws\ ts & \leftarrow & \mathcal{W}[\![\, Cons\ x\ (a\ (a\ xs\ ws)\ ts) \,]\!] \\
 & \Rightarrow & Cons\ \mathcal{W}[\![\, x \,]\!]\ \mathcal{W}[\![\, a\ (a\ xs\ ws)\ ts \,]\!] \\
\qquad\quad \textbf{where} & & \\
\qquad\quad \mathcal{W}[\![\, x \,]\!] & \Rightarrow & x \\
\qquad\quad \mathcal{W}[\![\, a\ (a\ xs\ ws)\ ts \,]\!] & \Rightarrow & g_1\ xs\ ws\ ts
\end{array}
$$

Note that the second time $\mathcal{W}$ encountered the term $a\ (a\ xs\ ws)\ ts$, the branch of transformation was terminated, and similarly with $a\ ws\ ts$. Such easy cases of looping back do not always occur; we return to this in part III.

The final term and program is:

$$
\begin{array}{lll}
 & & g_1\ zs\ ws\ ts \\
g_1\ Nil\ ws\ ts & \leftarrow & g_2\ ws\ ts \\
g_1\ (Cons\ x\ xs)\ ws\ ts & \leftarrow & f\ x\ xs\ ws\ ts \\
f\ x\ xs\ ws\ ts & \leftarrow & Cons\ x\ (g_1\ xs\ ws\ ts) \\
g_2\ Nil\ ts & \leftarrow & ts \\
g_2\ (Cons\ x\ xs)\ ts & \leftarrow & Cons\ x\ (g_2\ xs\ ts)
\end{array}
$$

This is almost perfect; g_2 is the append function, and g_1 is double append with the allocation and deallocation eliminated. There is only one annoying thing: the intermediate f-function f. The term that this function records (the body of f) was never encountered again, so it was not necessary to introduce the call. We would rather have wanted the residual program to be the following, which is obtained by unfolding the calls to f:

$$
\begin{array}{lll}
 & & g_1\ zs\ ws\ ts \\
g_1\ Nil\ ws\ ts & \leftarrow & g_2\ ws\ ts \\
g_1\ (Cons\ x\ xs)\ ws\ ts & \leftarrow & Cons\ x\ (g_1\ xs\ ws\ ts) \\
g_2\ Nil\ ts & \leftarrow & ts \\
g_2\ (Cons\ x\ xs)\ ts & \leftarrow & Cons\ x\ (g_2\ xs\ ts)
\end{array}
$$

But we could not know that f was not needed, until we were done. □

To compensate for such redundant intermediate f-functions we perform after application of $\mathcal{W}$ a *postunfolding phase*, as described in the following definition.

DEFINITION **3.2.6** (Postunfolding) Given residual term and program t, p. Replace every term $s \equiv e[f\ t_1 \ldots t_n]$ with $e[t^f\{v_i^f := t_i\}_{i=1}^n]$, where s is either t or a right hand side of p and where the total number of calls to f in the residual term and program is 1. Unless stated otherwise, $\mathcal{W}[\![\, t \,]\!]$ henceforth denotes the result of applying the transformation phase (driving and folding) followed by the postphase (postunfolding) to t. □

We have more to say on the efficiency of residual programs in Chapter 11.

3.3 Generalizing

In this section we describe a transformation, *generalization*, on M_1 programs which can be applied manually to suspend transformation. In part III we shall encounter several reasons for wishing to suspend transformation along with automatic ways of finding suitable generalizations to achieve the desired suspensions.

Recall from Section 2.2 that local definitions are not intended for the user, but as a special form to make certain terms stand out clearly for metaalgorithms.

EXAMPLE **3.3.1** Consider the following variant of the double-append program.

$$\begin{array}{lcl} & & \text{let } vs = a\ zs\ ws \text{ in } a\ vs\ ts \\ a\ Nil\ ys & = & ys \\ a\ (Cons\ x\ xs)\ ys & = & Cons\ x\ (a\ xs\ ys) \end{array}$$

Applied to the above program $\mathcal{W}$ yields the exact same term and program unchanged, so the inefficiency has not been removed. □

The operation of turning the term $a\ (a\ zs\ ws)\ ts$ into the term let $vs = a\ zs\ ws$ in $a\ vs\ ts$ is called generalization. More specifically we introduce the following terminology.

DEFINITION **3.3.2** (Generalization.) Let $e()$ denote a term with exactly one occurrence of () at a place where a subterm could have occurred, and let $e(t)$ denote the result of substituting t for the occurrence of ().

The operation of turning $e(h\,t_1 \ldots t_n)$ into

$$\text{let } v = t_i \text{ in } e(h\,t_1 \ldots t_{i-1}\,v\,t_{i+1} \ldots t_n)$$

is called *generalization of h's i'th argument*, and turning $e(h\,t_1 \ldots t_n)$ into

$$\text{let } v = h\,t_1 \ldots t_n \text{ in } e(v)$$

is called *generalization of the call to h.* □

In the remainder of the paper, the reader should bear in mind that the *only* purpose of let-expressions in M_1 is to allow suspensions. One might say that the *real* language of our study is $M_{1/2}$.

3.4 A Burstall-Darlington explanation

A well-known framework for *fold/unfold* transformations was given in [Bur77]. According to this framework, one performs the following operations non-deterministically:

definition introduce a new definition whose left hand side is not an instance of the left hand side of any previous definition.

instantiation introduce a passive instance of an existing definition.

unfolding replace a function call by the body of the function after appropriate substitutions;

folding replace an instance of a function's right side by a call to that function;

abstraction replace an expression by a variable binding the expression. This can be done by means of a local definition, *e.g.* a call $f\ t$ can be transformed to let $v = t$ in $f\ v$.

laws use algebraic laws to transform terms.

The actions of $\mathcal{W}$ can be cast into the fold/unfold framework as follows. In clause (3b) the term is transformed into a call to a new *residual* function. This involves a *define* step: a new function is defined; an *instantiation* step: the new function is defined by patterns; an *unfold* step: the body of the new function is unfolded one step; and a *fold step*: the original term is replaced by a call to the newly defined function.

Clauses (2) and (3a) are similar except that there is no need for an instantiation step. We might also say that the instantiation step is trivial, regarding a variable as a trivial pattern and f-functions as defined by patterns. The operation of instantiation followed by unfolding of the different branches is called *driving* by Turchin.

Clauses (4a),(4b) can be understood as unfold steps similar to clauses (2),(3a), and clause (4c) can be understood as an instantiation step similar to clause (3b).

Clause (5) does not seem to have an immedieate explanation in the Burstall-Darlington framework.

This completes our Burstall Darlington explanation of positive supercompilation. Note that what we have called generalization in Section 3.3 is called abstraction in the Burstall-Darlington framework.

3.5 Postive supercompilation as generalized interpretation

Recall that we have already mentioned that the rewrite rules can be understood intuitively as mimicking the actions of a call-by-name evaluator. Indeed, we can imagine that $\mathcal{W}$ "thinks" as follows: I get a term t containing variables. At run-time the values for these variables will be supplied, yielding a ground term t', and thereby enough information to calculate the result of applying the interpreter to t'. I don't have those values, but let me see how much of the result I can figure out nevertheless.

When in clause (0) $\mathcal{W}$ encounters a variable, it really knows nothing about what this will be at run-time; consequently it simply returns the variable. In clauses (1a),(1b),(2),(3a) all the necessary information is present, and $\mathcal{W}$ does the same as $\mathcal{I}$ would. In clause (3b), $\mathcal{W}$ does not know what the value for v will be. Therefore it does not know which definition of g to pick. A simple decision would be to do as in clause (0), simply return $e[g\, v\, t_1 \ldots t_n]$ as the result. However, the present clause (3b) is vital for the positive supercompiler to be able to perform optimizations such as that mentioned in Section 3.2 as well as some of the others mentioned in part II. Basically, the positive supercompiler generates a residual case analysis of the variable and takes suitable actions in each of the cases.

The action in clauses (4a)-(4c) is similar to (3b). In clause (4a) the positive supercompiler discovers that the test in the conditional will always turn out true and therefore only takes the true-branch. Similarly, in clause (4b) it is discovered that the test always turns out false, and accordingly only the false-branch is taken. However, in clause (4c) it cannot be decided whether the conditional at run-time will turn out true or false, so the term is turned into a residual conditional and the outcome of the test of that conditional determines at run-time which branch is taken.

Finally, the action in clause (5) is *not* as for $\mathcal{I}$, for reasons which were explained in Section 3.3.

In conclusion: clauses (1a),(1b),(2) of $\mathcal{W}$ simulate clauses (1a),(1b),(2), respectively, of $\mathcal{I}$; clauses (3a),(3b) of $\mathcal{W}$ simulate clause (3) of $\mathcal{I}$; clauses (4a),(4b),(4c) together simulate clauses (4a),(4b) of $\mathcal{I}$; and clause (5) of $\mathcal{W}$ does not simulate clause (5) of $\mathcal{I}$.

3.6 The essence of driving

Recall that when $\mathcal{W}$ encounters a call of the form $e[b = b' \;\rightarrow\; t \;\Box\; t']$, where b, b' have a most general unifier $\{v_i := t_i\}_{i=1}^{n}$, the result is a conditional. An important feature of $\mathcal{W}$ is that in the true-branch it takes into account that the test is assumed to be true; that is, in the true-branch the unifier is applied to the term to be transformed.

EXAMPLE **3.6.1** $\mathcal{W}$ will transform the term $v = A \;\rightarrow\; (v = B \;\rightarrow\; 0 \;\Box\; 1) \;\Box\; 2$ into $v = A \;\rightarrow\; 1 \;\Box\; 2$. When $\mathcal{W}$ goes to the true-branch of the outer conditional it records that v is A by the substitution in clause (4c), and so the true-branch in the inner conditional will be found to be impossible by clause (4b). $\Box$

Also recall that when $\mathcal{W}$ encounters a call of the form $e[g\ v\ t_1 \ldots t_n]$, a residual function is defined by patterns on v. An important feature of $\mathcal{W}$ is that it takes into account each pattern of v in the corresponding clause; that is, in the clause corresponding to the pattern p for the residual function $g^{\square}$, p is substituted for v.

EXAMPLE **3.6.2** Consider the following term and program.

$$
\begin{array}{lcl}
 & & g\ v\ v \\
g\ (Cons\ x\ xs)\ y & \leftarrow & h\ y \\
g\ Nil\ y & \leftarrow & y \\
h\ (Cons\ z\ zs) & \leftarrow & zs \\
h\ Nil & \leftarrow & Nil
\end{array}
$$

Transformation starts with $g\ v\ v$. By clause (3b) the result of transforming this term is a call $g_1\ v$ to a new function g_1. Initially the right hand sides of g_1 are defined to be $h\ (Cons\ x\ xs)$ and Nil. The first of these terms is then transformed to a call $f_2\ x\ xs$, and the function f_2 is initially defined to have right hand side xs.

The crucial point in this example is that when v was instantiated to $Cons\ x\ xs$, then *both* occurrences of v were instantiated, and due to the instantiation of the second occurrence of v we knew which clause to choose for h later on. □

It is noteworthy that we are propagating only *positive* information. Let us first realize what this means for equality tests. In clause (4c) of $\mathcal{W}$ we propagate information that a test was true to the true-branch. This information asserts that certain variables have certain values. We propagate the information simply by applying the unifier to the term in the true-branch. However, in the false-branch we do not propagate the *negative* information, that is, that the test failed. Such information restricts the values which variables can take. Both positive and negative information can arise from an equality test, but we propagate only the positive information. We note in passing that negative information does not seem to be representable by substitution (what instantiation should one make to express the fact that v is not equal to w?)

Now let us turn to tests on patterns. When we instantiate in clause (3b) of $\mathcal{W}$, one might say that we test what v is and propagate the resulting information to each of the branches; that is, we represent once again positive information by application of a substitution. But there is no notion of negative information arising from such a test. Negative information occurs only in the case of (implicit or explicit) "else" or "otherwise" constructs.

Before closing this section, a remark concerning the equivalence of having functions defined by patterns on one hand, and adopting an explicit **case** construct on the other hand, seems appropriate. As is well known, either approach can be translated to the other. For instance, a function defined on patterns in our language

$$
\begin{array}{c}
g\ p_1\ v_1 \ldots v_n \leftarrow t_1 \\
\vdots \\
g\ p_m\ v_1 \ldots v_n \leftarrow t_m
\end{array}
$$

can be expressed by **case** by the definition

$$g\ v\ v_1 \ldots v_n = \mathbf{case}\ v\ \mathbf{of}\ p_1 : t_1 \mid \ldots \mid p_m : t_m$$

In the latter case positive information propagation means that when we instantiate v and go to each of the branches, we should instantiate occurrences of v in the branches to the pattern of the branch in question, *i.e.*

$$T[\![\ \mathbf{case}\ v\ \mathbf{of}\ p_1 : t_1 \mid \ldots \mid p_m : t_m\ ]\!] \Rightarrow \mathbf{case}\ v\ \mathbf{of}\ T[\![\ p_1 : t_1\{v := P_1\}\ ]\!] \mid \ldots \mid T[\![\ p_m : t_m\{v := p_m\}\ ]\!]$$

This is probably easier to understand than the notion for functions defined by patterns.[4]

Transforming a program in our language with $\mathcal{W}$ is the same as transforming programs in a **case**-language with a version of $\mathcal{W}$ formulated for the **case**-language which uses the above rule.

[4]The reason why we have nevertheless adopted the latter approach is that terms using case grow large during transformation. This makes examples less concise; compare *e.g.* [Wad88] to [Fer88].

3.7 Overview of correctness issues

Although the whole of part III is devoted to the correctness of the positive supercompiler, it may be helpful with an overview of the issues of correctness for $\mathcal{W}$. This is provided by the present section.

There are three issues of correctness for $\mathcal{W}$ and related fold/unfold transformers, see *e.g.* Futamura's survey paper [Fut83, page 15]: preservation of input/output-behaviour, termination, and nondegradation of efficiency.

As for the first, if we transform the body of a function in some program then we would like the new function to yield the same result as the original in any application. What this means more precisely is stated in Chapter 10 which also proves that $\mathcal{W}$ in fact preserves semantics.

As for the second, we would like $\mathcal{W}$ to terminate, or at least we should know some classes of terms and programs for which it terminates. This is the subject of Chapters 12, 13, and 14.

As for the third, we would like the transformed program to be at least as efficient as the original program; otherwise there is hardly any point in the transformation. Chapter 11 is concerned with this problem.

Chapter 4

Trees and Graphs as Interpretation and Transformation

> Once a perfect driving mechanism is constructed,
> it is a solid ground for the further development.
> As a result, the problem of approximation
> has been driven into one corner: folding.
>
> *Robert Glück and Andrei Klimov.*[1]

Given a program and a ground term the rewrite interpreter from Chapter 2 can calculate the result, and given a program and a term the positive supercompiler can calculate a new term and program. As such, these two algorithms serve their purpose well. However, there are situations for which these algorithms are inappropriate, *viz.* when it is necessary to consider the individual steps towards the result of either $\mathcal{I}$ or $\mathcal{W}$ and when the result is infinite. In this chapter we develop a machinery that allows us to discuss such issues in terms of certain trees and graphs. The development is similar to that in [Glu93a] of *process trees* and *graphs*, but whereas the development in that paper is informal and intuitive, the present development is rigorous and precise.

Section 4.1 describes interpretation trees. These should be considered as a more precise formulation of the call-by-name semantics of M_1 than $\mathcal{I}$. Section 4.2 describes transformation trees and graphs which have several important applications. For now, the reader should just think of them as a more precise formulation of $\mathcal{W}$. Section 4.3 describes the relation between the interpretation trees and transformation trees. Section 4.4 describes the relation between residual programs and transformation graphs. Section 4.5 describes a measure of quality on residual programs and transformation graphs.

In the subsequent chapters we are sometimes concerned with $\mathcal{I}$, sometimes with interpretation trees; and similarly sometimes we deal with $\mathcal{W}$, sometimes with transformation trees and graphs. The choice in each case is determined by what notions are most conevenient.

4.1 Interpretation trees

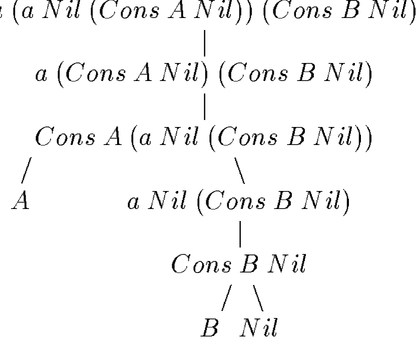
The idea of an *interpretation tree* is that it traces the evaluation of a program and ground term. Every node contains one term, and every step from parent to children records one step of $\mathcal{I}$. (Arcs are directed downwards.)

EXAMPLE **4.1.1** As an example of an *interpretation tree* consider the append function.

$$\begin{array}{lcl} a\ Nil\ ys & = & ys \\ a\ (Cons\ x\ xs)\ ys & = & Cons\ x\ (a\ xs\ ys) \end{array}$$

[1] In [Glu93a].

For the term $a\ (a\ Nil\ (Cons\ A\ Nil))\ (Cons\ B\ Nil)$ we get the following interpretation tree.



□

DEFINITION **4.1.2** (Interpretation tree.[2]) Let the notation $[l_1, \ldots, l_n]$ denote ordered lists. Let t_0 be a ground term. Define the *interpretation tree* $\mathcal{T}_{\mathcal{I}}[\![\, t_0 \,]\!]$ inductively as follows. The root contains t_0. If a node N contains t and $\mathcal{N}_{\mathcal{I}}[\![\, t \,]\!] = [t_1, \ldots, t_n]$, then N has left to right children $N_1 \ldots N_n$, where N_i contains t_i, and where $\mathcal{N}_{\mathcal{I}}$ is defined as follows.

$(1a)$	$\mathcal{N}_{\mathcal{I}}[\![\, c\, t_1 \ldots t_n \,]\!]$	$=$	$[]$	**if** all t_i are passive
$(1b)$	$\mathcal{N}_{\mathcal{I}}[\![\, c\, t_1 \ldots t_n \,]\!]$	$=$	$[t_1, \ldots, t_n]$	**if** not all t_i are passive
(2)	$\mathcal{N}_{\mathcal{I}}[\![\, e[f\, t_1 \ldots t_n] \,]\!]$	$=$	$[e[t^f\{v_i^f := t_i\}_{i=1}^n]]$	
(3)	$\mathcal{N}_{\mathcal{I}}[\![\, e[g\ (c\, t_{n+1} \ldots t_{n+m})\, t_1 \ldots t_n] \,]\!]$	$=$	$[e[t^{g,c}\{v_i^{g,c} := t_i\}_{i=1}^{n+m}]]$	
$(4a)$	$\mathcal{N}_{\mathcal{I}}[\![\, e[b = b' \ \rightarrow\ t \ \square\ t'] \,]\!]$	$=$	$[e[t]]$,	if $b \equiv b'$
$(4b)$	$\mathcal{N}_{\mathcal{I}}[\![\, e[b = b' \ \rightarrow\ t \ \square\ t'] \,]\!]$	$=$	$[e[t']]$,	if $b \not\equiv b'$
(5)	$\mathcal{N}_{\mathcal{I}}[\![\, e[\text{let } v = t \text{ in } t'] \,]\!]$	$=$	$[e[t'\{v := t\}]]$	

□

So, the interpretation tree is computed by starting out with a node containing a ground term and repeatedly applying $\mathcal{N}_{\mathcal{I}}$ to all current leaf nodes, yielding new child nodes. Informally, each application of $\mathcal{N}_{\mathcal{I}}$ corresponds to one step of the rewrite interpreter. Note specifically that $\mathcal{N}_{\mathcal{I}}[\![\, t \,]\!] = [\ldots t' \ldots]$ iff computing $\mathcal{I}[\![\, t \,]\!]$ entails computing $\mathcal{I}[\![\, t' \,]\!]$ in the next step.

Interpretation trees, and transformation trees and graphs introduced in the next section, are convenient for discussing termination of $\mathcal{I}$ and $\mathcal{W}$. Such notions are *a priori* imprecise, at least in the case of $\mathcal{W}$, because we have not specified the semantics of the metalanguage.

DEFINITION **4.1.3** For a ground term t in some program p, the set of terms that $\mathcal{I}[\![\, t \,]\!]$ encounters is the set of terms occurring in $\mathcal{T}_{\mathcal{I}}[\![\, t \,]\!]$. We say that $\mathcal{I}[\![\, t \,]\!]$ terminates (and that (t, p) is $\mathcal{I}$-terminating) if $\mathcal{T}_{\mathcal{I}}[\![\, t \,]\!]$ is finite. □

The connection between $\mathcal{I}$-termination and $\perp$ in the result of $\mathcal{I}$ is given in the following proposition.

PROPOSITION **4.1.4** *For a ground term t, there is a $b \neq \perp$ such that $\mathcal{I}[\![\, t \,]\!] \Rightarrow b$ iff $\mathcal{I}[\![\, t \,]\!]$ terminates.* □

PROOF: We must prove that $\mathcal{I}[\![\, t \,]\!] \Rightarrow b$ where b is not $\perp$ iff $\mathcal{T}_{\mathcal{I}}[\![\, t \,]\!]$ is finite, but this is easy. □

We end this section by showing how the result $b \not\equiv$ of $\mathcal{I}[\![\, t \,]\!]$ can be recovered from a finite interpretation tree.

DEFINITION **4.1.5** Given a finite interpretation tree I. For a node N define $\mathcal{R}(N)$ recursively as follows. If N contains $c\, t_1 \ldots t_k$ and has children $N_1 \ldots N_k$, then $\mathcal{R}(N)$ is $c\, \mathcal{R}(N_1) \ldots \mathcal{R}(N_k)$. *(ii)* If N contains $c\, t_1 \ldots t_k$ and has no children, then $\mathcal{R}(N)$ is $c\, t_1 \ldots t_k$. Otherwise N has one child and $\mathcal{R}(N) = \mathcal{R}(N_1)$. Now define the *value* of I as $\mathcal{R}(M)$ where M is the root of I. □

[2]Peter Sestoft has pointed out the similarity with "proof trees" in structural operational semantics, and David Sands has pointed out the similarity with algebraic semantics.

PROPOSITION **4.1.6** *Suppose that $\mathcal{T}_{\mathcal{I}}[\![t]\!]$ for a ground t is finite and that $\mathcal{I}[\![t]\!] \Rightarrow b$. Then b is the value of $\mathcal{T}_{\mathcal{I}}[\![t]\!]$.* □

PROOF: Easy. □

So when computation terminates, the rewrite interpreter and the interpretation semantics yield the same. However, when $\mathcal{I}$ does not terminate, *i.e.* when $\mathcal{I}[\![t]\!] \Rightarrow \bot$, the interpretation tree $\mathcal{T}_{\mathcal{I}}[\![t]\!]$ still gives details about the computation. We shall encounter an example in Chapter 10.1 where this is relevant. Also, interpretation trees are often convenient for showing that $\mathcal{I}[\![t]\!]$ terminates.

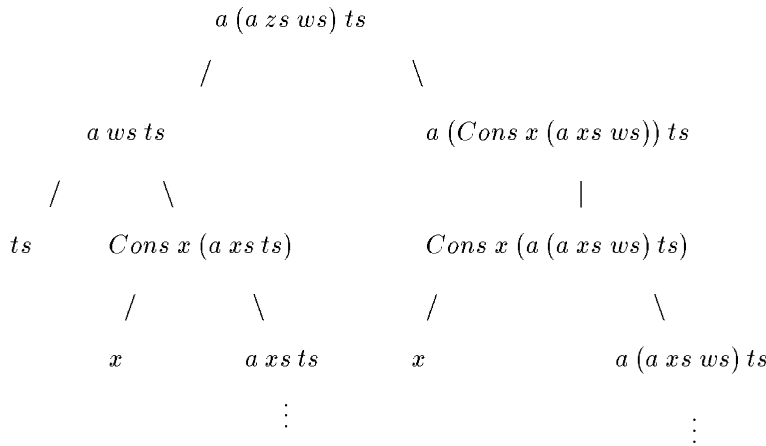
4.2 Transformation trees and graphs

The idea of a *transformation tree* is that it traces the transformation of a term and program. Every node contains one term, and every step from parent to children records one step of $\mathcal{W}$.

EXAMPLE **4.2.1** Suppose that we are to apply $\mathcal{W}$ to the term $a\ (a\ zs\ ws)\ ts$ where a is append from Example 3.2.5 which the reader may like to consult again before moving on.

The first term that occurs as argument to $\mathcal{W}$ is $a\ (a\ zs\ ws)\ ts$. The immediate result is a call to a new function g_1, which initially has right hand sides $a\ ws\ ts$ and $a\ (Cons\ x\ (a\ xs\ ws))\ ts$. $\mathcal{W}$ is then applied to each of these. For the second of the two terms, the result is a call to a new function f_1 with initial right hand side $Cons\ x\ (a\ (a\ xs\ ws)\ )ts$. Then $\mathcal{W}$ is applied to each of the subterms of the constructor, *etc.*

We can write the terms that occur as arguments to $\mathcal{W}$ in a not necessarily finite, *transformation tree*, where each node contains a term, and the children of a node N contain the terms that arise by one step of $\mathcal{W}$ from the term in N.



□

It will be convenient to talk about not only trees but also graphs. By a rooted *transformation graph* we mean the same as a transformation tree except that when the same term MVR with an f-function call or g-function call in the redex is encountered twice in a branch (path from the root), then we make an arc from the parent of the second occurrence to the first occurrence and do not develop the branch from the parent of the second occurrence any further.

In the above example, the transformation graph is obtained from the tree as follows. call the node just above the rightmost vertical dots N. The parent of N has an arc to the root, and the subtree with N as a root is deleted. The node just above the leftmost vertical dots is also deleted, and its parent has an arc to the node containing $a\ ws\ ts$.

We now make these notions rigorous.

DEFINITION **4.2.2** (Transformation tree and graph.) Let the notation $[l_1, \ldots, l_n]$ denote ordered lists, and let $[l_j]_{j=1}^n = [l_1, \ldots l_n]$. Let t_0 be a term. Define the *transformation tree* $\mathcal{T}_\mathcal{W}[\![\ t_0\ ]\!]$ inductively as follows. The root contains t_0. If a node N contains t and $\mathcal{N}_\mathcal{W}[\![\ t\ ]\!] = [t_1, \ldots, t_n]$, then N has left to right children $N_1 \ldots N_n$, where N_i contains t_i, and where $\mathcal{N}_\mathcal{W}$ is defined as follows.

$$
\begin{array}{lll}
(0) & \mathcal{N}_\mathcal{W}[\![\ v\ ]\!] & = [] \\
(1a) & \mathcal{N}_\mathcal{W}[\![\ c\, t_1 \ldots t_n\ ]\!] & = [] \\
 & \textbf{if } \text{all } t_i \text{ are passive} & \\
(1b) & \mathcal{N}_\mathcal{W}[\![\ c\, t_1 \ldots t_n\ ]\!] & = [t_1, \ldots, t_n] \\
 & \textbf{if } \text{not all } t_i \text{ are passive} & \\
(2) & \mathcal{N}_\mathcal{W}[\![\ e[f\, t_1 \ldots t_n]\ ]\!] & = [e[t^f\{v_i^f := t_i\}_{i=1}^n]] \\
(3a) & \mathcal{N}_\mathcal{W}[\![\ e[g\ (c\, t_{n+1} \ldots t_{n+m})\, t_1 \ldots t_n]\ ]\!] & = [e[t^{g,c}\{v_i^{g,c} := t_i\}_{i=1}^{n+m}]] \\
(3b) & \mathcal{N}_\mathcal{W}[\![\ e[g\ v\, t_1 \ldots t_n]\ ]\!] & = [(e[t^{g,c_j}\{v_i^{g,c_j} := t_i\}_{i=1}^n])\{v := p_j\}]_{j=1}^m \\
(4a) & \mathcal{N}_\mathcal{W}[\![\ e[b = b' \ \to\ t \,\Box\, t']\ ]\!] & = [e[t]] \\
 & \textbf{if } b, b' \text{ are ground and } b \equiv b' & \\
(4b) & \mathcal{N}_\mathcal{W}[\![\ e[b = b' \ \to\ t \,\Box\, t']\ ]\!] & = [e[t']] \\
 & \textbf{if } b, b' \text{ are ground and } b \not\equiv b' & \\
(4c) & \mathcal{N}_\mathcal{W}[\![\ e[b = b' \ \to\ t \,\Box\, t']\ ]\!] & = [(e[t])MGU(b, b'), e[t']] \\
 & \textbf{if } \text{not both } b, b' \text{ are ground} & \\
(5) & \mathcal{N}_\mathcal{W}[\![\ e[\text{let } v = t \text{ in } t']\ ]\!] & = [t, e[t']]
\end{array}
$$

The *transformation graph* $\mathcal{G}_\mathcal{W}[\![\ t_0\ ]\!]$ is defined as the graph obtained from $\mathcal{T}_\mathcal{W}[\![\ t_0\ ]\!]$ as follows. Start from the root and traverse all branches. Whenever a node N_2 is reached which contains a term t, where $t \equiv e[r]$ and r is an f-function call or a g-function call, and where a term identical to t MVR is contained in a previous node N_1 in the same branch, then let the parent of N_2 have an arc back to N_1, and delete the subtree with root N_2. □

So, the transformation tree is computed by starting out with a node containing a term and repeatedly applying $\mathcal{N}_\mathcal{W}$ to all current leaf nodes, yielding new children nodes. Informally, each application of $\mathcal{N}_\mathcal{W}$ corresponds to one step of $\mathcal{W}$. Note specifically that $\mathcal{N}_\mathcal{W}[\![\ t\ ]\!] \Rightarrow [\ldots t' \ldots]$ iff computing $\mathcal{W}[\![\ t\ ]\!]$ entails computing $\mathcal{W}[\![\ t'\ ]\!]$ in the next step. This motivates the following definition, which is not a proposition because the semantics of the metalanguage has not been specified, so that termination of $\mathcal{W}$ is not a precise notion.

DEFINITION **4.2.3** For a term t in some program p, the set of terms that $\mathcal{W}[\![\ t\ ]\!]$ encounters is the set of terms occurring in $\mathcal{G}_\mathcal{W}[\![\ t\ ]\!]$. We say that $\mathcal{W}[\![\ t\ ]\!]$ terminates (and that (t, p) is $\mathcal{W}$-terminating) if $\mathcal{G}_\mathcal{W}[\![\ t\ ]\!]$ is finite. □

It should be noted that transformation graphs are obtained from transformation trees by incorporating a folding scheme into the algorithm computing transformation trees, and this folding scheme is identical to the one in $\mathcal{W}$.

Transformation trees and graphs are very similar to, and indeed inspired by, *process trees* and *graphs*, respectively, as described by Glück and Klimov [Glu93a], which in turn stem from *graphs of states* in Turchin's papers [Tur80a,Tur80b].

4.3 Walks

REMARK **4.3.1** In this section we are concerned with $M_{1/2}$ terms and programs only. □

In the two preceding sections we have introduced interpretation trees and transformation trees and graphs as *computation histories* for the intepreter and positive supercompiler, respectively.

Recall from Section 3.5 that one can view $\mathcal{W}$ as a generalization of $\mathcal{I}$ to manipulate terms with free variables and take suitable actions to account for all run-time possibilities. In this section we show a similar results for trees, namely that one can view the transformation tree $\mathcal{T}_{\mathcal{W}}[\![t]\!]$ (and transformation graph $\mathcal{G}_{\mathcal{W}}[\![t]\!]$) as a model of all runs $\mathcal{T}_{\mathcal{I}}[\![t\theta]\!]$, where θ is a constant substitution, *i.e.* a mapping of variables to ground constants.

We first give an example of the notion of a *walk* which will be used to assign this significance to transformation trees and graphs.

EXAMPLE **4.3.2** Consider again the double-append term and append program. The transformation tree $\mathcal{T}_{\mathcal{W}}[\![a\ (a\ zs\ ws)\ ts]\!]$ was given in Section 4.2. Let $\theta = \{zs := Nil, ws := Cons\ A\ Nil, ts := Cons\ B\ Nil\}$. The following tree is the walk $\mathcal{V}_{\mathcal{T}}[\![a\ (a\ zs\ ws)\ ts]\!](\theta)$ in that transformation tree.

```
              |θ
       a (a zs ws) ts
              |θ
          a ws ts
              |θ'
      Cons x (a xs ts)
     /θ'           \θ'
    x            a xs ts
                   |θ'
                   ts
```

where $\theta' = \theta \cup \{x := A, xs := Nil\}$.

Recall that a branching in a transformation tree $\mathcal{T}_{\mathcal{W}}[\![t]\!]$ corresponds to different instantiations of a variable or different outcomes of an equality test (except for children of a node containing a constructor term with non-passive subterms.) The above example shows how one can choose a path (actually a subtree, since constructor terms may have several children) in a transformation tree $\mathcal{T}_{\mathcal{W}}[\![t]\!]$ by assigning constants to the variables of t.

Note how θ' arises from θ. When a branch in the transformation tree is chosen according to the value b of a variable v the new variables in the child are assigned the subvalues of b.

Consider the tree obtained by replacing the contents t of every node N by $t\sigma$, where σ is the substitution on the arc into N, and erasing the labels. This is the interpretation tree $\mathcal{T}_{\mathcal{I}}[\![t\theta]\!]$ with θ as above that we considered in Section 4.1. In other words: the walks $\mathcal{V}_{\mathcal{T}}[\![t]\!](\theta)$ of a transformation tree $\mathcal{T}_{\mathcal{W}}[\![t]\!]$ correspond to different runs $\mathcal{T}_{\mathcal{I}}[\![t\theta]\!]$. □

We now introduce walks precisely.

DEFINITION **4.3.3** (Walk in transformation tree and graph.) Let the notation $[l_1, \ldots, l_n]$ denote ordered lists. Let t_0 be a term, θ a constant substitution for t_0. Define the *walk* $\mathcal{V}_{\mathcal{T}}[\![t_0]\!](\theta)$ in the transformation tree $\mathcal{T}_{\mathcal{W}}[\![t_0]\!]$ inductively as follows. The root contains t_0, and there is an arc into the root labelled θ. If a node N contains t, θ is the substitution label of the arc into N, and $\mathcal{N}_{\mathcal{V}}[\![t]\!] = [(t_1, \theta_1), \ldots, (t_n, \theta_n)]$, then N has left to right children $N_1 \ldots N_n$, where N_i contains t_i, the arc from N to N_i is labelled with θ_i, and where $\mathcal{N}[\![$ is defined as follows.

(0)	$\mathcal{N}_\mathcal{V}[\![v]\!](\theta)$	$=$	$[]$
(1a)	$\mathcal{N}_\mathcal{V}[\![c\, t_1 \ldots t_n]\!](\theta)$ **if** all t_i are passive	$=$	$[]$
(1b)	$\mathcal{N}_\mathcal{V}[\![c\, t_1 \ldots t_n]\!](\theta)$ **if** not all t_i are passive	$=$	$[(t_1,\theta)\ldots,(t_n,\theta)]$
(2)	$\mathcal{N}_\mathcal{V}[\![e[f\, t_1 \ldots t_n]]\!](\theta)$	$=$	$[(e[t^f\{v_i^f := t_i\}_{i=1}^n],\theta)]$
(3a)	$\mathcal{N}_\mathcal{V}[\![e[g\ (c\, t_{n+1} \ldots t_{n+m})\, t_1 \ldots t_n]]\!](\theta)$	$=$	$[(e[t^{g,c}\{v_i^{g,c} := t_i\}_{i=1}^{n+m}],\theta)]$
(3b)	$\mathcal{N}_\mathcal{V}[\![e[g\ v\, t_1 \ldots t_n]]\!](\theta)$ **where** $\sigma = MGU(v\theta, p_j) \neq fail$, and $\theta' = \theta \circ \sigma$	$=$	$[(e[t^{g,c_j}\{v_i^{g,c_j} := t_i\}_{i=1}^n]\{v := p_j\},\theta')]$
(4a)	$\mathcal{N}_\mathcal{V}[\![e[b = b' \ \to\ t \,\Box\, t']]\!](\theta)$ **if** b, b' are ground, and $b \equiv b'$	$=$	$[(e[t],\theta)]$
(4b)	$\mathcal{N}_\mathcal{V}[\![e[b = b' \ \to\ t \,\Box\, t']]\!](\theta)$ **if** b, b' are ground, and $b \not\equiv b'$	$=$	$[(e[t'],\theta)]$
(4c1)	$\mathcal{N}_\mathcal{V}[\![e[b = b' \ \to\ t \,\Box\, t']]\!](\theta)$ **if** not both of b, b' are ground, and $b\theta \equiv b'\theta$	$=$	$[((e[t])MGU(b,b'),\theta)]$
(4c2)	$\mathcal{N}_\mathcal{V}[\![e[b = b' \ \to\ t \,\Box\, t']]\!](\theta)$ **if** not both of b, b' are ground, and $b\theta \not\equiv b'\theta$	$=$	$[(e[t'],\theta)]$

The walk $\mathcal{V}_\mathcal{G}[\![t_0]\!](\theta)$ in the transformation graph $\mathcal{G}_\mathcal{W}[\![t_0]\!]$ is defined as the graph obtained from $\mathcal{V}_\mathcal{T}[\![t_0]\!](\theta)$ as follows. Start from the root and traverse all branches. Whenever a node N_2 is reached which contains t, where $t \equiv e[r]$ and r is a f-function call or a g-function call, t is identical MVR to t', σ is the renaming with $t\sigma \equiv t'$, and t' is contained in a previous node N_1 in the same branch, then let the parent of N_2 have an arc back to N_1 labelled by σ, and delete the subtree with root N_2. □

DEFINITION **4.3.4** (Tree morphisms, *etc.*) A *tree monomorhism* ϕ from a tree T to a tree T' is a mapping from nodes of T to nodes of T' such that: (i) if the root of T is N then the root of T' is $\phi(N)$; (ii) if the different children of N in T are $N_1 \ldots N_k$ then $\phi(N_1)\ldots\phi(N_k)$ are different children of $\phi(N)$ in T'.

If for every node N in T, $\phi(N_1)\ldots\phi(N_k)$ are *all* the different children of $\phi(N)$ in T', ϕ is a *tree isomorphism.*

T is *a subtree* of T' if the identity mapping is a tree monomorphism from T to T'.

Similar notions for graphs are obtained by replacing *children* by *descendants* above. □

The following proposition states that a walk essentially is a path in a transformation tree (and graph) and that a transformation tree (and graph) for a term and program is a model of all computations with the term and program, where each computation can be obtained as a walk.

PROPOSITION **4.3.5** *Given t and a constant substitution θ for t. Let $V_\mathcal{T} = \mathcal{V}_\mathcal{T}[\![t]\!](\theta)$, $T = \mathcal{T}_\mathcal{W}[\![t]\!]$, $V_\mathcal{G} = \mathcal{V}_\mathcal{G}[\![t]\!](\theta)$, $G = \mathcal{G}_\mathcal{W}[\![t]\!]$, $I = \mathcal{T}_\mathcal{I}[\![t\theta]\!]$.*

(1) The result obtained by erasing the labels on arcs in $V_\mathcal{T}$ is a subtree of T. (2) The result obtained by erasing the labels on arcs in $V_\mathcal{G}$ is a subgraph of G. (3) The result obtained by replacing the contents t of every node N of $V_\mathcal{T}$ by $t\sigma$, where σ is the substitution on the arc into N, and erasing the labels on arcs in $V_\mathcal{T}$ is I. □

PROOF: (1) and (2): Let for t, θ $\mathcal{N}_{\mathcal{V}}[\![t]\!](\theta) = [(t_1, \theta_1), \ldots, (t_n, \theta_n)]$, $\mathcal{N}_{\mathcal{W}}[\![t]\!] = [t'_1, \ldots, t'_m]$, and note that $[t_1, \ldots, t_n]$ is a sublist of $[t'_1, \ldots, t'_m]$.

(3): Let for t, θ $\mathcal{N}_{\mathcal{V}}[\![t]\!](\theta) = [(t_1, \theta_1), \ldots, (t_n, \theta_n)]$, $\mathcal{N}_{\mathcal{I}}[\![t\theta]\!] = [t'_1, \ldots, t'_m]$, and prove by cases on t that $n = m$ and $t_i \equiv t'_i\theta$. (For the technicalities concerning MGU's see the proof of Proposition 10.2.2.)
□

By (i) it should be clear what is meant by the question whether a node in a transformation tree is contained in some walk.

4.4 Residual programs from finite transformation graphs

The preceding two sections have introduced transformation trees (and graphs) as computation histories for $\mathcal{W}$ and as models of computations by $\mathcal{I}$. This section adds a new significance to transformation trees (and graphs): we show how one can derive the residual program and term p', t' from the transformation tree (and graph.)

To actually provide a constructive way of computing residual programs, we shall be concerned with finite transformation graphs only.

The basic idea is that every term $e[r]$ in the transformation graph, where r is a f-function call or g-function call, gives rise to a new definition, and the body, or bodies, of the new function can be derived from the descendants of $e[r]$ in the graph.

DEFINITION **4.4.1** (Residual program from finite transformation graph.) Let the notation $[l_1, \ldots, l_n]$ denote ordered lists. Let $T = \mathcal{G}_{\mathcal{W}}[\![t_0]\!]$ for some term t_0. Let t be the term in the root of T. The residual term is then $\mathcal{R}[\![t]\!]$, and the residual program contains:

(i)

$$f^{\square}\ u_1 \ldots u_k \leftarrow \mathcal{R}[\![t]\!]$$

for every node N containing $e[f\, t_1 \ldots t_n]$ with descendant containing t;

(ii)

$$f^{\square}\ u_1 \ldots u_k \leftarrow \mathcal{R}[\![t]\!]$$

for every node N containing $e[g\ (c\, t_{n+1} \ldots t_{n+m})\ t_1 \ldots t_n]$ with descendant containing t;

(iii)

$$\begin{array}{rcl} g^{\square}\ p_1\, u_1 \ldots u_k & \leftarrow & \mathcal{R}[\![t_1]\!] \\ & \vdots & \\ g^{\square}\ p_m\, u_1 \ldots u_k & \leftarrow & \mathcal{R}[\![t_m]\!] \end{array}$$

for every node N containing $e[g\ v\, t_1 \ldots t_n]$ with descendants containing $t_1 \ldots t_m$;

Here $\mathcal{R}$ is defined as follows.

$$
\begin{array}{lll}
(0) & \mathcal{R}[\![\, v \,]\!] & \Rightarrow \; v \\
(1a) & \mathcal{R}[\![\, c\, t_1 \ldots t_n \,]\!] & \Rightarrow \; c\, t_1 \ldots t_n \\
 & \quad \textbf{if } \text{all } t_i \text{ are passive} & \\
(1b) & \mathcal{R}[\![\, c\, t_1 \ldots t_n \,]\!] & \Rightarrow \; c\, (\mathcal{R}[\![\, t_1 \,]\!]) \ldots (\mathcal{R}[\![\, t_n \,]\!]) \\
 & \quad \textbf{if } \text{not all } t_i \text{ are passive} & \\
(2) & \mathcal{R}[\![\, e[f\, t_1 \ldots t_n] \,]\!] & \Rightarrow \; f^{\Box}\, u_1 \ldots u_k \\
(3a) & \mathcal{R}[\![\, e[g\, (c\, t_{n+1} \ldots t_{n+m})\, t_1 \ldots t_n] \,]\!] & \Rightarrow \; f^{\Box}\, u_1 \ldots u_k \\
(3b) & \mathcal{R}[\![\, e[g\, v\, t_1 \ldots t_n] \,]\!] & \Rightarrow \; g^{\Box}\, v\, u_1 \ldots u_k \\
(4a) & \mathcal{R}[\![\, e[b = b' \;\rightarrow\; t \,\Box\, t'] \,]\!] & \Rightarrow \; \mathcal{R}[\![\, e[t] \,]\!] \\
 & \quad \textbf{if } b, b' \text{ are ground and } b \equiv b' & \\
(4b) & \mathcal{R}[\![\, e[b = b' \;\rightarrow\; t \,\Box\, t'] \,]\!] & \Rightarrow \; \mathcal{R}[\![\, e[t'] \,]\!] \\
 & \quad \textbf{if } b, b' \text{ are ground and } b \not\equiv b' & \\
(4c) & \mathcal{R}[\![\, e[b = b' \;\rightarrow\; t \,\Box\, t'] \,]\!] & \Rightarrow \; b = b' \;\rightarrow\; \mathcal{R}[\![\, e[t] \,]\!] \,\Box\, \mathcal{R}[\![\, e[t'] \,]\!] \\
 & \quad \textbf{if } \text{not both of } b, b' \text{ are ground} & \\
(5) & \mathcal{R}[\![\, e[\text{let } v = t \text{ in } t'] \,]\!] & \Rightarrow \; \text{let } v = \mathcal{R}[\![\, t \,]\!] \text{ in } \mathcal{R}[\![\, e[t'] \,]\!]
\end{array}
$$

□

The main difference between computing residual programs with $\mathcal{W}$ and as described in the preceding definition is that in the transformation graph all unfolding and folding of function calls has already been done. The following proposition shows that the approaches are equivalent.

PROPOSITION **4.4.2** *Given term t and program p. Then either (1) $\mathcal{W}[\![\, t \,]\!]$ does not terminate and the transformation graph $\mathcal{G}_{\mathcal{W}}[\![\, t \,]\!]$ is infinite; or (2) $\mathcal{W}[\![\, t \,]\!]$ terminates, the transformation graph $\mathcal{G}_{\mathcal{W}}[\![\, t \,]\!]$ is finite, and $(t_1, p_1) \equiv (t_2, p_2)$ where t_1, p_1 are the residual term and program computed by $\mathcal{W}$ and t_2, p_2 are the term and program as computed in the above definition.* □

PROOF: By definition, both or none of the two conditions in (1) are true. It is easy to see that (2) holds, provided that (1) does not. □

4.5 Perfect transformation trees and graphs

This section attempts to make precise the notion of a *perfect process tree* which was considered rather informally in [Glu93a].

REMARK **4.5.1** In the present section we shall be concerned with the sublanguage $M_{1/2}$ of M_1, see Section 2.4. □

EXAMPLE **4.5.2** Consider the following program and term.

$$
\begin{array}{lcl}
 & & f\, x \\
f\, x & \leftarrow & x = Zero \;\rightarrow\; Zero \,\Box\, g\, x \\
g\, y & \leftarrow & y = Zero \;\rightarrow\; Succ\; Zero \,\Box\, y
\end{array}
$$

The term $f\ x$ in fact always returns x. The point is that a call to g from f can never enter the true-branch of the conditional in g. Not applying postunfolding, the residual program and term are identical to the originals. So the residual program contains a redundant test. The redundant test comes from an obvious redundant test in the original program, but we shall in Chapter 8 encounter a program where the redundant test in the original program is far more subtle. The reason why the redundant test is not removed is that $\mathcal{W}$ propagates only positive information, *i.e.* information to the true-branch. If $\mathcal{W}$ also propagated negative information, *i.e.* information to the false-branch, it would have seen that a call from f to g can never enter the true-branch of g, and so would have produced the residual term and program:

$$\begin{array}{lcl} & & f\ x \\ f\ x & \leftarrow & x = Zero \rightarrow Zero \,\square\, g\ x \\ g\ y & \leftarrow & y \end{array}$$

The transformation tree for the above term and program is:

```
                      f x
                       |
             x = Zero  ->  x □ g x
        /                     \
     Zero                     g x
                               |
             x = Zero  ->  Succ Zero □ x
                             /        \
                       Succ Zero       x
```

Informally, there is no application of the original term and program that follows the branch in this tree ending in *Succ Zero*. To be precise: there is no walk in the transformation tree that contains the node with *Succ Zero*. □

Redundant tests in residual programs stem from redundancies in the original program. As seen in the example, by further information propagation we could prevent the redundancy of the original program to appear in the residual program. To isolate the problem more precisely consider supercompilation as the construction of trees, folding these into graphs, and producing residual programs from these graphs. When the redundancy from the original program appears in the residual program it is reflected in the intermediate tree by nodes that are not contained in any walk.

DEFINITION **4.5.3** A transformation tree $\mathcal{T}_{\mathcal{W}}\llbracket\, t \,\rrbracket$ is called *perfect* if for all nodes N there is a constant substitution θ such that the walk $\llbracket\ {}_{\mathcal{T}}\llbracket\, t \,\rrbracket$ contains the node N. A transformation graph is perfect if for all nodes N there is a constant substitution θ such that the walk $\llbracket\ {}_{\mathcal{G}}\llbracket\, t \,\rrbracket$ contains the node N. □

As mentioned, the transformation trees $\mathcal{T}_{\mathcal{W}}\llbracket\, t \,\rrbracket$ are not generally perfect. The "problem" is that $\mathcal{W}$ (and thereby transformation trees $\mathcal{T}_{\mathcal{W}}$) does not propagate negative information. In Chapters 5, 7, and 8 we shall see that the positive supercompiler can nevertheless achieve the same effects as those which have been given in the literature for transformers which construct perfect transformation trees.

For M_0 programs, the transformation trees are perfect. This is because there is no notion of negative information for M_0 functions, and so nothing is lost by propagating only positive information. If M_0 programs allowed "catch-all" clauses, *i.e.* definitions

$$\begin{array}{lcl} g\ p_1\ u_1 \ldots u_k & \leftarrow & s_1 \\ & \vdots & \\ g\ p_m\ u_1 \ldots u_k & \leftarrow & s_m \\ g\ x\ u_1 \ldots u_k & \leftarrow & s_{m+1} \end{array}$$

which would be transformed

$$\mathcal{W}\llbracket\, e[g\ v\ t_1 \ldots t_n] \,\rrbracket \Rightarrow g^{\square}\ v\ u_1 \ldots u_k$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{where}} \\
g^{\Box}\, p_1\, u_1 \ldots u_k & \leftarrow & \mathcal{W}[\![\, e[s_1\{u_i := t_i\}_{i=1}^{n}]\{v := p_1\}\,]\!] \\
 & \vdots & \\
g^{\Box}\, p_m\, u_1 \ldots u_k & \leftarrow & \mathcal{W}[\![\, e[s_m\{u_i := t_i\}_{i=1}^{n}]\{v := p_m\}\,]\!] \\
g^{\Box}\, x\, u_1 \ldots u_k & \leftarrow & \mathcal{W}[\![\, e[s_{m+1}\{u_i := t_i\}_{i=1}^{n}]\,]\!]
\end{array}
$$

then M_0 transformation trees would no longer be perfect.

It is appropriate to note a small point in the definition of perfectness: A tree may be perfect even if there is a path in it that is not part of any walk. Specifically, it may be the case that a transformation tree is infinite and yet all walks are finite.

We end the chapter with a brief review of related work. In [Glu93a] it is stated that it is possible to construct, in general, perfect trees, but not perfect *finite* graphs. [Glu93a] suggests as a criterion for comparing transformers, to see whether they construct perfect trees. In [Tur80a] where the notion of perfect graphs is introduced and suggested as a means of comparing programs it is shown that it is not generally possible to construct perfect graphs from programs.[3]

[3]Peter Sestoft has noted that in the context of partial evaluation, Weise and Ruf have considered transformation in a similar style, representing residual programs by graphs. The author was not aware of the details of this work.

Part II

Applications

Chapter 5

Effects of Positive Supercompilation

It seems plausible that the number of metasystem transitions
we have to make in the computational approach
is equal to the number of loops of induction.

V.F. Turchin.[1]

This chapter presents three types of applications of positive supercompilation.

Section 5.1 shows that $\mathcal{W}$ can eliminate intermediate data structures from programs. Section 5.2 shows that $\mathcal{W}$ can achive program specialization. Section 5.3 describes $\mathcal{W}$ as a theorem prover. Three other applications are studied in more detail in the next three chapters.

Each section first introduces the problem, and if relevant its history, shows by examples that $\mathcal{W}$ can solve the problem, then explains by investigation of the internal machinery how $\mathcal{W}$ achieves the effect that solves the problem, and finally states a general result concerning the achivement of the effect in question, if possible.

It should be clear that we differentiate between a program transformer and the effect it achives; for instance, partial evaluation achieves the specialization of a program to some known inputs. In Chapter 9 we compare the machinery of $\mathcal{W}$ with the internal machinery of other transformers, which is why we make only brief connections to related transformers in the present chapter. This comparison will explain why these transformers can or cannot obtain similar effects.

The exposition in this chapter, and the next three chapters, serves five purposes: (*i*) to show *what* Turchin's supercompiler can do; (*ii*) to show *what* the positive supercompiler can do, *viz.* the same as Turchin's to a large extent; (*iii*) to show *how* the positive supercompiler achieves its effects, *i.e.* to show what features of the positive supercompiler are needed for which applications; (*iv*) to lay a ground for comparisons with other program transformation methodologies which can achieve one or more of the effects mentioned; (v) in part III we shall be concerned with means of ensuring termination of the positive supercompiler. In that connection it is important to have a variety of *types* of applications of supercompilation to test the techniques on.

5.1 Elimination of intermediate data structures

Modern, lazy functional programming language such as Miranda or Haskell support a certain modular programming style which uses intermediate data strucures; Hughes [Hug90a] gives illuminating examples. For instance the program in Example 3.2.5 appends three lists by appending the two first, yielding an intermediate list, and then appending the last list to that. Another point of view is that we are *merging loops*; the first loop appends two lists, and the second loop appends the third. In lazy languages, function

[1] In [Tur80c].

composition represents, in a sense, loop composition where the steps of the loops are interleaved. A third view is that we are turning a program that passes over a structure several times into a one-pass program.

While such programs are usually very elegant they are also inefficient, because construction and subsequent destruction of data structures takes up time and space: (*i*) time to actually compute; (*ii*) time to garbage collect; (*iii*) space for the structures. Work has been done to eliminate intermediate data structures in programs automatically, see Section 9.1. In the context of supercompilation, such work occurs in most of Turchin's papers, see [Tur80a,Tur82,Tur93] for some early and recent results.

EXAMPLE **5.1.1** As a first example, which has already been mentioned, recall that in Example 3.2.5, the positive supercompiler turned the term and program

$$\begin{array}{lcl} & & a\ (a\ zs\ ws)\ ts \\ a\ Nil\ ys & \leftarrow & ys \\ a\ (Cons\ x\ xs)\ ys & \leftarrow & Cons\ x\ (a\ xs\ ys) \end{array}$$

into

$$\begin{array}{lcl} g_1\ Nil\ ws\ ts & \leftarrow & g_2\ ws\ ts \\ g_1\ (Cons\ x\ xs)\ ws\ ts & \leftarrow & Cons\ x\ (g_1\ xs\ ws\ ts) \\ g_2\ Nil\ ts & \leftarrow & ts \\ g_2\ (Cons\ x\ xs)\ ts & \leftarrow & Cons\ x\ (g_2\ xs\ ts) \end{array}$$

where the former constructs intermediate data structures, but the latter does not. □

Now let us see what it is in $\mathcal{W}$ that allows the elimination of intermediate data structures. Consider a term

$$g_1\ (g_2\ \ldots (g_n\ (f\ t_1 \ldots t_k)\ t_1^n \ldots t_{m_n}^n) \ldots)\ t_1^1 \ldots t_{m_1}^1 \quad (n \geq 0)$$

where the terms $t_1^i \ldots t_{m_i}^i$ are arguments of g_i. The idea is that if the redex $f\,t_1 \ldots t_n$, through a number of unfold steps, becomes a term with an outermost constructor, it *produces* a constructor (using the terminology of Chin [Chi90].) This allows the surrounding g-function to be unfolded, *consuming* exactly the outermost constructor from the term, since patterns are one constructor deep. This latter unfolding can itself, through a number of subsequent unfoldings, produce a constructor, thus allowing the next surrounding g-function to be unfolded. In this way, the constructor can propagate all the way to the root of the term. Once constructor propagation stops, the transformation proceeds to each of the arguments of the constructor in a similar fashion.

In Example 5.1.1 the innermost call was to a g-function, not a f-function, but the instantiation and unfolding in clause (3b) of $\mathcal{W}$ allowed the outwards propagation of the *Cons*-constructor. Once it reached the root, clause (1b) made $\mathcal{W}$ proceed with the components. It should be noted that in the example the substitutions $\{v := p_i\}$ in clause (3b) are inessential.

In the following example, which is very common in the literature on supercompilation, $\mathcal{W}$ turns a two-pass program into a one-pass program.

EXAMPLE **5.1.2** The following program consists of two functions, f_a, f_b. The first takes a list and turns all A's into B's, the second turns all B's into C's. The term turns all A's and B's into C's.

$$f_B\ (f_A\ xs)$$

$$\begin{array}{lcl} f_A\ Nil & \leftarrow & Nil \\ f_A\ (Cons\ x\ xs) & \leftarrow & x = A\ \rightarrow\ Cons\ B\ (f_A\ xs)\ \Box\ Cons\ x\ (f_A\ xs) \\ & & \\ f_B\ Nil & \leftarrow & Nil \\ f_B\ (Cons\ x\ xs) & \leftarrow & x = B\ \rightarrow\ Cons\ C\ (f_B\ xs)\ \Box\ Cons\ x\ (f_B\ xs) \end{array}$$

The positive supercompiler turns this term and program into the following.

$$\begin{array}{lcl} & & f_{AB}\ xs \\ f_{AB}\ Nil & \leftarrow & Nil \\ f_{AB}\ (Cons\ x\ xs) & \leftarrow & x = A\ \rightarrow\ Cons\ C\ (f_{AB}\ xs)\ \Box\ x = B\ \rightarrow\ Cons\ C\ (f_{AB}\ xs)\ \Box\ Cons\ x\ (f_{AB}\ xs) \end{array}$$

This program passes over its input only once. □

Here it is the instantiation and unfolding in clause (4c) together with the unfolding in clause (4a) that achieves the effect.

We close the section by a general result concerning $\mathcal{W}$'s elimination of intermediate data structures.

DEFINITION **5.1.3** Consider the classes of terms T, R, S.

$$
\begin{array}{lcl}
T & ::= & v \mid c\, T_1 \ldots T_n \mid f\, u_1 \ldots u_n \mid g\, v\, u_1 \ldots u_n \\
S & ::= & v \mid c\, S_1 \ldots S_n \mid f\, u_1 \ldots u_n \mid g\, v\, u_1 \ldots u_n \mid b_1 = b_2 \;\rightarrow\; S_1 \,\square\, S_2 \\
R & ::= & v \mid c\, R_1 \ldots R_n \mid f\, u_1 \ldots u_n \mid g\, v\, u_1 \ldots u_n \mid b_1 = b_2 \;\rightarrow\; R_1 \,\square\, R_2 \mid \text{let } v = R_1 \text{ in } R_2 \\
b & ::= & v \mid c\, b_1 \ldots b_n
\end{array}
$$

where $v, u_1, \ldots, u_k$ are variables.

A term is M_0 [$M_{1/2}$,M_1] *treeless* if it adheres to the grammar for T [S,R]. A program is M_0 [$M_{1/2}$,M_1] treeless if all the right hand sides in the program are M_0 [$M_{1/2}$,M_1] treeless. □

Treeless terms and programs appear trivial. This shows how powerful the positive supercompiler is:

PROPOSITION **5.1.4** *(i) Let t, p be an M_0 [$M_{1/2}$,M_1] term and program and t', p' the output term and program from $\mathcal{W}$ excluding postunfolding. Then t', p' are M_0 [$M_{1/2}$,M_1] treeless. (ii) This also holds after postunfolding.* □

PROOF: (*i*) By inspection of the right hand sides of $\Rightarrow$ in $\mathcal{W}$. Note that let-expressions are only introduced when $\mathcal{W}$ is applied to let-expressions, and conditionals are only introduced when $\mathcal{W}$ is applied to conditionals.

(*ii*) Note that the classes are closed under the operation of unfolding a call to a treeless f-function. □

A similar result was stated for deforestation in [Wad88].

COROLLARY **5.1.5** *If $\mathcal{W}$ is applied to an $M_{1/2}$ treeless term and program then the resulting term and program contain no nested function calls.* □

PROOF: Immediate. □

Construction of intermediate data structures in the modular form treated in the present Chapter only occurs in the presence of nested function calls $g\ (f\ x)$ where f produces and g consumes. So, if we start out with an $M_{1/2}$ term and program, then $\mathcal{W}$ returns a term and program which do not construct *any* intermediate data structures. The sublanguage $M_{1/2}$ is particular interesting because the local definitions of M_1 are present only to facilitate termination of $\mathcal{W}$, see Section 3.3.

It may also be noted that $M_{1/2}$ treeless terms and programs are *order-of-evaluation independent*. The difference between call-by-value and call-by-name evaluation is only manifest on programs with nested calls, but $M_{1/2}$ treeless terms and programs do not contain nested calls.

5.2 Program specialization

Given that we have a function f of two parameters x_1, x_2, and one known argument d_1 for x_1 we can construct a function of just the argument x_2. If we want to call f many times with d_1, it may be worthwhile to construct the specialized function once and for all, and then use that instead of the general function. This is because the specialized can be made efficient by computing at specialization time whatever computations that depend only on the argument x_1 and constants in the program.

A more complicated situation arises when an argument is not simply *known* or *unknown*, but possibly *partly known i.e. a partially static structure* where it may still be the case that certain computations can be performed at transformation time.

Program specialization is discussed in most works on supercompilation, where it is usually stated that supercompilation can achieve more than program specialization, see the quotation to Chapter 9.

The following example shows how supercompilation elegantly handles partly known values.

EXAMPLE **5.2.1** Recall the append function from Example 5.1.1, and suppose that we want to append a list that starts with the elements One, Two with another one starting with the element $Three$.

$$\begin{array}{lcl} & & a\ (Cons\ One(Cons\ Two\ y))\ (Cons\ Three\ z) \\ a\ Nil\ ys & \leftarrow & ys \\ a\ (Cons\ x\ xs)\ ys & \leftarrow & Cons\ x\ (a\ xs\ ys) \end{array}$$

The positive supercompiler turns this program and term into the following.

$$\begin{array}{lcl} & & Cons\ One\ (Cons\ Two\ (f\ y\ z)) \\ f\ Nil\ z & \leftarrow & Cons\ Three\ z \\ f\ (Cons\ x\ xs)\ z & \leftarrow & Cons\ x\ (f\ xs\ z) \end{array}$$

Here we could compute some of the steps of a resulting from the known parts of the first argument. □

The computations above are done by instantiation and unfolding in clause (3b). In the generation of a residual function from a term in clauses (2),(3a),(3b) the function becomes a function only of the free variables of the term. The known parts are propagated to the body of the residual function.

As another example we consider *Ackermann's function.*

EXAMPLE **5.2.2** Consider Ackermann's function specialized to known first argument. The formulation here uses a local definition to ensure termination of $\mathcal{W}$. Local definitions to ensure terminatiom of $\mathcal{W}$, including the one below, can be calculated automatically. We return to this in part III.

$$\begin{array}{lcl} f\ n & \leftarrow & ack\ (Succ\ (Succ\ Zero\ ))\ n \\ ack\ Zero\ n & \leftarrow & Succ\ n \\ ack\ (Succ\ m)\ Zero & \leftarrow & ack\ m\ (Succ\ Zero) \\ ack\ (Succ\ m)\ (Succ\ n) & \leftarrow & \text{let } v = ack\ (Succ\ m)\ n \text{ in } ack\ m\ v \end{array}$$

Specialization of the term $f\ n$ yields the term $f'\ n$ and the following faster program:

$$\begin{array}{lcl} f'\ n & \leftarrow & ack_2\ n \\ ack_2\ 0 & \leftarrow & \text{let } v = Succ\ Succ\ ZeroSucc\ v \text{ in} \\ ack_2\ (Succ\ n) & \leftarrow & \text{let } v = ack_2\ n \text{ in } ack_1\ v \\ ack_1\ 0 & \leftarrow & Succ\ (Succc\ Zero) \\ ack_1\ (Succ\ n) & \leftarrow & \text{let } v = ack_1\ n \text{ in } ack_0\ v \\ ack_0\ n & \leftarrow & Succ\ n \end{array}$$

□

We close the section with some remarks concerning the achievement of specialization by $\mathcal{W}$.

COROLLARY **5.2.3** *If $\mathcal{W}$ is applied to an $M_{1/2}$ treeless term and program then the resulting term and program contain no calls with arguments containing constructors.* □

PROOF: Immediate from Proposition 5.1.4. □

So if we start out with an $M_{1/2}$ term and program, the resulting term and program are *completely* specialized to any partly or wholy known constants.

It is also easy to see that in any residual conditionals, the test has form $b = b'$, where at least one of b, b' is non-ground. However, it may be the case that the test has form *e.g.* $C\ u = C\ v$. Such a test can obviously be replaced by the simpler test $u = v$. It might be tempting to generally replace the test $b = b'$ by the test $C\,v_1 \ldots v_n = C\,t_1 \ldots t_n$, where $\{v_i := t_i\}_{i=1}^{n} = MGU(b, b')$, and where C is used to implement compund tests. This is not done in $\mathcal{W}$, because this transformation does not preserve semantics, see Chapter 10 and the remark at the last subsection of Section 11.2.

5.3 Theorem proving

By Theorem Proving we have in mind the kind of tasks that are undertaken more or less automtically in systems such as HOL [Gor93] or Nuprl [Con86]. For instance we can imagine a representation of Peano Arithmetic, see *e.g.* [Men87], and then ask for proofs or refutations of statemens like $x + y = y + x$.

Although it is well-known that no such general decision procedure can exist, one can still go some of the way; *automatic* theorem proving is an active field of research.

In the context of supercompilation, theorem proving was considered mainly in [Tur80a,Tur80c], as described below.

EXAMPLE **5.3.1** The following program represents some of the axioms for equality and addition in Peano arithmetic.

$$\begin{array}{lcl} e\ Z\ Z & \leftarrow & True \\ e\ (S\ x)\ (S\ y) & \leftarrow & e\ x\ y \\ e\ x\ y & \leftarrow & False \\ a\ x\ Z & \leftarrow & x \\ a\ x\ (S\ y) & \leftarrow & S\ (a\ x\ y) \end{array}$$

Here we have assumed the possibility of having an *otherwise* clause, *i.e.* a g-function with a "catch-all" left hand side $e\ x\ y$. The positive supercompiler propagates no information to this branch.

The theorem $0 + x = x$ is proved by induction in Peano Arithmetic, see [Men87] p119. To prove it using $\mathcal{W}$ we can apply $\mathcal{W}$ to the term $e\ (a\ Z\ x)\ x$ and check that the result is the function that always returns $True$. In fact, the term and function generated is

$$\begin{array}{lcl} & & c\ x \\ c\ Z & \leftarrow & True \\ c\ (S\ z) & \leftarrow & c\ z \end{array}$$

Of course, to give the function $c\ x \leftarrow True$, the supercompiler must be adapted to recognize this situation and perform suitable transformation, but this holds equally for Turchin's supercompiler. □

In the example, the transformation did a case analysis on x, *i.e.* an instantiation by clause (3b) of a's argument. In the base case, $x \equiv Zero$, $\mathcal{W}$ unfolded by clause (3b) and discovered that the base case pulled through, yielding $c\ Z \leftarrow True$. In the induction step, $x \equiv s\ Z$, $\mathcal{W}$ unfolded by clause (3b), and ended up with the original term, looping back. This corresponds to an application of the induction hypothesis.

So again, the important feature of $\mathcal{W}$ is the instantiation and unfolding in clause (3b). It should be noted that the example omly works because *all* occurrences of the variable in the redex position, *viz.* x, are instantiated. Alternatively to transforming $e\ (a\ Zx)\ x$ and recognizing that the resulting term always returns $True$, we could have transformed the term $a\ Z\ x$ into

$$\begin{array}{lcl} & & c'\ x \\ c'\ Z & \leftarrow & Z \\ c'\ (S\ y) & \leftarrow & S\ (c'\ y) \end{array}$$

and recognized that the term $c'\ x$ always returns x. In the latter case we no longer rely on more than one occurrence of x being instantiated in clause (3b) of $\mathcal{W}$.

It is very interesting to note that the supercompiler seems to be able to perform only proofs using *one* induction loop; for instance it cannot prove $x + y = y + x$ which requires two induction loops. However, by self-applying the supercompiler such a proof can be obtained. More generally, it would seem that a proof requiring k induction loops can be achieved by k self-applications. This idea is pursued further in [Tur80a,Tur80c]; see also the quotation to the present chapter.

Chapter 6

Programming Systems and Compilers

> The problem reminds that of constructing a flying machine:
> it must have an engine powerful enough to raise it in the air,
> but trying to increase its power we also increase its weight.
>
> *Valentin F. Turchin.*[1]

In this chapter we describe the the role of supercompilation in compilation, compiler generation, and compiler generator generation. Historically speaking, this was the most important motivation for the whole supercompiler project, see Sections 1.1 and 1.2. The history of the theoretical and practical development of self-applicable partial evaluation and supercompilation was described in detail in Sections 1.2 and 1.3 and is not repeated here.

Section 6.1 describes the programs=specifications paradigm. Section 6.2 describes the Futamura projections as one finds them in the paper [Jon85] and in subsequent papers in the Copenhagen tradition. The Futamura projections show how one compiles and generates compilers and compiler-generators from specializers and interpreters. Section 6.3 gives some intuition on the Futamura projections in λ-calculus terms.

Section 6.4 describes metasystem transition in a style similar to how one finds it in Turchin's and Glück's papers on supercompilation. This section also discusses the problem of encoding terms as values which is neglected in the preceding sections, and it is argued that the main difference between the notation for metasystem transition and self-application in partial evaluation is that the former is explicitly concerned with encoding.

Section 6.5 describes how one can extract the compiler part and interpreter part of a specializer by metasystem transition. This is not the same as the Futamura projections, where one uses both specializers and interpreters to obtain compilers. Section 6.6 shows how one extracts specializers from specializers.

Section 6.7 describes the Futamura projections as special cases of metasystem transition. Section 6.8 describes Glück's specializer projections which are also special cases of metasystem transition. The relationship between the Futamura projections and the specializer projections is, in a certain sense, the same as the one between compiler extraction and specializer extraction.

The material in Sections 6.4 to 6.8 is from [Glu91a,Glu91b,Glu94a].

Section 6.9 finally relates the postive supercompiler to all these results.

[1] In [Tur86a].

6.1 Programs=specifications

It is an old dream that the programmer should not describe *how* to achieve a task, but only *what* the task is. For instance, logic programming languages and functional programming languages are sometimes motivated by this desire; they are claimed to be *declarative* rather than *procedural.*

The full extent of the dream is reached when, given a large-scale problem, the user introduces a language, or a hierarchy of languages, for writing specifications that solve the problem, in as simple and natural a way as we today write a hierachy of procedures.

How does one introduce a new language S? As is well known there are primarily two ways of doing this. One can write an S-interpreter in a language T which can already be run on the computer, or one can write an S to T-compiler. The first is the easiest, but the second gives the shortest running times on the computer.

The purpose of a compiler-generator is to transform automatically a description of the new language given by the user into a compiler. Such compiler-generators have been devised since the late 1970's. For instance, Mosses' SIS system turns a denotational semantics into a compiler [Mos79]. The description can take various forms such as denotational, axiomatic, or operational semantics, but all can be viewed as interpreters. For instance, a denotational semantics is, very roughly, an interpreter written in Scheme [Ree86]. A compiler generator, then, is a program that turns interpreters into compilers.

So compiler generators allow the user in the programs=specifications paradigm to have the best of both interpreters and compilers: the new languages for writing application specific specifications are introduced by interpreters, which are easy to write, and which are turned automatically into compilers that can generate efficient code.

One of the most important and most frequently cited applications of partial evaluation is compilation, compiler generation, and compiler generator generation. For instance, this role and its connections to the specifications=programs paradigm, is discussed by Futamura [Fut71], Ershov [Ers78] Jones *et al.* [Jon85], and Turchin [Tur79,Tur80b]. As mentioned in Section 1.3, actual compilers and compilers have been generated, but the first ones were for small toy languages only. More recently, automatic generation of compilers by partial evaluation for realistic languages producing programs comparable in efficiency with the output of commercial compilers have appeared [Jor92a].

The next section shows technically how specializers can produce compilers from interpreters.

6.2 The Futamura Projections in Partial Evaluation

The *Futamura projections*, independently due to Futamura, Ershov, Turchin, and Sandewall *et al.* [Fut71, Tur80b,Ers77,Bec75] state that given a partial evaluator with L as implementation, input and output language[2] and a S-interpreter written in L, one can (*i*) compile S programs to L programs; (*ii*) generate an S to L compiler, and (*iii*) generate a compiler generator, such that the generated compilers translate to L. Below we review these results in a style similar to that in *e.g.* [Jon85].

DEFINITION **6.2.1** We assume a fixed set D whose elements are programs in various languages, as well as their input and output. D is assumed closed under list formation $[d_1 \ldots d_n]$. For a programming language L let $\mathtt{L}\,p\,d$ denote the semantics of the application of the L-program p to the value d. Functions of several arguments are encoded as functions of one list argument. We assume some notion of semantic equivalence, $=$. The following are the *interpreter, compiler, and specializer equations.*

An S-*interpreter* (in L) is an L-program i such that for all S-programs p

$$\mathtt{S}\,p\,[d_1 \ldots d_n] = \mathtt{L}\,i\,[p, d_1 \ldots d_n]$$

An S *to* T-*compiler* (in L) is an L-program c such that for all S-programs p

$$\mathtt{S}\,p\,[d_1 \ldots d_n] = \mathtt{T}\,(\mathtt{L}\,c\,p)\,[d_1 \ldots d_n]$$

[2] The requirement on the output language can be relaxed as we shall see.

An S *to* T*-specializer* (in L) is an L-program m such that for all S-programs p

$$\mathtt{S}\,p\,[d_1 \ldots d_{n+k}] = \mathtt{T}\,(\mathtt{L}\,m\,[p, d_1 \ldots d_n])\,[d_{n+1} \ldots d_{n+k}]$$

An L to T-specializer in L is called *self-applicable.* □

REMARK **6.2.2** What we have called the specializer equation is often called the *mix equation* for historical reasons, see Section 1.3. □

PROPOSITION **6.2.3** *(Futamura Projections.) Let m be an R to T-specializer in L, i an S-interpreter in R, p an S-program, $[d_1, \ldots, d_n]$ its input.*

1. *Let $target = \mathtt{L}\,m\,[i, p]$. Then $\mathtt{T}\,target\,[d_1 \ldots d_n] = \mathtt{S}\,p\,[d_1 \ldots d_n]$.*
2. *Assume $R = L$, and let $comp = \mathtt{L}\,m\,[m, i]$. Then $\mathtt{T}\,comp\,[p] = target$.*
3. *Assume again $R = L$, and let $cogen = \mathtt{L}\,m\,[m, m]$. Then $\mathtt{T}\,cogen\,[i] = comp$*

□

PROOF: Easy. □

The first equation shows that given an R to T-specializer in L, an S interpreter in R, and an S-program, one can get a T program. So, one can compile a new language S to the output language of the specializer, provided that an interpreter for S is given in the input language of the specializer. Even if $R = T = L$, one can compile a new language S to L by writing an S-interpreter in L.

The second equation shows that if $R = L$, then one can generate an S to T-compiler in T, provided that an S-interpreter in L is given. Even if $R = T = L$, one can get a compiler for a new language S to L in L by writing an S-interpreter in L.

The third equation shows that if $R = L$, then one can generate a compiler generator in T which given an S-interpreter in L produces an S to T-compiler in T. If $R = T = L$, one can get a compiler generator in L that given an S-interpreter in L produces an S to L-compiler in L.

We note in conclusion that the Futamura projections are concerned with compilation, compiler generation, and compiler generator generation *from interpreters.*

6.3 A simple intuition on the Futamura projections

The Futamura equations may not be easy to grasp at first.[3] As is well-known, in the $s-m-n$ Theorem the construction of $f_{d_1 \ldots d_n}$ from f merely "assigns" $d_1 \ldots d_n$ to the variables of f; no computation is performed. As such, the theoretical essence of program specialization is the *change in functionality.*

We now use this idea in an informal λ-calculus setting to give a simple intuition on the Futamura projections. The reader not familiar with the λ-calculus may consult any simple introduction.It should be noted that below, like in the preceding section, we ignore the issue of encoding.

Let D be the language of λ-expressions, assuming that lists are encoded somehow, and $L = R = T$ the λ-calculus, with $\mathtt{L}\,p\,d$ denoting the partial function returning the β-normal form of $p\,d$, if any. S is some other language. We assume that functions f expect two arguments coded as $[x_1, x_2]$, and a specializer must produce a function of one argument, x_2. Then

$$m_\lambda \equiv \lambda[f, x_1].\lambda x_2.f\,[x_1, x_2]$$

is in fact a specializer, for $m\,[p, d_1] \equiv (\lambda[f, x_1].\lambda x_2.f\,[x_1, x_2])\,[p, d_1] \rightarrow_\beta \lambda x_2.p\,[d_1, x_2]$, and clearly applying the latter term to d_2 yields the same normal form, if any, as that of $p\,[d_1, d_2]$.

The definition of m_λ reveals very clearly that it is the change in functionality that makes it a specializer: it expects a function f and a value x_1 and returns something, a residual program, which when supplied the value x_2 will produce the output.

[3] To quote [Gom90]:"Although easy to verify, it must be admitted that the intuitive significance of these equations is hard to see."

Now we can explain the Futamura equations. First, $target = m_\lambda\,[i,p]$ is a translation of the S-program p into λ-calculus since $target \rightarrow_\beta \lambda x_2.i\,[p,x_2]$, *i.e.* $target$ is something which when supplied input will return the result, *viz.* the application of the S-interpreter to the S-program and its data. Second, $comp = m_\lambda\,[m_\lambda, i]$ is a compiler since $comp \rightarrow_\beta \lambda x_2.m_\lambda\,[i,x_2]$, *i.e.* $comp$ is something which waits for a program p and then yields $m_\lambda\,[i,p]$ ($\rightarrow_\beta \lambda z.i\,[p,z]$) which we have just seen to be a translation of x_2. Finally, $cogen = m_\lambda\,[m_\lambda, m_\lambda]$ is a compiler generator since $cogen \rightarrow_\beta \lambda x_2.m_\lambda\,[m_\lambda, x_2]$, *i.e.* $cogen$ is something waiting for an interpreter i and then yielding

$$m_\lambda\,[m_\lambda, i]\ \ (\rightarrow_\beta \lambda p.\lambda z.i\,[p,z])$$

which is known to be a compiler.

6.4 Metasystem transition

In this section we review metasystem transition in a style similar to that in [Glu91a]. The reader familiar with that paper may note that we have avoided the notion of *demetacoding.*

Recall how the postive supercompiler works. It takes a term t, possibly containing free variables, and a program p and returns a new term t', possibly containing free variables, and a new program p'. Now, for self-applcable specializers, the input language is the same as the language in which the specializer is implemented. Let us imagine this is Scheme, to make the discussion more concrete. So the specializer, say α, is written in Scheme and is applied to a Scheme term t. But an application $(\alpha\, t)$ where t contains free variables, does not make sense in Scheme; t must be a Scheme *value.* This means that terms and programs must be *encoded* as Scheme values. Actually, the main pleasent feature of Scheme is the ease with which terms are encoded as values: one can encode t as $'t$.

We now return to the general setting.

DEFINITION **6.4.1** We assume sets V, D, T. All programs are elements in T, all input and output values for all languages are elements in D, and V is a set of *metavariables.* We assume a function $\underline{\bullet}$ mapping elements from $V \cup D \cup T$ to D. We retain the remaining assumptions concerning D from Definition 6.2.1.

The *principle of metasystem transition*, is then given by the following definition. An *S to T-specializer* (in L) is an L-program m such that for all S-programs p, values d_i, c_j, metavariables x_i:[4]

$$\mathtt{S}\,p\,[c_1 \ldots c_n, d_1 \ldots d_k] = \mathtt{T}\,(\mathtt{L}\,m\,[\underline{p}, \underline{c_1} \ldots \underline{c_n}, \underline{x_1} \ldots \underline{x_k}])\,[d_1 \ldots d_k]$$

An L to T-specializer in L is called *self-applicable.* □

Note that the only difference between the mix equation and principle of metasystem transition, as it is stated here, is that only the latter is explicitly concerned with encoding.

The following example shows *repeated metasystem transition.*

EXAMPLE **6.4.2** Assume that m is an L to L specializer in L. Then

$$\begin{aligned} \mathtt{L}\,p\,[d_1, d_2, d_3] &= \mathtt{L}\,(\mathtt{L}\,m\,[\underline{p}, \underline{d_1}, \underline{d_2}, \underline{x_3}])\,[d_3] \\ &= \mathtt{L}\,(\mathtt{L}\,(\mathtt{L}\,m\,[\underline{m}, \underline{\underline{p}}, \underline{\underline{d_1}}, \underline{x_2}, \underline{\underline{x_3}}])\,[\underline{d_2}])\,[d_3] \end{aligned}$$

Note that the depth of the encoding determines which version of m a metavariable refers to. The variable x_3 is encoded twice and hence refers to the m (and value d_3) two levels further out, whereas x_2 is only encoded once and hence refers to the first surrounding m (and value d_2.) □

Note that in the example, the specializer m becomes encoded, and the program p becomes encoded twice. Taking another metasystem transition means that m also becomes encoded twice. This means that if one is not careful in the choice of encoding, self-applicable specialization can take up vast amounts of space and time. There are solutions to this problem. One can avoid some of the encoding and choose an encoding which is cheap on objects in D. Since encoded things are objects in D, the double encoding is almost as cheap as a single encoding, see [Lau91].

[4] There is a problem with this notation, namely that one can only "pull out' a *postfix* of the arguments, in the notation: $d_1 \ldots d_k$. Below we shall need to pull out other arguments than a prefix. Then it must be understood, somehow, which pulled out arguments correspond to which metavariables. In view of this, the author admits that the notation invented in the present section is not so great after all.

6.5 Compiler and interpreter extraction

As mentioned, the difference between the notation in the mix equation and in the principle of metasystem transition is that the required encoding is explicit in the latter case. This notation may seem more cumbersome, but it also has its good sides.

A good property of the notation is that it allows us to say that we would like to specialize a function to, say, known first argument and unknown second and third argument, but such that the second and third argument are the same. This is done by $\mathtt{L}\, m\, [\underline{p}, \underline{d}, \underline{x_1}, \underline{x_1}]$.

Another good property is that the relation between interpretation, compilation, and specialization is much clearer here, as we now explain.

It is well-known that a specializer must contain both an interpreter and a compiler: if all arguments are known everywhere in the program the specializer should be able to compute everything, just like an interpreter, and if no arguments are known the specializer must generate a residual program, just like a compiler. Notice how well this fits with the situations $k = 0$ and $n = 0$, respectively, in the principle of metasystem transition. For $k = 0$, all arguments to p are given, and m computes a T-constant (or a T-function of 0 variables.) For $n = 0$, none of the k arguments are given, and m yields a T program of k arguments, *i.e.* a compiled version of p. Notice that this is not the same way of compilation as that in the first Futamura projection.

Below we make this idea systematic for the case of compilation. Similar results hold for interpretation. First let us make clear what interpreters and compilers now are:

Definition **6.5.1** An *S-interpreter* (in L) is an L-program i such that for all S-programs p

$$\mathtt{S}\, p\, [d_1 \ldots d_n] = \mathtt{L}\, i\, [\underline{p}, \underline{d_1} \ldots \underline{d_n}]$$

An *S to T-compiler* (in L) is an L-program c such that for all S-programs p

$$\mathtt{S}\, p\, [d_1 \ldots d_n] = \mathtt{T}\, (\mathtt{L}\, c\, \underline{p})\, [d_1 \ldots d_n]$$

□

Below we shall for $d = [d_1 \ldots d_n]$ let $[a_1, \ldots, a_k; d]$ denote $[a_1, \ldots a_k, d_1 \ldots d_n]$. Also, $[a_1, \ldots, a_k; \underline{d}]$ denotes $[a_1, \ldots a_k, \underline{d_1} \ldots \underline{d_n}]$, and similarly with multiple encodings.

Glück calls the following proposition *compiler extraction* because it shows how one can extract the compiler part of a specializer. As mentioned, there are similar *interpreter extraction* results.

Proposition **6.5.2** *(Compiler extraction.) Let m be an S to T-specializer in L, p an S-program, $d = [d_1, \ldots d_n]$ its input, and $x_d = [x_1 \ldots x_n]$.*

1. *Let target* $= \mathtt{L}\, m\, [\underline{p}; \underline{x_d}]$. *Then* $\mathtt{T}$ *target* $[d_1 \ldots d_n] = \mathtt{S}\, p\, [d_1 \ldots d_n]$.
2. *Let comp* $= \mathtt{L}\, m\, [\underline{m}, \underline{x_p}; \underline{\underline{x_d}}]$. *Then* $\mathtt{T}$ *comp* $[\underline{p}] =$ *target.*
3. *Let cogen* $= \mathtt{L}\, m\, [\underline{m}, \underline{x_m}, \underline{\underline{x_p}}; \underline{\underline{\underline{x_d}}}]$. *Then* $\mathtt{T}$ *cogen* $[\underline{m}] =$ *comp*

□

Proof: Easy. □

The first equation shows that given an S to T-specializer in L one can get a T program. So, one can compile to the output language of the specializer, but only programs in S, not for arbitrary new languages. The second equation shows that we can construct an L to T compiler, and the third that we can get L to T compiler-generators, both constructions provided that $S = L$.

Remark **6.5.3** If in the second equation the unencoded m is an L to T-specializer in R, and the encoded m is an S to K-specializer in R, the result is an S to K-compiler in T. If in the third equation, the unencoded m is an R to T-specializer in L, the encoded m is an S to K-specializer in R, and the variable x_m ranges over U to V-specializers in S, then the result is a compiler-generator in T which when applied to a U to V-specializer in S will produce a U to V-compiler in K. So we can get compilers for other languages than L, but only at the expense of writing new specializers, which of course is more complicated than writing interpreters. □

6.6 Specializer extraction

The preceding section showed how interpretation and compilation can be viewed as special cases of specialization. The section also showed how one could compile, generate a compiler, and generate a compiler generator by self-application of a specializer, and suggested that one could similarly interpret, generate an interpreter, and generate an interpreter generator by self-application of the specializer. Given that interpretation and compilation are special cases of specialization one would imagine that these two types of self-application are special cases of a more general type of self-application whereby one can specialize, generate specializers, and generate specializer generators.

Indeed, this is the case. In this section we review the general format of which the equations in the preceding section as well as the similar equations for interpreation are special cases.

PROPOSITION **6.6.1** *(Specializer extraction.) Let* m *be an* S *to* T*-specializer in* L*,* p *an* S*-program,* $[c_1 \ldots c_n, d_1 \ldots d_k]$ *its input,* $x_c = [y_1 \ldots y_n]$*,* $x_d = [x_1 \ldots x_k]$*.*

1. *Let* $resid = \mathtt{L}\, m\, [\underline{p}, \underline{c_1} \ldots \underline{c_n}; \underline{x_d}]$*. Then* $\mathtt{T}\, resid\, [d_1 \ldots d_n] = \mathtt{S}\, p\, [c_1 \ldots c_n, d_1 \ldots d_k]$*.*
2. *Let* $spec = \mathtt{L}\, m\, [\underline{m}, \underline{x_p}; \underline{x_c}; \underline{\underline{x_d}}]$*. Then* $\mathtt{T}\, spec\, [\underline{p}, \underline{c_1} \ldots \underline{c_n}] = resid$*.*
3. *Assume* $S = L$*. Let* $spgen = \mathtt{L}\, m\, [\underline{m}, \underline{x_m}, \underline{\underline{x_p}}; \underline{\underline{x_c}}; \underline{\underline{\underline{x_d}}}]$*. Then* $\mathtt{T}\, spgen\, [\underline{m}] = spec$

□

PROOF: Easy. □

So we can specialize, generate specializers, and generate specializer generators by self-application. We can also combine specializers for different languages in a way similar to the one described in Remark 6.5.3.

6.7 Futamura projections by metasystem transition

We arrive at the Futamura projection from the compiler extraction equations by inserting an interpreter between the specializer and the program.

PROPOSITION **6.7.1** *(Futamura Projections by metasystem transition.) Let* m *be an* R *to* T*-specializer in* L*,* i *an* S*-interpreter in* R*,* p *an* L*-program,* $d = [d_1 \ldots d_n]$ *its input,* $x_d = [x_1 \ldots x_n]$*.*

1. *Let* $target = \mathtt{L}\, m\, [\underline{i}, \underline{p}; \underline{x_d}]$*. Then* $\mathtt{T}\, target\, [d_1 \ldots d_n] = \mathtt{S}\, p\, [d_1 \ldots d_n]$*.*
2. *Let* $comp = \mathtt{L}\, m\, [\underline{m}, \underline{\underline{i}}, \underline{x_p}; \underline{\underline{x_d}}]$*. Then* $\mathtt{T}\, comp\, [\underline{p}] = target$*.*
3. *Let* $cogen = \mathtt{L}\, m\, [\underline{m}, \underline{\underline{m}}, \underline{x_i}, \underline{\underline{x_p}}; \underline{\underline{\underline{x_d}}}]$*. Then* $\mathtt{T}\, cogen\, [\underline{\underline{i}}] = comp$

□

PROOF: Easy. □

These equations have the same significance as the versions of the Futamura equations explained in Section 6.2.

6.8 Specializer projections

The main difference between compiler extraction and specializer extraction is that in the latter case the specializer receives not only the program to specialize, but also some of its arguments. Note that in the Futamura projections, the specializer does not receive any of the program's arguments. By letting it have have some of the program's arguments we get from the Futamura projections to the specializer projections.

PROPOSITION **6.8.1** *Let m be an R to T-specializer in L, i an S-interpreter in R, p an L-program, $[c_1 \ldots c_n, d_1 \ldots d_k]$ its input, $x_c = [y_1 \ldots y_n]$, $x_d = [x_1 \ldots x_k]$.*

1. *Let* $resid = \mathtt{L}\, m\, [\underline{i}, \underline{p}, \underline{c_1} \ldots \underline{c_n}; \underline{x_d}]$. *Then* $\mathtt{T}\, resid\, [d_1 \ldots d_k] = \mathtt{S}\, p\, [c_1 \ldots c_n, d_1 \ldots d_k]$.
2. *Let* $spec = \mathtt{L}\, m\, [\underline{m}, \underline{\underline{i}}, \underline{x_p}; \underline{x_c}; \underline{\underline{x_d}}]$. *Then* $\mathtt{T}\, spec\, [\underline{p}] = resid$.
3. *Let* $spgen = \mathtt{L}\, m\, [\underline{m}, \underline{\underline{m}}, \underline{x_i}, \underline{\underline{x_p}}; \underline{\underline{x_c}}; \underline{\underline{\underline{x_d}}}]$. *Then* $\mathtt{T}\, spgen\, [\underline{i}] = spec$

□

PROOF: Easy. □

The first equation shows that given an R to T-specializer in L, and S interpreter in R and an S-program, one can get a residual T program. So, one can specialize programs in a new language S to the output language of the specializer, provided that an interpreter for the S is given in the input language of the specializer. Even if $R = T = L$, one can specialize a new language S to L by writing an S-interpreter in L.

The second equation shows that if $R = L$, then one can generate an S to T-specializer in T, provided that an S-interpreter in L is given. Even if $R = T = L$, one can get a specializer for a new language S to L in L by writing an S-interpreter in L.

The third equation shows that if $R = L$, then one can generate a specializer generator in T that given an S-interpreter in L produces an S to T-specializer in T. If $R = T = L$, one can get a specializer-generator in L that given an S-interpreter in L produces an S to L-specializer in L.

Even in the case where the interpreter is meta-circular, *i.e.* is an L-interpreter in L, the specializer equations are important. The point is that the interpreter can manipulate information which by the specialization process becomes inlined in the original specializer, so that the generated specializer is more powerful than the original. For instance, Glück and Jørgensen have generated supercompilers from partial evaluators, see [Glu94a].

It should be noted that in terms of the principle of metasystem transition, the difference between the Futamura projections and the specializer projections is apparent in the encoding. In the notation from Section 6.2, this difference is not apparent. In fact, the second and third specializer projection would in that notation look exactly like the second and third Futamura projection.

We have now described four types of equations: (*i*) compiler extraction, (*ii*) specializer extraction, (*iii*) Futamura projections, and (*iv*) specializer projections.

The two first, the extractions, are similar in that they do not have an interpreter between the specializer and the program, whereas the two latter, the projections, are similar in that they have an interpreter between the specializer and the program.

We can also group the equations along another axis. The compiler extractions and the Futamura projections are similar in that the program in both cases does not receive any of its arguments, whereas the specializer extractions and specializer projections are similar in that the program in both cases receives some of its arguments.

We close this section with the hope of having conveyed the impression that there is more to self-application of specializers than the Futamura projections.

6.9 Can $\mathcal{W}$ generate compiler-generators?

There are three properties a transformer must satisfy to be able to compile and generate compilers and compiler generators.

First of all, it must perform the change in functionality of its object programs that we saw in Sections 6.2 and 6.3. Given a function f of 2 variables, $\mathcal{W}$ applied to the term $f\ b\ v$, where b is a constant and v is a variable, yields a term $f'\ v$ with one free variable. So we have transformed f of two variables into f' of one variable, and so $\mathcal{W}$ does change the functionality as required.

Second, it must be written in its own input language to be self-applicable. At the present $\mathcal{W}$ is written in an informal metalanguage but might of course be written in M_1.

Thirdly, an actual implementation of self-application must solve practical problems concerning termination of the transformer, and size and efficency of the residual programs. Such problems for $\mathcal{W}$ are beyond discussion at the present point since we have not even formulated $\mathcal{W}$ in M_1. However it should be noted that problems are likely to arise, not least because $\mathcal{W}$ is stronger than partial evaluation, see Section 9.2 and the quotation to the present chapter.

In conclusion, if $\mathcal{W}$ is formulated in M_1, then it can, *in principle*, compile and generate compilers and compiler generators.

Chapter 7

Logic Programming by Positive Supercompilation

The concept of an algorithm, a process,
remains at the basis of computer science,
and it makes no sense to dress it up,
mandatorily, as a system of relations.

Valentin F. Turchin.[1]

This chapter attempts to show that one can use the positive supercompiler, mainly the driving part, for logic programming and problem solving.[2] By logic programming we mean the interpretation of logic programs. By problem solving we mean the solving in logic programming of such tasks as natural language processing and deductive database management, see the textbooks [Amb87,Bra86].

Section 7.1 gives a simple example of problem solving. Section 7.2 shows that it can be performed by the positive supercompiler. Section 7.3 develops a general, precise correspondence between Prolog interpretation and positive supercompilation in terms of transformation trees and so-called SLD-trees. Section 7.4 reviews previous relations between supercompilation and problem solving.

7.1 Logic programming

Basic logic programming notions such as *term, atom, clause, definite clause, empty clause etc.* are assumed familiar, see [Llo87].

The following simple example and its continuation in the next section is due to Glück.

EXAMPLE **7.1.1** Consider the Prolog predicate *connect*.[3]

connect(x,y,[]):-flight(x,y).
connect(x,y,[z|zs]):-flight(x,z),connect(z,y,zs).

flight(Vienna,Paris).
flight(Vienna,Rome).
flight(Rome,Paris).
flight(Paris,London).
flight(Paris,Copenhagen).

[1] In [Tur86a].

[2] The ideas of this chapter have at the time of printing been developed further; see the preface to the revised edition.

[3] We use upper case letters for constructors (in logic programming terms: *functors*) and lower case letters for variables. In logic programming, the convention is usually the opposite.

The query *connect(z,y,zs)* is true if there are flights (x_1, x_2), (x_2, x_3) ... (x_{n-2}, x_{n-1}), (x_{n-1}, x_n) such that x is x_1, y is x_n and zs is $[x_2, \ldots x_{n-1}]$. For instance,

?-connect(Vienna,Copenhagen,via).

via=[Paris];
via=[Rome,Paris];
No more solutions

□

7.2 Logic programming by driving

We shall translate the Prolog program of the preceding section into an M_1 program so that the supercompiler, in a certain sense, simulates the behaviour of the Prolog interpreter. Since there are no free variables in the body of any predicates, it is straight-forward to turn the Prolog program into a functional program:

$$
\begin{array}{lcl}
connect\ x\ y\ Nil & \leftarrow & flight\ x\ y \\
connect\ x\ y\ (Cons\ z\ zs) & \leftarrow & flight\ x\ z = True\ \rightarrow\ connect\ z\ y\ zs\ \square\ False \\
\\
flight\ Vienna\ Paris & \leftarrow & True \\
flight\ Vienna\ Rome & \leftarrow & True \\
flight\ Rome\ Paris & \leftarrow & True \\
flight\ Paris\ London & \leftarrow & True \\
flight\ Paris\ Copenhagen & \leftarrow & True \\
flight\ x\ y & \leftarrow & False
\end{array}
$$

If we run $\mathcal{W}$ with the term *connect Vienna Copenhagen via* we get the following term and program.

$$c^1\ via$$

$$
\begin{array}{lcl}
c^1\ Nil & \leftarrow & False \\
c^1\ (Cons\ z\ zs) & \leftarrow & f^1\ z\ zs \\
f^1\ Paris\ zs & \leftarrow & c^2\ zs \\
f^1\ Rome\ zs & \leftarrow & c^3\ zs \\
f^1\ x\ zs & \leftarrow & False \\
\\
c^2\ Nil & \leftarrow & True \\
c^2\ (Cons\ z\ zs) & \leftarrow & f^2\ z\ zs \\
f^2\ London\ zs & \leftarrow & c^4\ zs \\
f^2\ Copenhagen\ zs & \leftarrow & c^5\ zs \\
f^2\ x\ zs & \leftarrow & False \\
\\
c^3\ Nil & \leftarrow & False \\
c^3\ (Cons\ z\ zs) & \leftarrow & f^3\ z\ zs \\
f^3\ Paris\ zs & \leftarrow & c^2\ zs \\
f^3\ x\ zs & \leftarrow & False \\
\\
c^4\ Nil & \leftarrow & False \\
c^4\ (Cons\ z\ zs) & \leftarrow & f^4\ z\ zs \\
f^4\ x\ zs & \leftarrow & False \\
\\
c^5\ Nil & \leftarrow & False \\
c^5\ (Cons\ z\ zs) & \leftarrow & f^5\ z\ zs \\
f^5\ x\ zs & \leftarrow & False
\end{array}
$$

For the sake of brevity we have cheated slightly in that the call to c^2 in f^3 should actually be a call to a a copy d^2 of c^2 where the subfunctions of d^2 are copies of the subfunctions of c^2.

It is mainly the instantiation in clauses (3b),(4c) that account for this result. In particular, in this example it is significant that *all* the occurrences of v are instantiated in clause (3b).

If, in addition, the positive compiler would collect subsequent instantiations in one nested pattern (*i.e.* perform what we call ***backwards substitution*** in Section 9.5), and if it would also perform the optimization of checking whether different branches yielded the same result, and in this case omit the corresponding instantiations or tests, we could get:

$$\begin{array}{lcl} c\,(Cons\ Paris\ Nil) & \leftarrow & True \\ c\,(Cons\ Rome\ (Cons\ Paris\ Nil)) & \leftarrow & True \\ c\ x & \leftarrow & False \end{array}$$

It then holds that $\mathcal{I}[\![\, c\, d \,]\!] \Rightarrow True$ iff *connect(Vienna,d,[Rome,Paris])* is satisfied, where d is a M_1 list and Prolog list, respectively. One might say that the function c is a representation of the list of answers *L=[[Paris], [Rome,Paris]]* that the Prolog interpreter finds, in the sense that $\mathcal{I}[\![\, c\ d \,]\!] \Rightarrow True$ iff d is a member of L. But we have more than that—the internal structure of c reveals this fact in a very obvious way: c is basically a case dispatch with an entry for every true answer, and an otherwise clause for all false cases.

So, driving works very much as a Prolog interpreter. Such a similarity may at first sight be surprising, but can in fact be explained precisely. This is the purpose of the next section.

7.3 Transformation trees and SLD-trees

The following definition is right out of the textbooks.

DEFINITION **7.3.1** (SLD-tree with leftmost computation rule.) An *SLD-tree* for a logic program P and goal G is a tree of goals defined as follows. (*i*) the root contains G. (*ii*) let $A_1, A_2, \ldots, A_n$ be the goal in a node N, where the A_i's are atoms. Then for every program clause $A : -B_1, \ldots, B_m$ such that A and A_1 are unifiable with $MGU\ \theta$, N has a child $(B_1, \ldots, B_m, A_2, \ldots, A_n)\theta$. (*iii*) nodes containing the empty clause have no children.

Empty nodes are called *success nodes*, and other leaf nodes are called *failure nodes*. Branches ending in success and failure nodes are called success and failure branches, respectively.

For a success branch, the composition $\theta_1 \circ \ldots \circ \theta_n$, where θ_i is computed in the i'th application of (*ii*) above, is called a *computed answer*. □

Recall the description of transformation trees in Section 4.2. The structure of transformation trees for M_1 programs are similar to SLD-trees for Prolog programs. For instance, the transformation tree for

$$connect\ Vienna\ Copenhagen\ via$$

is structurally identical to the SLD-tree for

$$connect(Vienna, Copenhagen, via)$$

The SLD-tree ends with success nodes and failure nodes, the transformation tree ends with nodes containing $True$ and nodes containing $False$. The SLD-tree contains nodes such as

$$flight(Vienna,x), connect(x, Copenhagen, xs)$$

whereas the transformation tree contains nodes with

$$flight\ Vienna\ x = True\ \rightarrow\ connect\ x\ Copenhagen\ xs\ \square\ False$$

These are the same kinds of nested calls in different disguises. And just as the Prolog interpreter (SLD-tree searcher) performs instantiations from nodes to their children, so does the transformation tree.

In conclusion, the construction of an SLD-tree for a goal in a Prolog program and the construction of a transformation tree for a term in an M_1 program are very similar operations. Considering a transformation tree as a model of all computations with the term, see Section 4.2, and an SLD-tree as a model of all computations with a goal, this is perhaps less surprising. Specifically, the Prolog interpreter returns for every branch that ends in a success node the instantiations made along the way on that branch. The mechanism for building residual programs from transformation trees (graphs) returns a program containing a clause for every branch of the tree (provided that we allow backwards substitutions), where every left hand side is the result of applying the computed answer for the similar branch in the SLD-tree to the original left hand side, and the right hand side is $True$ or $False$, depending on the end node in the branch of the SLD-tree.

The remainder of this section develops a general result concerning the similarity between transformation trees and SLD-trees. We define a certain class M_s of M_0 terms and programs and a translation scheme from this class to Prolog goals and programs such that a transformation tree for an M_s term and program is isomorphic, in a certain sense, to the SLD-tree for the corresponding Prolog goal and program.

DEFINITION **7.3.2** (M_s terms and programs.) Let t, b, d, p range over M_s terms, values, definitions, and patterns, respectively:

$$\begin{array}{lcl} t & ::= & b \mid f\, b_1 \ldots b_n \mid g\, t\, b_1 \ldots b_n \\ b & ::= & v \mid c\, b_1 \ldots b_n \\ \\ d & ::= & f\, v_1 \ldots v_n \leftarrow t \\ & \mid & g\, p_1\, v_1 \ldots v_n \leftarrow t_1 \\ & & \quad \vdots \\ & & g\, p_m\, v_1 \ldots v_n \leftarrow t_m \\ \\ p & ::= & c\, v_1 \ldots v_n \end{array}$$

□

The interesting thing about M_s terms and programs is the limited way in which nested calls occur. For instance, an M_s can have form

$$g\ (f\, b'_1 \ldots b'_m)\, b_1 \ldots b_n$$

but the b's must be values, they cannot contain function calls. Also, function calls cannot occur under constructors. In short, function calls can only occur on the path from the top-level to the redex position. This means that programs and terms are order-of-evaluation independent, and that there is a complete separation of data-flow and control-flow. The latter property makes it easy to translate M_s terms and programs into Prolog goals and programs.

We have the usual notions of context, redex, and observable:

DEFINITION **7.3.3** Let e, r range over M_s contexts and redexes, respectively:

$$\begin{array}{lcl} e & ::= & [] \mid g\, e\, b_1 \ldots b_n \\ r & ::= & f\, b_1 \ldots b_n \mid g\, b_0\, b_1 \ldots b_n \end{array}$$

□

We also have a variation of the unique decomposition property:

PROPOSITION **7.3.4** *(The unique decomposition property.) For all t, either there exists a unique context-redex pair e, r such that $t \equiv e[r]$, or $t \equiv b$.* □

PROOF: Easy. □

DEFINITION **7.3.5** (Translation from M_s to Prolog.) Given M_s term and program t, p. Let r be a specific chosen variable, let in each clause y be a fresh variable, and define a translation $\underline{\bullet}$ from terms to lists of atoms as follows:

$$\begin{array}{lcl} \underline{b} & \equiv & r = b. \\ \underline{e[f\, b_1 \ldots b_n]} & \equiv & P_f(b_1, \ldots, b_n, y), \underline{e[y]} \\ \underline{e[g\ (c\, b_{n+1} \ldots b_{n+m})\, b_1 \ldots b_n]} & \equiv & P_g((c\, b_{n+1} \ldots b_{n+m}), b_1, \ldots, b_n, y), \underline{e[y]} \\ \underline{e[g\ v\, b_1 \ldots b_n]} & \equiv & P_g(v, b_1, \ldots, b_n, y), \underline{e[y]} \end{array}$$

Here we have assumed for simplicity that the Prolog and M_s syntax for terms are the same, so that Prolog terms are written *e.g.* $Cons\ x\ xs$, and not $[x|xs]$ or $cons(x, xs)$.

A translation on programs $\underline{p}$ is given by the following translation on each definition:

$$\underline{f\, v_1 \ldots v_n \leftarrow t} \equiv P_f(v_1, \ldots, v_n, r) :- \underline{t}$$

and

$$\underline{\begin{array}{lcl} g\, p_1\, v_1 \ldots v_n & \leftarrow & t_1 \\ & \vdots & \\ g\, p_m\, v_1 \ldots v_n & \leftarrow & t_m \end{array}} \equiv \begin{array}{lcl} P_g(p_1, v_1, \ldots, v_n, r) & :- & \underline{t_1} \\ & \vdots & \\ P_g(p_m, v_1, \ldots, v_n, r) & :- & \underline{t_m} \end{array}$$

□

EXAMPLE **7.3.6** The tail-recursive reverse function

$$\begin{array}{lcl} rev\ xs & \leftarrow & r\ xs\ Nil \\ r\ Nil\ ys & \leftarrow & ys \\ r\ (Cons\ x\ xs)\ ys & \leftarrow & r\ xs\ (Cons\ x\ ys) \end{array}$$

is translated into

$$\begin{array}{lcl} P_{rev}(xs, r) & :- & P_r(xs, Nil, y), y = r. \\ P_r(Nil, ys, r) & :- & ys = r. \\ P_r((Cons\ x\ xs), ys, r) & :- & P_r(xs, (Cons\ x\ ys), y), y = r. \end{array}$$

This program is equivalent to:

$$\begin{array}{lcl} P_{rev}(xs, r) & :- & P_r(xs, Nil, r). \\ P_r(Nil, r, r). & & \\ P_r((Cons\ x\ xs), ys, r) & :- & P_r(xs, (Cons\ x\ ys), r). \end{array}$$

which is indeed the standard, efficient Prolog list reversal predicate. □

REMARK **7.3.7** The latter predicate can be obtained instead of the former at the expense of some minor technical complications. One can also simply postprocess the Prolog program as follows: on every right hand side erase the atom $y = r$, and replace the occurrence of y with r. In the case where the predicate has only the atom $y = r$ on the right hand side, the right hand side dissappears.

Below we assume this is done. □

We would like a result stating that the M_s programs and their translated Prolog counterparts compute the same results. A natural idea is to state such a result in terms of our interpreter $\mathcal{I}$ and a Prolog interpreter. However, it is more convenient to state the result in terms of interpretation trees and SLD-trees.

The following proposition shows that interpretation trees can be used in a simple way to find the result on M_s terms and programs of the rewrite interpreter. A similar result does not hold generally for M_1 programs due to the possibility of function calls under constructors.

PROPOSITION **7.3.8** *For a ground M_s term t and an M_s program, (i) every node in $\mathcal{T}_\mathcal{I}[\![t]\!]$ has at most one child. (ii) $\mathcal{I}[\![t]\!] \Rightarrow b$, where b is not $\bot$, iff $\mathcal{T}_\mathcal{I}[\![t]\!]$ is finite and the last node contains b.* □

PROOF: Easy. □

Similarly, SLD-trees give the semantics of Prolog programs by means of computed answers. The SLD-trees of translated M_s terms and programs have a particular simple form.

PROPOSITION **7.3.9** *For any M_s term and program t, p, the SLD-tree for $\underline{t}, \underline{p}$ has exactly one branch.* □

PROOF: Easy induction on the length of the path from the root to nodes in the SLD-tree. □

The core of this phenomenon is that atoms are always called with all variables completely instantiated, except for the variable in the last argument, and this variable is instantiated after the clause is satisfied. The instantiation of the variable represents the returning of a value by a function in M_s programs.

We can now state that our translation preserves semantics.

PROPOSITION **7.3.10** *Given a ground M_s term t and a program p. Let T be the interpretation tree for t, T' be the SLD tree for $\underline{t}, \underline{p}$. It then holds that either both T and T' are infinite, or the last node of T contains b and T' has the computed answer $\{r := b\}$ in its single branch.* □

PROOF: Tedious, but not difficult. □

We finally have the desired equivalence between transformation trees and SLD-trees.

PROPOSITION **7.3.11** *Given M_s term and program t, p. Let T transformation tree for t, p, and T' the SLD-tree for $\underline{t}, \underline{p}$. Then there is a tree isomorphism ϕ from T to T' that maps a node N containing t to a node N' containing the empty clause if t is a constant, and $\underline{t}$ otherwise.* □

PROOF: Tedious, but not difficult. □

EXAMPLE **7.3.12** As an example consider the trees for the term $rev\ xs$ and program in Example 7.3.6. The beginning of one of the branches in the transformation tree is:

$$rev\ xs$$

$$|$$

$$r\ xs\ Nil$$

$$|$$

$$r\ xs'\ (Cons\ x'\ Nil)$$

$$|$$

$$r\ xs''\ (Cons\ x''\ (Cons\ x'\ Nil))$$

$$\vdots$$

The corresponding branch in the SLD-tree for the goal $P_{rev}(xs, r)$ and second Prolog program from Example 7.3.6 is:

$$P_{rev}(xs, r)$$

$$|$$

$$P_r(xs, Nil, r)$$

$$|$$

$$P_r(xs', (Cons\ x'\ Nil), r)$$

$$|$$

$$P_r(xs'', (Cons\ x''\ (Cons\ x'\ Nil)), r)$$

$$\vdots$$

□

The Proposition can be extended to state the obvious relationship between the substitutions calculated along the branches in thw two trees.

7.4 Previous relations between supercompilation and problem solving

A problem which falls conveniently within problem solving is the inversion of functions. In logic programming one expresses a program as a collection of relations, and computation consists in finding elements that are in relations. A Prolog predicate is basically a (partial) function to the domain $\{True, False\}$, so the operation of satisfying a goal is, roughly, the computation of the inverse image of $True$.

Examples of computation of inverse Refal functions by means of supercompilation are given in [Tur82, Glu89,Rom91]. The close connection between driving and interpretation of logic programs was noted by Glück in [Glu92a].

Chapter 8

Positive Supercompilation of Pattern Matching

> [...] a strategy that keeps a static track of dynamic values across conditional expressions [...] is enough to produce residual programs that traverse the dynamic data linearly.
>
> *C. Consel & O. Danvy.*[1]

This chapter gives a **case** study: transformation of a pattern matching program. This example is particularly interesting because it is an example of a transformation that partial evaluation and deforestation cannot perform automatically.

Section 8.1 introduces the problem as a general criterion for testing the strength of transformers. Section 8.2 discusses how to evaluate the specific output of transformers on the pattern matching program. Section 8.3 reviews previous results on transformation of pattern matching programs. Section 8.4 shows that the positive supercompiler performs well on the problem. Section 8.5 introduces some machinery for discussing the complexity of output programs from $\mathcal{W}$. Section 8.6 proves a theorem on the complexity of the output of the positive supercompiler on the pattern matching program.

8.1 A test for program transformers

A way to test a method's power is to see whether it can derive certain well-known efficient programs from equivalent naive and inefficient programs. One of the most popular such tests is to generate, from a naive pattern matcher and a fixed pattern, an efficient pattern matcher as output by the Knuth-Morris-Pratt algorithm [Knu77]. We shall call this *the KMP test.*

We give two programs for string pattern matching. The first contains nested calls, the second is tail-recursive. Both programs are naive, the former slightly more than the latter.[2]

[1] In [Con93].

[2] In this chapter we use the Miranda notation for lists, and we also use the short notation AAB for $[A, A, B]$.

DEFINITION **8.1.1** (General matcher, nested calls.)

$$
\begin{array}{lcl}
match\ p\ s & \leftarrow & prefix\ p\ s = True\ \rightarrow\ True \ \Box\ next\ p\ s \\[1em]
next\ p\ [] & \leftarrow & False \\
next\ p\ (s:ss) & \leftarrow & match\ p\ ss \\[1em]
prefix\ []\ ss & \leftarrow & True \\
prefix\ (p:ps)\ [] & \leftarrow & False \\
prefix\ (p:ps)\ (s:ss) & \leftarrow & p = s\ \rightarrow\ prefix\ ps\ ss \ \Box\ False
\end{array}
$$

□

DEFINITION **8.1.2** (General matcher, tail-recursive version.)

$$
\begin{array}{lcl}
match\ p\ s & \leftarrow & loop\ p\ s\ p\ s \\[1em]
loop\ []\ ss\ op\ os & \leftarrow & True \\
loop\ (p:pp)\ []\ op\ os & \leftarrow & False \\
loop\ (p:pp)\ (s:ss)\ op\ os & \leftarrow & p = s\ \rightarrow\ loop\ pp\ ss\ op\ os \ \Box\ next\ op\ os \\[1em]
next\ op\ [] & \leftarrow & False \\
next\ op\ (s:ss) & \leftarrow & loop\ op\ ss\ op\ ss
\end{array}
$$

□

Throughout this chapter our results will be shown for the tail-recursive version, but in fact all results carry over with only minor, irrelevant changes (for instance, the transformation graphs for the nested version are slightly different.)

Consider the naively specialized program $f\ u \leftarrow match\ AAB\ u$ which matches the fixed pattern AAB with a string u. Evaluation of $match\ AAB\ u$, given u, proceeds by comparing A to the first component of u, A to the second, B to the third. If at some point the comparison failed, the process is restarted with the tail of u.

However, this strategy is not optimal. Assume, for example, that the string u begins with three A's. In this case the first two components of the string match with the first two components of the pattern AAB, but the last comparison fails. It is inefficient to restart the comparison with the tail $AA...$ of u since it is already known that the first two tests of AAB against $AA...$ will succeed.

The specialized program corresponding to the DFA constructed by the Knuth-Morris-Pratt algorithm [Knu77] takes this information into account.

EXAMPLE **8.1.3** (KMP-style specialized matcher for AAB.)

$$
\begin{array}{lcl}
f\ u & \leftarrow & loop_{AAB}\ u \\[1em]
loop_{AAB}\ [] & \leftarrow & False \\
loop_{AAB}\ (s:ss) & \leftarrow & A = s\ \rightarrow\ loop_{AB}\ ss \ \Box\ loop_{AAB}\ ss \\[1em]
loop_{AB}\ [] & \leftarrow & False \\
loop_{AB}\ (s:ss) & \leftarrow & A = s\ \rightarrow\ loop_{B}\ ss \ \Box\ loop_{AAB}\ ss \\[1em]
loop_{B}\ [] & \leftarrow & False \\
loop_{B}\ (s:ss) & \leftarrow & B = s\ \rightarrow\ True \ \Box\ A = s\ \rightarrow\ loop_{B}\ ss \ \Box\ loop_{AAB}\ ss
\end{array}
$$

□

It is our aim to obtain this program automatically from the naive general matcher and a fixed pattern.

8.2 A comment on measuring complexity

One must be careful when discussing complexity of multi-input programs, especially in the context of program specialization when some inputs are fixed.

For an example, let π be either of the general pattern matchers seen earlier, and let $|p|, |s|$ denote the length of the pattern p and string s, respectively. Let $t_\pi(p,s)$ be the running time of program π on inputs p,s. Finally, let π_p be the result of specializing π to known pattern p by some transformation algorithm.

When π_p is a specialized KMP style pattern matcher, it is customary to say that the general $O(|p|\cdot|s|)$ time program has been transformed to an $O(|s|)$ time program. This is, alas, *always true*, even for trivial transformations such that of Kleene's $s-m-n$ Theorem [Kle52]. The reason is that as soon as $|p|$ is fixed, then $O(|p|\cdot|s|) = O(|s|)$, even though the coefficient in $O(_)$ is proportional to $|p|$.

To be more precise, define the *speedup function* as in [Jon93] as

$$speedup_p(s) = \frac{t_\pi(p,s)}{t_{\pi_p}(s)}$$

Now for any p there is a constant a and there are infinitely many subject strings s such that $t_\pi(p,s) \geq a\cdot|p|\cdot|s|$. Using a trivial specializer as in the $s-m-n$ Theorem it is easy to see that π_p has essentially the same running time as π, so $speedup_p(s) \approx 1$.

On the other hand, using non-trivial transformers (see later), the program π_p satisfies $t_{\pi_p}(s) \leq b\cdot|s|$ for any subject string s, where b is independent of p. As a consequence

$$speedup_p(s) \geq \frac{a\cdot|p|}{b}$$

This is particularly interesting because the speedup is not only significantly large, but becomes larger for longer patterns.

We shall say that the KMP test is passed by a transformer when there is a b such that for all p $t_{\pi_p}(s) \leq b\cdot|s|$.

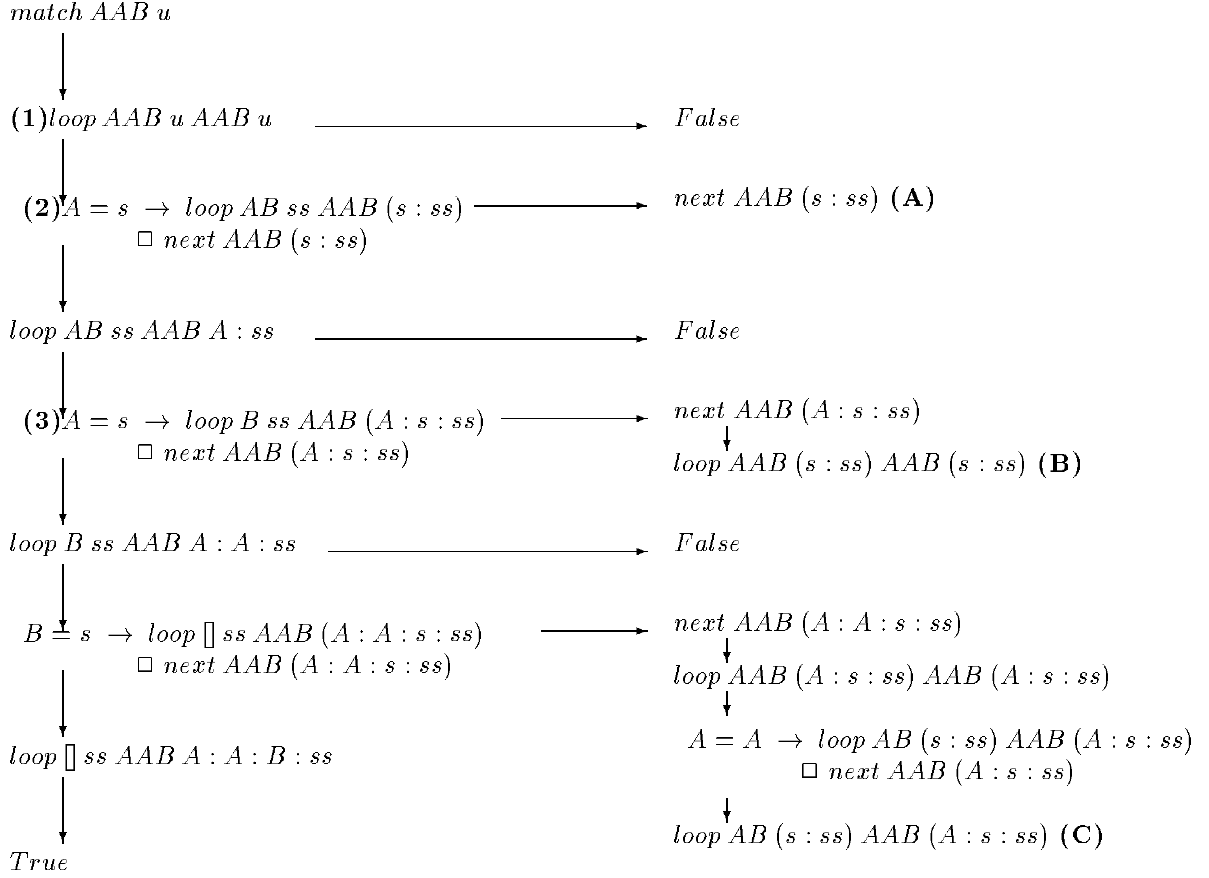
8.3 Previous results on the KMP test

Bird showed that his technique of *tabulation* or *recursion introduction* could pass the KMP test [Bir77]; but his technique was not mechanical and started from a much less natural matcher than either of the two above (it was essentially a 2-way pushdown automaton in program form.) Futamura and Nogi showed that generalized partial computation could pass the KMP test [Fut88] on the tail-recursive version of the algorithm, if one assumes the existence of a sufficiently powerful theorem prover. Consel and Danvy showed that a simple partial evaluator system could automatically pass the KMP test, but needed a "small insight" to rewrite the naive program to a significantly less naive form than those above [Con89]; see also [Jon93]. Glück and Turchin showed that a Refal-based supercompiler system could pass the KMP test automatically using the naive algorithm [Glu90]. Smith showed that a partial evaluator for a logic programming language could pass the KMP test, using a logic program like the tail-recursive one above [Smi91], but not on the logic program corresponding to the first of our programs. Recently Glück and Klimov have shown that a very simplified version of the supercompiler can pass the KMP test for a more traditional language [Glu93a], using the tail-recursive algorithm. As another approach, Glück and Jørgensen have shown that one can specialize a partial evaluator to an information propagating interpreter, thereby obtaining a supercompiler which can automatically pass the KMP test [Glu94a].

8.4 Supercompilation of a pattern matching program

In this section we show that $\mathcal{W}$ can derive a program almost as efficient as the KMP style pattern matcher.

Consider the transformation graph for the term *match AAB u* and the naive pattern matcher.



The nodes labelled (A),(B),(C) in the right column have arcs back to the nodes labelled (1),(2),(3), respectively, in the left column.

Notice how the instantiation of *all* occurrences of u and ss allows information to be passed to $next$ above $(A), (B), (C)$. We call the subsequently more instantiated versions of the fourth argument *the backup* of the string. It is crucial for the computation of the backup that the instantiation in clause (3b) of $\mathcal{W}$ apply to *all* occurrences of the variable v. These calls to $next$ can then be unfolded and some of the subsequent comparisons can be calculated. Specifically, above (C) it is known that we have a string $(A : A : s : ss)$, where s was not B. Moving one step to the right in the string we thus already know that the comparison between the head of the string and the head of the pattern will succeed (both are A); this is in fact what is calculated above (C). The generated program is:

EXAMPLE **8.4.1** (Almost KMP style matcher.)

$$loop_{AAB}\ u$$

$$\begin{array}{lcl}
loop_{AAB}\ [] & \leftarrow & False \\
loop_{AAB}\ (s : ss) & \leftarrow & A = s\ \rightarrow\ loop_{AB}\ ss \,\square\, next_{AAB}\ ss\ s \\
 & & \\
loop_{AB}\ [] & \leftarrow & False \\
loop_{AB}\ ss & \leftarrow & A = s\ \rightarrow\ loop_{B}\ ss \,\square\, next_{AB}\ ss\ s \\
 & & \\
loop_{B}\ [] & \leftarrow & False \\
loop_{B}\ (s : ss) & \leftarrow & B = s\ \rightarrow\ True \,\square\, A = s\ \rightarrow\ loop_{B}\ ss \,\square\, next_{AB}\ ss\ s \\
 & & \\
next_{AAB}\ ss\ s & \leftarrow & loop_{AAB}\ ss \\
next_{AB}\ ss\ s & \leftarrow & A = s\ \rightarrow\ loop_{AB}\ ss \,\square\, next_{AAB}\ ss\ s
\end{array}$$

□

This is almost the desired KMP matcher. Comparing to Example 8.1.3 we see that the only difference is that the calls to $next_{AB}$ and $next_{AAB}$ in $loop_{AAB}$,$loop_{AB}$, and $loop_B$ should have been calls to $loop_{AAB}$. Well, the call to $next_{AAB}$ *is* an indirect call to $loop_{AAB}$, but the two calls to $next_{AB}$ are not. The body of $next_{AB}$ tests $s = A$, and only if this is found not to hold is $loop_{AAB}$ called (via $next_{AAB}$.) But $next_{AB}$ is called only in false-branches of tests $A = s$, so the test in $next_{AB}$ will actually always be false.

In other words, the program is correct, but it contains a call to a function with a test which is known to be false. We shall see that these tests do not affect the asymptotic behaviour of the generated specialized matchers; in the terminology of Section 8.2, the last section will prove that there is a b such that for all p $t_{\pi_p}(s) \leq b \cdot |S|$.

It is not hard to explain the presence of the redundant tests mentioned above. The reason why these branches are not cut off in the graph above is that we are only propagating *positive* information. For instance, we get to (B) by the false branch in a test of s against A. Therefore, in (B) we actually know that s cannot be A, but this information is thrown away, and so we loop back to the similar situation in the left column where we did not know what s was and had to test it.

In the terminology of Section 4.5 the problem is that the underlying transformation tree is not perfect.

8.5 Transformation graph schemes

We are aiming towards a theorem stating for any pattern $[d_1, \ldots, d_n]$ a certain bound on the running time of the residual program obtained by applying $\mathcal{W}$ to the term $match\ [d_1, \ldots d_n]\ ss$.

The main difficulty lies in the fact that the information we are dealing with is rather complex. The reader may take another look at the transformation graph for $match\ AAB\ ss$ in Section 8.4 and consider how the transformation graphs for $match\ [d_1, \ldots, d_n]\ ss$ look *in general, i.e.* when we do not know what the d_i's are. This is not easy to explain in detail.

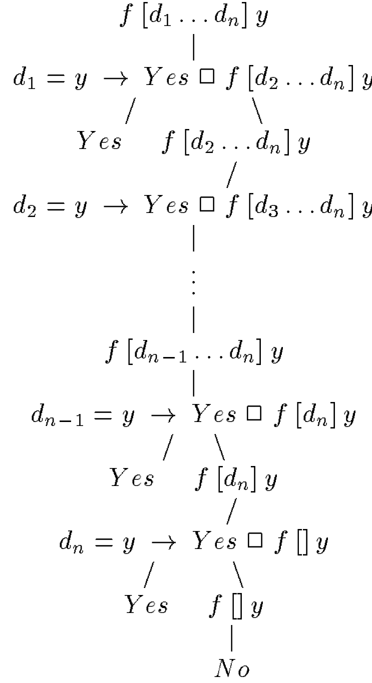
This section develops the notion of *transformation graph schemes* which will facilitate such a general explanation.

EXAMPLE **8.5.1** Consider the program

$$\begin{array}{lcl}
f\ []\ y & \leftarrow & No \\
f\ (x : xs)\ y & \leftarrow & x = y\ \rightarrow\ Yes \,\square\, f\ xs\ y
\end{array}$$

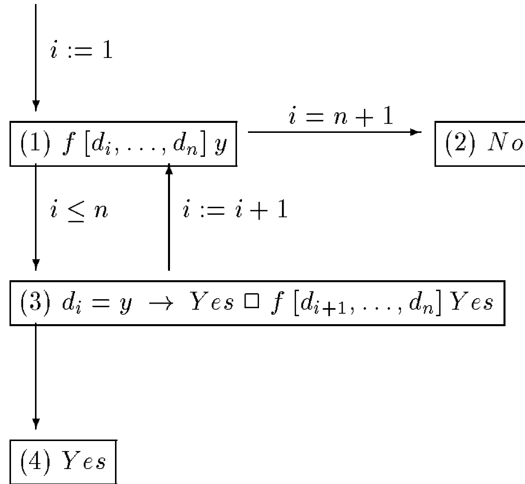
f answers Yes iff the second argument occurs in the first argument which is a list.

The transformation graph for $f\ [d_1 \ldots d_n]\ y$ for any given list $[d_1 \ldots d_n]$ is:



Here the situation is so simple that there is no difference between the transformation tree and the transformation graph.

The following is a transformation graph scheme for the original term and program. Below we put $[d_i, \ldots, d_n] = []$ when $i > n$.



□

The idea in understanding this diagram as a transformation graph is to read it as a non-deterministic flowchart program F with integer variable i and an integer constant n. To execute the program, one maintains a state consisting of the current control point (between (1) and (4)) and a value v_i for i. Execution starts in state $((1), 1)$. Whenever execution has reached a state $s = ((A), v_i)$ and there is a nonconditional transition (arrow with no condition attached to it) from (A) to (B), execution proceeds to a new state $s' = ((B), v_i')$. If there is a conditional transition from (A) to (B) (arrow with a condition attached to it) execution proceeds to s', provided the condition is true in state s. In both cases v_i' for

s' is obtained by excuting the assignments, if any, attached to the arrow left to right. The program is non-deterministic because there are two non-conditional transitions from (3).

To be perfectly explicit, the flowchart program is:

```
(0) i:=1; goto (1);
(1) if i<=n goto (3) else goto (2);
(2) halt;
(3) goto (4) OR (i:=i+1; goto (1));
(4) halt;
```

where the OR means that there is a free choice between either possibility.

Every state $s = ((A), v_i)$ reachable by F uniquely determines a term, t_s, obtained by substituting v_i for i in the term in box (A).

PROPOSITION **8.5.2** *There is a path from the root to a node containing t visiting k nodes (including the first and last) in the transformation graph $\mathcal{G}_{\mathcal{W}}[\![f\ [d_1, \ldots, d_n]\ y]\!]$ iff there is an execution of F in k steps ending in state $s = ((A), v_i)$ where $t_s \equiv t$.* □

PROOF: Each direction is shown by appropriate induction. □

Thus, if we read the labels in the transformation graph scheme as assignments and tests in a non-deterministic flowchart program, then the diagram represents the transformation graph.

Recall from Section 4.4 how the residual term and program are derived from the transformation graph. The residual term and program are: (before postunfolding)

$$\begin{array}{lll} & & f_1\ y \\ f_1\ y & \leftarrow & d_1 = y\ \rightarrow\ Yes \ \Box\ f_2\ y \\ & \vdots & \\ f_{n-1}\ y & \leftarrow & d_{n-1} = y\ \rightarrow\ Yes \ \Box\ f_n\ y \\ f_n\ y & \leftarrow & d_n = y\ \rightarrow\ Yes \ \Box\ f_{n+1}\ y \\ f_{n+1} y & \leftarrow & No \end{array}$$

Similarly we can extract the program from the transformation graph scheme. Then it would be natural to write it schematically:

$$\begin{array}{llll} & & f_1\ y & \\ f_i\ y & \leftarrow & d_i = y\ \rightarrow\ Yes \ \Box\ f_{i+1}\ y & (i \leq n) \\ f_i y & \leftarrow & No & (i = n+1) \end{array}$$

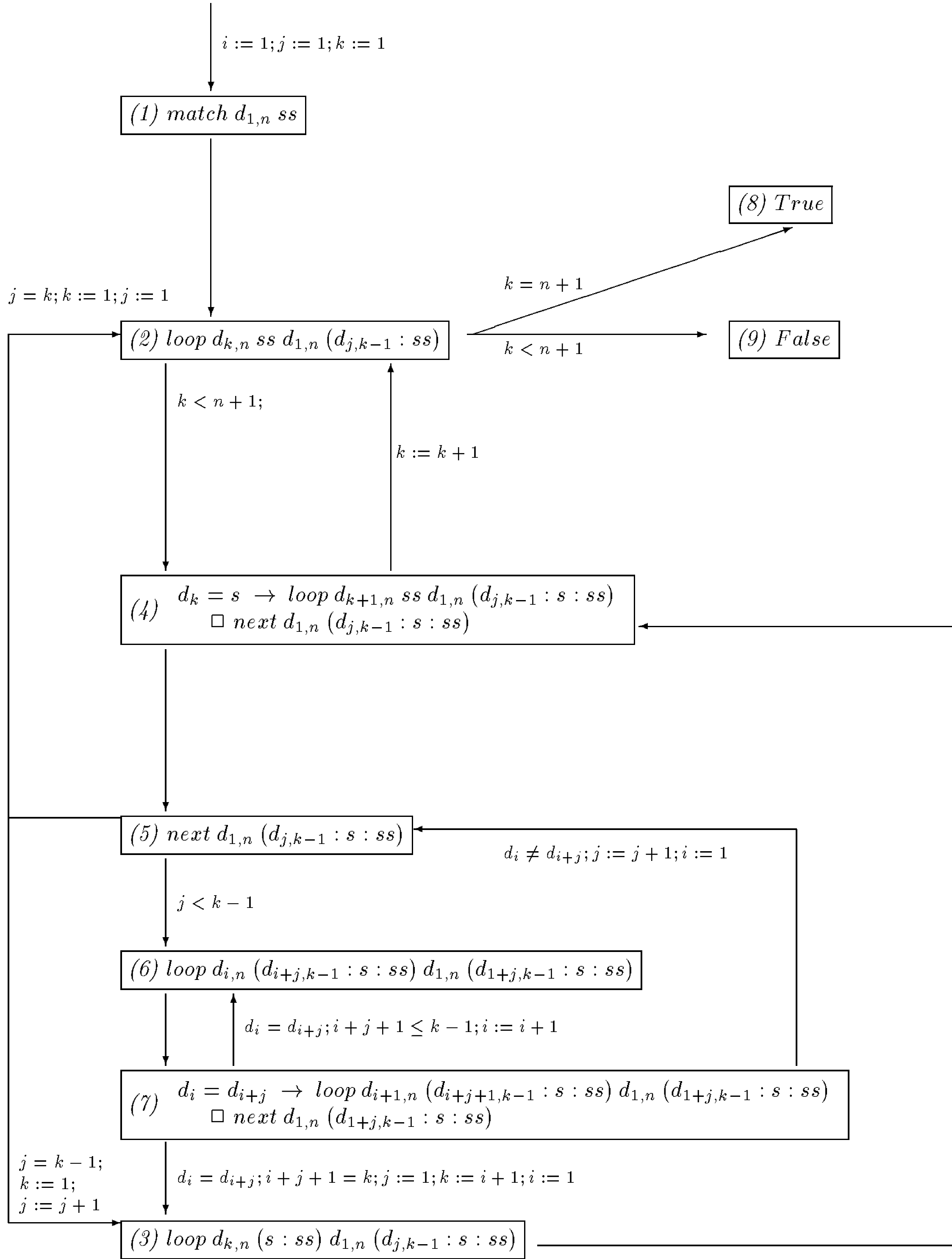
8.6 Theorem on complexity of specialized matchers

We now proceed to the theorem we are interested in. We first find a transformation graph scheme representing the real transformation graph. Then we show that this graph is finite so that the residual program can be extracted from it (and so that $\mathcal{W}$ terminates.) We then show the form of the residual term and program. We then give a measure of run-time complexity. Finally, we describe the effect of postunfolding, and conclude with a bound on the run-time of the residual program.

Transformation graph scheme

We use the notation $d_{i,j}$ for subpatterns $d_i \ldots d_j$. We put $d_i \ldots d_j = []$ when $i > j$. We also use the notation $d_{i,j} : ss$ for $d_i : \ldots : d_j : ss$, and put $d_{i,j} : ss = ss$ when $i > j$.

LEMMA **8.6.1** *There is a path from the root to a node containing t, visiting k nodes (including the first and last), in $\mathcal{G}_{\mathcal{W}}[\![match\ [d_1, \ldots, d_n]\ u]\!]$ iff there is an execution of the following transformation graph scheme F in k steps ending in state s where $t_s \equiv t$.*



□

Proof: From left to right is shown by induction on k as follows. For $k = 0$, the initial term $match\, d_{1,n}\, ss$ is in the state is $((1), 1, 1, 1)$. For a term t', encountered in $k + 10$ steps, $\mathcal{N}_{\mathcal{W}}[\![\, t \,]\!]$ contains t' for some term t encountered in k steps. The induction hypothesis says that $t = t_s$ for some state s. Now check

for every s in the diagram that when $\mathcal{N}_{\mathcal{W}}[\![\, t_s \,]\!]$ contains t' there is a transition by F from s to a state s' with $t_{s'} = t'$.

Right to left is similar. □

REMARK **8.6.2** The variables i, j, k correspond to some interesting positions in the pattern during transformation: j is an index into d signifying the first element of the backup of the string, $k-1$ is the position in d of the last element of the backup, and i is the offset from the beginning of the pattern (and from the beginning of the backup of the string) when the pattern is being matched against the backup. □

Termination of $\mathcal{W}$ on the example

LEMMA **8.6.3** *$\mathcal{W}$ applied to match $d_{1,n}$ ss terminates.* □

PROOF: We are to show that the transformation graph contains only finitely many different terms. This amounts to showing that F has only finitely many different reachable states. By Königs lemma, see *e.g.* [Tro88], we must show that every branch of execution of F reaches only finitely many different states.

This can be shown by showing that the invariant $1 \leq i, j, k \leq n+1$ is true in all states. This in turn can be shown by induction on the number of steps taken by F to reach the state, as follows. For the initial state the assertion is obvious. If s is reached by F in $p > 0$ steps, we must check that if the assertion holds in the $p-1$'th state, then it also holds in s. This can be done as follows. First check that the properties $1 \leq k \leq n+1, 1 \leq i, j$ are preserved by F's transitions. Then note that whenever j is increased by one it holds that $j \leq k-1$ before the incrementation, and whenever i is increased by one it holds that $i \leq k$ before the incrementation. □

Form of the residual program before postunfolding

LEMMA **8.6.4** *Before postunfolding, the residual term and program take the following forms:*

$$m_{((1),1,1,1)}\ ss$$

$m_{((1),1,1,1)}\ ss$	$\leftarrow$	$l_{((2),1,1,1)}\ ss$	
$l_{((2),1,j,k)}\ ss$	$\leftarrow$	$True$	where $k = n+1$
$l_{((2),1,j,k)}\ Nil$	$\leftarrow$	$True$	where $k < n+1$
$l_{((2),1,j,k)}\ (s : ss)$	$\leftarrow$	$d_k = s\ \rightarrow\ l_{((2),1,j,k+1)}\ ss\ \Box\ n_{((5),1,j,k)}\ ss\ s$	where $k < n+1$
$l_{((3),1,j,k)}\ ss\ s$	$\leftarrow$	$d_k = s\ \rightarrow\ l_{((2),1,j,k+1)}\ ss\ \Box\ n_{((5),1,j,k)}\ ss\ s$	
$n_{((5),1,j,k)}\ ss\ s$	$\leftarrow$	$l_{((6),1,j,k)}\ ss\ s$	where $j < k-1$
$n_{((5),1,j,k)}\ ss\ s$	$\leftarrow$	$l_{((3),1,j+1,k)}\ ss\ s$	where $j = k-1$
$n_{((5),1,j,k)}\ ss\ s$	$\leftarrow$	$l_{((2),1,j,k)}\ ss$	where $j = k$
$l_{((6),i,j,k)}\ ss\ s$	$\leftarrow$	$n_{((5),1,j+1,k)}\ ss\ s$	where $d_i \neq d_{i+1}$
$l_{((6),i,j,k)}\ ss\ s$	$\leftarrow$	$l_{((3),1,1,i+1)}\ ss\ s$	where $d_i = d_{i+1}, k = i+j+1$
$l_{((6),i,j,k)}\ ss\ s$	$\leftarrow$	$l_{((6),i+1,j,k)}\ ss\ s$	where $d_i = d_{i+1}, k > i+j+1$

□

PROOF: By derivation of the residual program from the graph scheme. □

It is important to realize that *e.g.* $l_{((2),1,j,k)}$ is a *class* of functions, one for every assignment of values to j, k, *i.e.* one for every different state $((2), 1, v_j, v_k)$ that the machine F from Lemma 8.6.1 encounters.

A measure for run-time complexity

When $\mathcal{I}$ is applied to the above program, $\mathcal{I}$ never encounters terms with an outermost non-0-ary constructor. This means that evaluation trees degenerate to sequences. In this case, a natural measure of the run-time is the length of the interpretation sequence. Informally, this is the number of rewritings

performed by $\mathcal{I}$ to arrive at the result. Specifically, unfolding an f- or g-function costs one unit and proceeding from a conditional to one of the branches costs one unit.

Thus, for $s \equiv [u_1, \ldots u_m]$ we let $t_{\pi_p}(s)$ denote the length of the interpretation sequence $\mathcal{T}_{\mathcal{I}}[\![\ \pi_p(s)\ ]\!]$. In Chapter 11 we introduce a general notion of run-time complexity of which the present measure is a special case.

Postunfolding

The postunfolding phase will reduce some of the f-function calls away, improving efficiency. We shall, however, not get into details with postunfolding; to keep things simple we assume that only the $l_{(6)}$ functions are reduced away, resulting in the program P. If at least the $l_{(6)}$ functions are reduced away by the postunfolding phase of $\mathcal{W}$, then clearly, the real residual program produced by the postunfolding phase has at least as good complexity as P.

LEMMA **8.6.5** *All calls to $l_{(6)}$ functions are unfolded by the postunfolding phase.* $\square$

PROOF: We know that the residual term and program before postunfolding has the form depicted in the preceding lemma. Recall that the postunfolding phase unfolds calls to functions which are called only once in the residual term and program. Now note that functions from the class $l_{(6)}$ only appear from the classes (i) $n_{(5)}$ and (ii) $l_{(6)}$. In case (i) the different calls have different j, k and so are mutually different, and similarly in class (ii). We therefore only need to show that there cannot be a call to the same function in both (i) and (ii), but this follows from the fact that all the calls in (ii) have $i > 1$ whereas the ones in (i) have $i = 1$. $\square$

Theorem on complexity

We start out with some informal intuition and then proceed to a rigorous proof.

The redundant tests that we saw in Section 8.4 occur when evaluation proceeds from an $l_{(2)}$ or $l_{(3)}$ function to an $l_{(3)}$ function via a number of $n_{(5)}, l_{(6)}, l_{(3)}$ functions. Letting l denote the current position in the string, it is obvious that whenever $l_{(2)}$ is called, l increases by one. Of course, l is bounded by the length of the string, m.

Assuming that the redundant tests are the only problem in ensuring that the complexity of the residual program is $b \cdot m$, for some b independent of m, n it suffices to show that the number of steps between two calls of $l_{(2)}$ are bounded by b, independent of m, n, because then the number of tests, and *a forteriori* of *redundant* tests by $l_{(3)}$ during the entire execution will be at most $b \cdot m$.

Actually we are not this lucky: there may be m redundant tests in a row not interrupted by calls to $l_{(2)}$, but we can show a sufficient weaker property by amortizing: the total number of calls to $l_{(3)}$ during the entire execution is at most m. It will turn out that every time we make progress in the string ($l_{(2)}$ is called), the enemy is granted one "credit" that he can spend on redundant tests whenever he like, but every test costs him one credit, and so the total number of redundant tests is at most m and we are safe (see [Tar83] if necessary.)

We now proceed to the proof.

THEOREM **8.6.6** *For any $p \equiv [d_1, \ldots, d_n], s = [u_1, \ldots, u_m]$, $t_{\pi_p}(s) \le 2m + 2$.* $\square$

PROOF: Given pattern $p \equiv d_1 \ldots d_n$ of length n, and string $s \equiv u_1 \ldots u_m$ of length m. Below, when we say i, j, k we mean the value of i, j, k in calls $f_{((A),i,j,k)}$ for various f.

We assume (for simplicity and without loss of generality) that the $l_{(6)}$ functions are the only ones that are postunfolded. Then it is easy to see that in any computation the number of steps from any term to a call of either $l_{(2)}$ or $n_{(5)}$ or to True or False is at most 2 (not counting calls to $l_{(6)}$.)

Let l denote the current position in the string. Each time we unfold a call to $l_{(2)}$, l is increased by one. This happens at most m times. Recall from the proof of Lemma 8.6.3 that $k - j$ is always non-negative. Each time we unfold a call to $n_{(5)}$ $k - j$ is decreased. $k - j$ is only increased when unfolding a call to $l_{(2)}$. So during the total execution $k - j$ is increased at most m times Since $k - j$ is non-negative

it is also decreased at most m times. Therefore there can be at most $2m+2$ steps in the total execution. □

The measure $k - j$ is the length of the current backup of the string that we have not used yet. In the calls to $n_{(5)}$ we are consuming the backup and so decreasing the measure. When we go to the $l_{(2)}$ we read another symbol from the string and so increase the backup's length with one. Clearly, the number of such incrementations is bounded by m.

Each step of $l_{(6)}$ corresponds to a comparison of an element in the pattern with an element in the backup of the string. The postunfolding of these calls is required for the preceding theorem to hold.

Chapter 9

Positive Supercompilation and Other Transformers

Although supercompilation includes partial evaluation,
it does not reduce to it.

Valentin F. Turchin.[1]

This long chapter relates $\mathcal{W}$ to other program transformers. We are concerned with: deforestation, partial evaluation of functional programs, generalized partial computation, interpretation and partial evaluation of logic programs, and Turchin's formulation of the supercompiler.

For the first two of a these transformers there is a section, first defining the transformer and then reviewing which of the effects of Chapters 5,6,7 and 8 can be achieved. We also try to explain why the effects can, or cannot, be achieved, thereby relating the different techniques in considerable detail. Each of the remaining transformers has a section which proceeds in lesser detail.

The chapter ends with Section 9.6 which sums up the results of the entire chapter. Specifically it describes the differences between call-by-value and call-by-name transformation and between various degrees of information propagation.

The history of the various transformers is described in Chapter 1, and is not repeated here.

9.1 Deforestation

Wadler invented the *listless transformer* [Wad84,Wad85] which eliminates intermediate lists, and subsequently the *deforestation algorithm* [Wad88,Fer88] which eliminates intermediate structures in general. The language studied in [Fer88] is what we call M_0; in [Wad88] a related language is studied. We shall be concerned with a deforestation algorithm $\mathcal{S}$ for M_1 which is identical to that in [Fer88] when applied to M_0 programs.

A deforestation algorithm

Perhaps it is best to start out with an example showing the difference between $\mathcal{S}$ and $\mathcal{W}$.

EXAMPLE **9.1.1** For the term and program

$$g\, v\, v$$

$$\begin{array}{lcl} g\,(S\,x)\,y & \leftarrow & y \\ g\,Zero\,y & \leftarrow & Zero \end{array}$$

[1] In *e.g.* [Tur86b].

the deforestation algorithm will return

$$g'\ v\ v$$

$$\begin{array}{lcl} g'\ (S\ x)\ y & \leftarrow & y \\ g'\ Zero\ y & \leftarrow & Zero \end{array}$$

whereas the positive supercompiler will return

$$g'\ v$$

$$\begin{array}{lcl} g'\ (S\ x) & \leftarrow & S\ x \\ g'\ Zero & \leftarrow & Zero \end{array}$$

□

DEFINITION **9.1.2** (Deforestation algorithm, $\mathcal{S}$) The exact rules of $\mathcal{S}$ can be explained from the 9 rules for $\mathcal{W}$ as follows.

In rule (4c) the substitution in the true branch is omitted. In rule (3b) a small change is made in the way residual functions are generated. For a term $e[g\ v\ t_1 \ldots t_n]$ we make a residual function call $g\ v\ u_1 \ldots u_k$. However, if v occurs among $t_1 \ldots t_n$ it is included among the variables $u_1 \ldots u_k$, so that the call has two occurrences of v, and in the bodies of the clauses of the residual function, only the one occurrence between g and $t_1 \ldots t_n$ is instantiated. □

REMARK **9.1.3** This is the *driving* part of deforestation. $\mathcal{S}$ needs also (*i*) a pregeneralization phase like our $\mathcal{W}$, (*ii*) a folding mechanism, and it is natural to include (*iii*) a postunfolding phase; see Section 3.7.

Below we assume the same folding and postunfolding strategies as for $\mathcal{W}$. (The folding mechanism in [Fer88] is essentially the same, and postunfolding is discussed only informally.) It will not be necessary to discuss generalizations, but see Section 13.1 in part III. □

Definitions of transformation graphs and trees for deforestation similar to those for positive supercompilation can be obtained as follows.

DEFINITION **9.1.4** (Transformation trees and graphs for deforestation.) Modify Definition 4.2.2 as follows. In clause (3b) only the occurrence of v between g and t_1 should be instantiated by the substitution $\{v := p_j\}$. Other occurrences of v must remain. In clause (4c) delete the application of MGU to $e[t]$. □

Now we are ready to discuss the effects of deforestation. We proceed in the following order: pattern matching, elimination of intermediate data structures, specialization, theorem proving, problem solving, compiler generation.

Pattern matching

The result of applying $\mathcal{S}$ to the naive specialized program $match\ AAB\ u$ is as follows.

EXAMPLE **9.1.5** (Non-improved specialized matcher.)

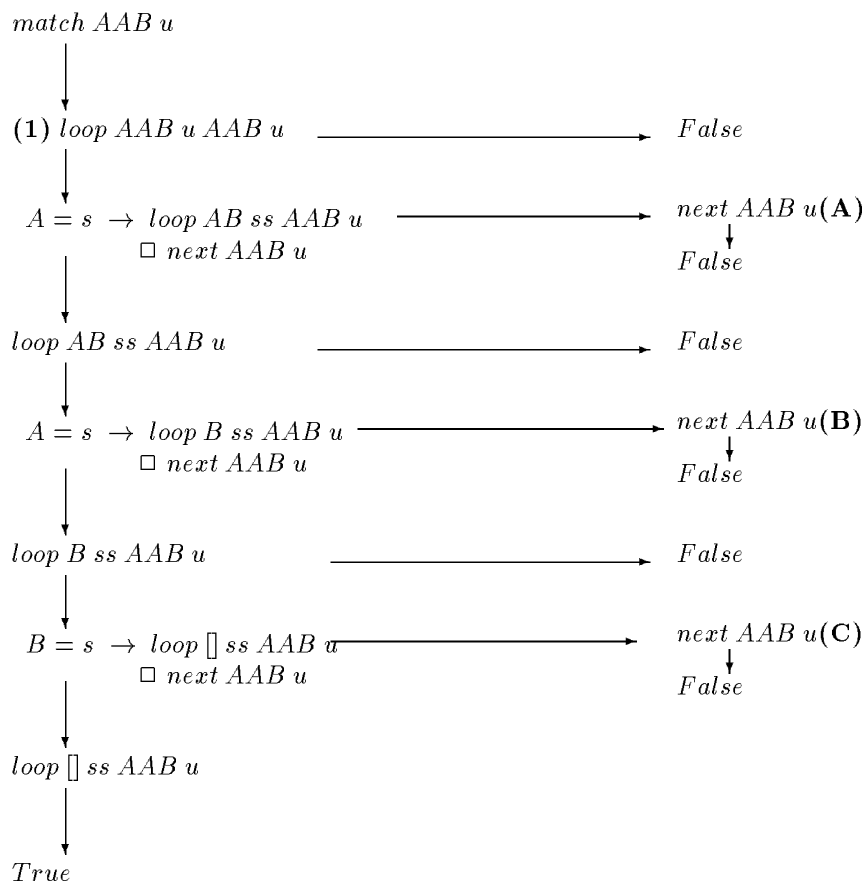
$$loop_{AAB}\ u\ u$$

$$\begin{array}{lcl} loop_{AAB}\ []\ os & \leftarrow & False \\ loop_{AAB}\ (s:ss)\ os & \leftarrow & A = s \rightarrow loop_{AB}\ ss\ os \ \Box\ next\ os \\ & & \\ loop_{AB}\ []\ os & \leftarrow & False \\ loop_{AB}\ (s:ss)\ os & \leftarrow & A = s \rightarrow loop_{B}\ ss\ os \ \Box\ next\ os \\ & & \\ loop_{B}\ []\ os & \leftarrow & False \\ loop_{B}\ (s:ss)\ os & \leftarrow & B = s \rightarrow True \ \Box\ next\ os \\ & & \\ next\ [] & \leftarrow & False \\ next\ (s:ss) & \leftarrow & loop_{AAB}\ ss\ ss \end{array}$$

□

This program is only improved in the sense that the p argument has been removed, *i.e.* $\mathcal{S}$ has performed program specialization. But each time a match fails, the head of the string is skipped, and the match starts all over again.

Consider the transformation graph for $\mathcal{S}$:



The nodes in the right column labelled (A),(B),(C) each has an arc back to the node in the left column labelled (1).

Here we see the problem clearly: the information gathered along the way about the string is not stored in the variable u; after a test of s against the head of the pattern, the matching is not instantiated in u in the true branch, and hence whenever a mismatch occurs, we always loop back to the same state.

One can rewrite the original program to manipulate this information explicitly. This is what the following program, adopted from [Con89,Jon93], does. For reasons that will become apparent in Section 9.2, the program is also called a *binding-time improved* matcher.

DEFINITION **9.1.6** (Information propagating pattern-matcher.)

$$
\begin{array}{lcl}
match'\ p\ s & \leftarrow & loop\ p\ s\ p\ []\ [] \\
 & & \\
loop\ []\ s\ op\ b\ ob & \leftarrow & True \\
loop\ (p : ps)\ []\ op\ []\ ob & \leftarrow & False \\
loop\ (p : ps)\ (s : ss)\ op\ []\ ob & \leftarrow & p = s\ \rightarrow\ loop\ ps\ ss\ op\ []\ (a\ ob\ [p])\ \Box\ next\ op\ (s : ss)\ ob \\
loop\ (p : ps)\ ss\ op\ (b : bs)\ ob & \leftarrow & p = b\ \rightarrow\ loop\ ps\ ss\ op\ bs\ ob\ \Box\ loop\ op\ ss\ op\ (tl\ ob)\ (tl\ ob) \\
 & & \\
next\ op\ (d : ds)\ [] & \leftarrow & loop\ op\ ds\ op\ []\ [] \\
next\ op\ (d : ds)\ (b : bs) & \leftarrow & loop\ op\ (d : ds)\ op\ bs\ bs
\end{array}
$$

Where a is the usual append function, and tl is the function that returns the tail of a list. □

The first argument of *loop* is the current pattern; this is always a suffix of the original pattern, which is stored in the third argument of *loop*. The second argument is the current string. The fifth argument is the prefix of the string that we have currently successfully matched with the pattern, *the backup*. This is maintained by always appending the head of the pattern to the backup after a succesful match against the string (true branch of third clause of *loop*.) At all times, the "original" string (*i.e.* the current candidate to have the pattern as a prefix) is the concatenation of the fifth and the second argument.

When we are matching the pattern with elements from the string, the third clause of *loop* is used. When we are matching the pattern against the backup, the fourth clause is used; then the fourth argument is the current suffix of the backup, and the fifth is the original backup which is retained in case a mismatch occurs while matching the pattern with the backup.

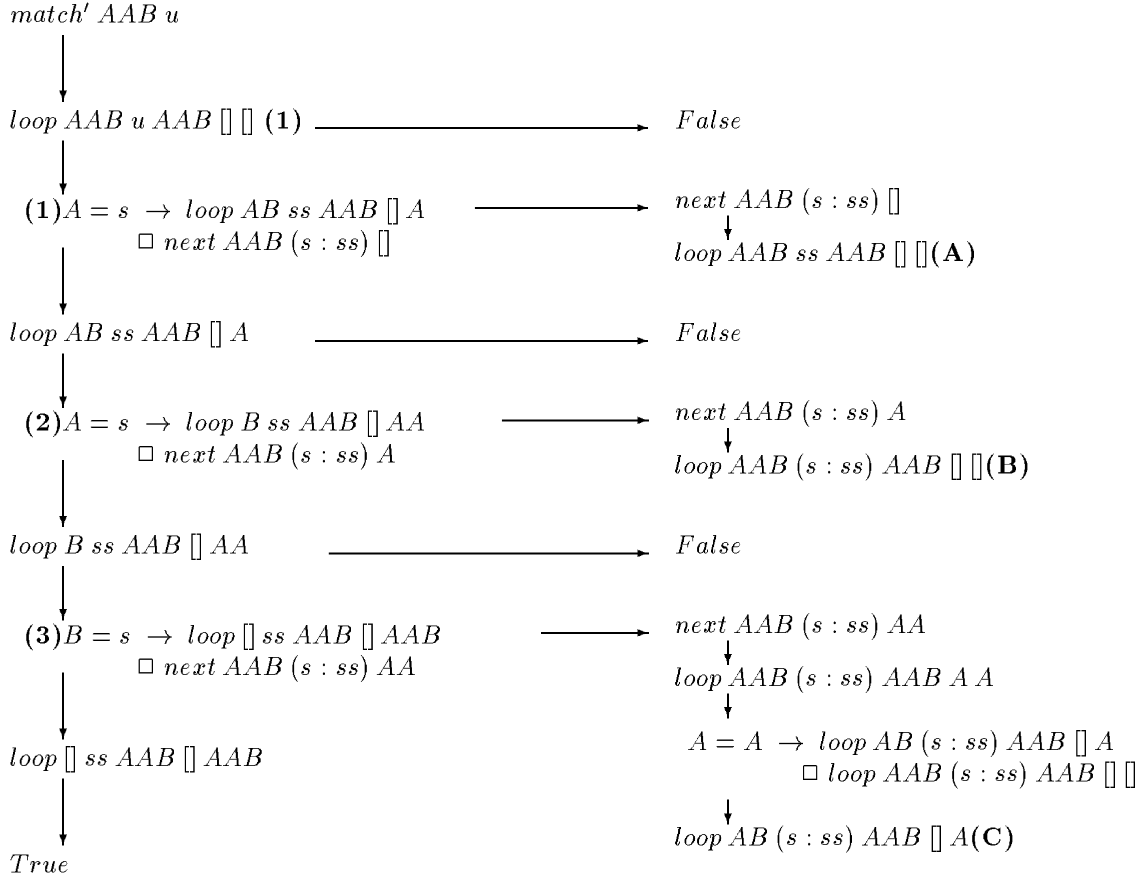
When a mismatch occurs there are two different cases.

The first case occurs when we were comparing the pattern against the string (third clause of *loop*.) The call to *next* in the false-branch then differentiates between two subcases: (*i*) the backup is empty (first clause of *next*) and (*ii*) the backup is non-empty (second clause of *next*.) In case (*i*) the "original" string and the current string are the same, and so we simply move one step to the right in the current string and start matching with empty backup. In case (*ii*) we move one step to the right in the backup and start matching with the new backup.

The second case occurs when we were matching the pattern against the backup (fourth clause of *loop*.) Then we take the "original" backup, *i.e.* the backup as it appeared when we started matching the pattern against it, move one step to the right in it, store this in the fifth argument as the new original backup, and start matching the pattern against it.

Note that this program is not more efficient than any of the naive matchers, it merely manipulates more information.

The transformation graph for the deforestation algorithm applied to $match'\ AAB\ u$ is structurally identical to the one for the positive supercompiler applied to $match\ AAB\ u$ in the tail-recursive program:



The labels (A)-(C) signify arcs back to (1)-(3) as usual. Here we have cheated slightly by calculating calls to a and tl before actually needed, but the reader may persuade himself that this makes no difference. The residual program produced by the deforestation algorithm is syntactically identical to the output of the positive supercompiler on the naive version.

In conclusion, deforestation is weaker than the positive supercompiler in its propagation of information: deforestation propagates the values of formal parameters when unfolding calls, but, unlike positive supercompilation, does not propagate the results of case tests on variables. Therefore, the deforestation algorithm does not pass the KMP test automatically.

Elimination of intermediate data structures

The very purpose of deforestation is to eliminate intermediate data structures which it does well, see [Wad88,Fer88]. For instance, deforestation gives the same as the positive supercompiler on the two examples in Section 5.1.

More generally, we have:

PROPOSITION **9.1.7** *For a linear M_1 term t and linear M_1 program p the residual term and program computed by $\mathcal{W}$ and $\mathcal{S}$ are the same.* □

PROOF: Obvious since the only differences are in clause (3b) and (4c), which are not manifest on linear terms and programs. □

This shows that for a large class of terms and programs, positive supercompilation and deforestation are the same.

REMARK **9.1.8** In fact, in [Fer88] deforestation is restricted to linear terms and programs to ensure non-degradation of efficiency, see Section 11.2. □

Specialization

Perhaps not so well-known deforestation can perform program specialization. For instance, the effect of deforestation on the two examples in Section 5.2 is the same as that of the positive supercompiler.

The above proposition shows that deforestation specializes as well as $\mathcal{W}$ on all linear M_1 terms and programs, a substantial class.

Theorem proving

We shall argue that $\mathcal{S}$ is a weaker theorem prover than $\mathcal{W}$.

In [Wad88] it is noted that deforestation can prove associativity of append: the terms $a\ (a\ xs\ xy) zs$ and $a\ xs (a\ ys\ zs)$ are transformed into the same residual term and function (they are identical MVR) which we showed in Example 5.1.1.

As for the program and term in Example 5.3.1, deforestation returns the following tail-recursive form:

$$c\ x\ x$$

$$\begin{array}{lcl}
c\ Z\ x & \leftarrow & e'\ x \\
c\ (S\ y)\ x & \leftarrow & g\ y\ x \\
\\
g\ v\ Z & \leftarrow & False \\
g\ v\ (S\ y) & \leftarrow & f\ v\ v \\
\\
e'\ Z & \leftarrow & True \\
e'\ (S\ y) & \leftarrow & False
\end{array}$$

This program also returns $True$ for all numbers x, but this is a lot harder to see than in the program produced by the positive supercompiler. The problem is that the fact that both arguments of c are the same was not taken into account when producing the residual program.

However, one can do things slightly differently, improving on the result. In the append example, we did not transform

$$e\ (a\ (a\ xs\ xy) zs)\ (a\ xs (a\ ys\ zs))$$

for some equality function e; we transformed each of the two arguments and observed that the result is the same.

So instead of requiring that the term $e\ (a\ Z\ x)\ x$ transform to $True$ we could require that the term $a\ Z\ x$ transform to x for the different cases of x. This corresponds to programming the simultaneous instantiations manually. Deforestation transforms $a\ Z\ x$ into:

$$\begin{array}{lcl}
 & & a'\ x \\
a'\ Z & \leftarrow & Z \\
a'\ (S\ y) & \leftarrow & S\ (a'\ y)
\end{array}$$

which must then recognized to be the identity. This is not harder than recognizing that the function in Example 5.3.1 is the $True$ function.

In conclusion, deforestation can prove some theorems. In the case of non-linear terms, deforestation is not as powerful as the positive supercompiler. At least in some cases this problem can be solved by splitting into cases and comparing the results manually. (Of course, when the non-linearity is irrelevant for the proof, *e.g.* when the variable with multiple occurrences is never instantitated, this does not have to be done.)

Problem solving

As was the case in the preceding subsection, deforestation can yield the desired results at the expense of some extra work.

Deforestation turns *connect Vienna Copenhagen via* in the functional program in the example in Section 7.2 into the following:

$$c_1\ x$$

$$\begin{array}{lcl}
c_1\ Nil & \leftarrow & False \\
c_1\ (Cons\ z\ zs) & \leftarrow & f_1\ z\ z\ zs \\
\\
f_1\ Paris\ z\ zs & \leftarrow & c_2\ zs\ z \\
f_1\ Rome\ z\ zs & \leftarrow & c_2\ zs\ z \\
f_1\ x\ z\ zs & \leftarrow & False \\
\\
c_2\ Nil\ z & \leftarrow & f_2\ z \\
c_2\ (Cons\ z'\ zs)\ z & \leftarrow & f_3\ z\ z'\ z'\ zs \\
\\
f_2\ Paris & \leftarrow & True \\
f_2\ x & \leftarrow & False \\
\\
f_3\ Vienna\ Paris\ z'\ zs & \leftarrow & c_2\ zs\ z' \\
f_3\ Vienna\ Rome\ z'\ zs & \leftarrow & c_2\ zs\ z' \\
f_3\ Rome\ Paris\ z'\ zs & \leftarrow & c_2\ zs\ z' \\
f_3\ Paris\ London\ z'\ zs & \leftarrow & c_2\ zs\ z' \\
f_3\ Paris\ Copenhagen\ z'\ zs & \leftarrow & c_2\ zs\ z' \\
f_3\ x\ y & \leftarrow & False
\end{array}$$

No doubt an extra phase to perform backwards substitution and identify branches with the same result, see Section 7.2, could also turn this program into the desired

$$\begin{array}{lcl}
c\ (Cons\ Paris\ Nil) & \leftarrow & True \\
c\ (Cons\ Rome\ (Cons\ Paris\ Nil)) & \leftarrow & True \\
c\ x & \leftarrow & False
\end{array}$$

But of course, then this phase would somehow have to cope with calls $g\ z\ z$, where g is defined by patterns on the first argument; that is: the positive information propagation would have to be built into this phase.

Compiler generation

As for the self-application and automatic production of compilers, deforestation is *in principle* as well suited as positive supercompilation. The important thing in principle is the change in functionality obtained by specializing a function to a known argument, as explained in Section 6.3. However, whether self-application of deforestation is likely to be feasible in practice is unclear, and we have not investigated the subject any further.

Conclusion

Deforestation performs as well as the positive supercompiler in situations where the positive information propagation is (often) not important: specialization and elimination of intermediate data structures.

Deforestation is weaker than positive supercompilation in those respects where positive information propagation is relevant: specialization of naive pattern matchers, theorem proving, and problem solving. In these cases the program can be rewritten to obtain the same result. Alternatively one can modify the

technique, either by incorporating positive information in the transformer thereby arriving at positive supercompilation, or by propagating positive information manually as in the theorem proving example.

Deforestation's pertinence to compiler generation remains unresolved.

9.2 Partial Evaluation of Functional Programs

The problem of program specialization goes back to the sixties; transformers to achieve this effect are called partial evaluators. Since then, an overwhelming body of literature has appeared, and conferences devoted to the subject exist, see Chapter 1.

What a partial evaluator can or cannot do depends on the partial evaluator. And there are many different partial evaluators and different *types* of partial evaluators: on-line/off-line, continuation-based/non-continuation-based, with/without partially static structures, *etc.*

To make certain points in this section and in subsequent chapters, it will be useful to fix the discussion at a specific partial evaluator $\mathcal{M}$ that we sketch in the first subsection below. Apart from relating $\mathcal{M}$ to the different effects of Chapters 5, 6, 7, and 8, we shall make a number of references to applications described in the literature for other partial evaluators.

A call-by-value partial evaluator

We shall describe a partial evaluator for a call-by-value language. At first this may appear to be an unwise choice because it seems to confuse the issue and render the comparison to positive supercompilation, a transformer for a call-by-name language, more difficult. Such criticism misses, however, the point that virtually all partial evaluators *are* for call-by-value languages, and in fact we shall argue that some of the differences between positive supercompilation and partial evaluation stem mainly from the fact that the former simulates call-by-name evaluation and the latter call-by-value evaluation.

First let us make precise what we mean by call-by-value evaluation of M_1 programs. This is given by the following definition, which the reader may like to compare to the call-by-name rewrite interpreter in Definition 2.4.5. The definition is followed by a remark that explains how the definition should be read.

DEFINITION **9.2.1** (Object language M_1^v.) The syntax of M_1^v is the same as that for M_1 (Definition 2.2.1.)

The call-by-value rewrite semantics is given by $\mathcal{I}^v$ below, where t's range over ground terms and a's range over constants.

$$(1a)\quad \mathcal{I}^v[\![\, c\, t_1 \ldots t_n \,]\!] \;\Rightarrow\; c\, t_1 \ldots t_n \quad \textbf{if } \text{all } t_i \text{ are passive}$$

$$(1b)\quad \mathcal{I}^v[\![\, c\, t_1 \ldots t_n \,]\!] \;\Rightarrow\; c\, (\mathcal{I}^v[\![\, t_1 \,]\!]) \ldots (\mathcal{I}^v[\![\, t_n \,]\!]) \quad \textbf{if } \text{not all } t_i \text{ are passive}$$

$$(2)\quad \mathcal{I}^v[\![\, f\, t_1 \ldots t_n \,]\!] \;\Rightarrow\; \mathcal{I}^v[\![\, t^f\{v_i^f := a_i\}_{i=1}^n \,]\!] \quad \textbf{where } \mathcal{I}^v[\![\, t_i \,]\!] \Rightarrow a_i,\ i = 1 \ldots n.$$

$$(3)\quad \mathcal{I}^v[\![\, g\, t_0 \ldots t_n \,]\!] \;\Rightarrow\; \mathcal{I}^v[\![\, t^{g,c}\{v_i^{g,c} := a_i\}_{i=1}^{n+m} \,]\!] \quad \textbf{where } \mathcal{I}^v[\![\, t_0 \,]\!] \Rightarrow c\, a_{n+1} \ldots a_{n+m} \text{ and } \mathcal{I}^v[\![\, t_i \,]\!] \Rightarrow a_i,\ i = 1 \ldots n.$$

$$(4a)\quad \mathcal{I}^v[\![\, t_1 = t_2 \to t_3 \,\square\, t_4 \,]\!] \;\Rightarrow\; \mathcal{I}^v[\![\, t_3 \,]\!] \quad \textbf{where } \mathcal{I}^v[\![\, t_1 \,]\!] \Rightarrow b, \mathcal{I}^v[\![\, t_2 \,]\!] \Rightarrow b', b \equiv b'$$

$$(4b)\quad \mathcal{I}^v[\![\, t_1 = t_2 \to t_3 \,\square\, t_4 \,]\!] \;\Rightarrow\; \mathcal{I}^v[\![\, t_4 \,]\!] \quad \textbf{where } \mathcal{I}^v[\![\, t_1 \,]\!] \Rightarrow b, \mathcal{I}^v[\![\, t_2 \,]\!] \Rightarrow b', b \not\equiv b'$$

$$(5)\quad \mathcal{I}^v[\![\, \text{let } v = t \text{ in } t' \,]\!] \;\Rightarrow\; \mathcal{I}^v[\![\, t'\{v := a\} \,]\!] \quad \textbf{where } \mathcal{I}^v[\![\, t \,]\!] \Rightarrow a$$

□

Remark **9.2.2** Here the metalanguage should be read as an eager language. For instance, clause (2) must be read as follows: to evaluate a call $f\ t_1 \ldots t_n$ *first* evaluate the arguments to constants, *then* evaluate the body of f with these constants substituted for formal parameters.

Although not stated so, clause (4a) and (4b) should be read as follows. Given $t_1 = t_2 \rightarrow t_3 \,\Box\, t_4$, first evaluate t_1, t_2 to constants b, b'. If these are identical evaluate t_1, otherwise evaluate t_2. Similar remarks apply to the formulation of $\mathcal{M}$ below. □

Now we define the partial evaluator $\mathcal{M}$. The algorithm is followed by an explanation of the details of the algorithm. It is probably best to read the explanation and along the way look in the definition to see how the details are expressed in the notation.

Definition **9.2.3** (Partial evaluator $\mathcal{M}$.) Below t's range over terms; a's range over terms containing variables, constructors, calls to residual functions $g^{\Box}$, resiual conditionals, and residual local definitions; b ranges over passive terms. Used as indexes a and p denote the number of active and passive terms, respectively. The variables $u_1 \ldots u_k$ are always the variables of the passive terms.

(1a) $\mathcal{M}[\![\, c\, t_1 \ldots t_n \,]\!] \quad \Rightarrow \quad c\, t_1 \ldots t_n$
where all t_i are passive

(1b) $\mathcal{M}[\![\, c\, t_1 \ldots t_n \,]\!] \quad \Rightarrow \quad c\,(\mathcal{M}[\![\, t_1 \,]\!]) \ldots (\mathcal{M}[\![\, t_n \,]\!])$
where not all t_i are passive

(2a) $\mathcal{M}[\![\, f\, t_1 \ldots t_n \,]\!] \quad \Rightarrow \quad \mathcal{M}[\![\, t^f\{v_i^f := b_j\}_{i=j}^n \,]\!]$
where $\mathcal{M}[\![\, t_j \,]\!] \Rightarrow b_j$, $j = 1 \ldots n$.

(2b) $\mathcal{M}[\![\, f\, t_1 \ldots t_n \,]\!] \quad \Rightarrow \quad f^\square\, u_1 \ldots u_k\, a_{i_1} \ldots a_{i_a}$
where $\mathcal{M}[\![\, t_{i_j} \,]\!] \Rightarrow a_{i_j}, \mathcal{M}[\![\, t_{k_l} \,]\!] \Rightarrow b_{k_l}$, $j = 1 \ldots a$, $l = 1 \ldots p$.
$f^\square\, u_1 \ldots u_k\, v_{i_1}^f \ldots v_{i_a}^f \leftarrow \mathcal{M}[\![\, t^f\{v_{k_l}^f := b_{k_l}\}_{l=1}^p \,]\!]$

(3a) $\mathcal{M}[\![\, g\, t_0 \ldots t_n \,]\!] \quad \Rightarrow \quad \mathcal{M}[\![\, t^{g,c}\{v_j^{g,c} := b_j\}_{i=1}^{n+m} \,]\!]$
where $\mathcal{M}[\![\, t_0 \,]\!] \Rightarrow c\, b_{n+1} \ldots b_{n+m}$ and $\mathcal{M}[\![\, t_j \,]\!] \Rightarrow b_j$, $j = 1 \ldots n$.

(3b) $\mathcal{M}[\![\, g\, t_0 \ldots t_n \,]\!] \quad \Rightarrow \quad f^\square\, u_1 \ldots u_k\, a_{i_1} \ldots a_{i_a}$
where $\mathcal{M}[\![\, t_0 \,]\!] \Rightarrow c\, d_{n+1} \ldots d_{n+m}$ and $\mathcal{M}[\![\, t_i \,]\!] \Rightarrow d_i$, $i = 1 \ldots n$.
b_{k_l} are the passive, a_{i_j} the active among the d's
$f^\square\, u_1 \ldots u_k\, v_{i_1}^{g,c} \ldots v_{i_a}^{g,c} \leftarrow \mathcal{M}[\![\, t^{g,c}\{v_{k_l}^{g,c} := b_{k_l}\}_{l=1}^p \,]\!]$

(3c) $\mathcal{M}[\![\, g\, t_0 \ldots t_n \,]\!] \quad \Rightarrow \quad g^\square\, d\, u_1 \ldots u_k\, a_{i_1} \ldots a_{i_a}$
where $\mathcal{M}[\![\, t_0 \,]\!] \Rightarrow d, \mathcal{M}[\![\, t_i \,]\!] \Rightarrow d_i$, $i = 1 \ldots n$
b_{k_l} are the passive, a_{i_j} the active among the d_i's
and d does not have an outermost constructor
$g^\square\, p_1\, u_1 \ldots u_k\, v_{i_1}^{g,c_1} \ldots v_{i_a}^{g,c_1} \leftarrow [\![\, t^{g,c_1}\{v_{k_l}^{g,c_1} := b_{k_l}\}_{l=1}^p \,]\!]$
$\vdots$
$g^\square\, p_m\, u_1 \ldots u_k\, v_{i_1}^{g,c_r} \ldots v_{i_a}^{g,c_m} \leftarrow [\![\, t^{g,c_1}\{v_{k_l}^{g,c_r} := b_{k_l}\}_{l=1}^p \,]\!]$

(4a) $\mathcal{M}[\![\, t_1 = t_2 \;\rightarrow\; t_3 \,\square\, t_4 \,]\!] \quad \Rightarrow \quad \mathcal{M}[\![\, t_3 \,]\!]$
where $\mathcal{M}[\![\, t_1 \,]\!] \Rightarrow b, \mathcal{M}[\![\, t_2 \,]\!] \Rightarrow b'$, b, b are constants with $b \equiv b'$

(4b) $\mathcal{M}[\![\, t_1 = t_2 \;\rightarrow\; t_3 \,\square\, t_4 \,]\!] \quad \Rightarrow \quad \mathcal{M}[\![\, t_4 \,]\!]$
where $\mathcal{M}[\![\, t_1 \,]\!] \Rightarrow b, \mathcal{M}[\![\, t_2 \,]\!] \Rightarrow b'$, b, b are constants with $b \not\equiv b'$

(4c) $\mathcal{M}[\![\, t_1 = t_2 \;\rightarrow\; t_3 \,\square\, t_4 \,]\!] \quad \Rightarrow \quad a_1 = a_2 \;\rightarrow\; a_3 \,\square\, a_4$
where $\mathcal{M}[\![\, t_i \,]\!] \Rightarrow a_i$
a_1, a_2 are not both constants

(5) $\mathcal{M}[\![\, \text{let } v = t_1 \text{ in } t_2 \,]\!] \quad \Rightarrow \quad \text{let } v = a_1 \text{ in } a_2$
where $\mathcal{M}[\![\, t \,]\!] \Rightarrow a_i$

□

REMARK **9.2.4** In clause (1a) we have a constructor term where all the arguments are passive, *i.e.* contain only constructors and variables, and so no further tranformation is possible. In case (1b) there are arguments to the constructor containing function calls, conditionals, or local definitions, and so we proceed to the arguments.

In clause (2a) we have an f-function call, where all the arguments transform to passive terms, *i.e.* constructor terms (possibly containing variables). In the case the call is unfolded right away. In clause (2b), not all arguments are transformed to passive terms. This means that in one or more arguments a residual call, conditional, or local definition ended up. In this case we make a new residual f-function which has $k + m$ parameters, where k is the number of free variables in the passive arguments, and m is

the number of active arguments. The latter become arguments in the residual call, whereas all passive arguments are propagated to the body of the function we are defining.

In Clause (3a) we have a g-function call, where all the arguments transform to passive terms. This is similar to (2a). In (3b), not all are passive, but the first argument transforms to a term with an outermost constructor. This is similar to (2b). In case (3c) the first argument is either a variable, a residual call, a residual conditional, or a residual local definition. In this case, we define a new residual g-function by the patters of the original g-function. If the first argument is a variable v, then only that occurrence of v is instantiated; if there are more occurrences of v in the passives, then v occurs among $u_1 \dots u_k$.

In clause (4a) we have a conditional where the two terms to test transforms to identical constants. In this case, we directly choose the true-branch. In clause (4b) we similarly chose the false-branch. In clause (4c) at least one of the terms to compare does not transform to a constant. In this case a residual conditional is produced, where both branches are transformed.

For postunfolding we assume that all f-functions called exactly once in the residual call are postunfolded.

For folding we assume that in clauses (2b),(3b),(3c),(4c) $\mathcal{M}$ checks whether the same residual function has previously been defined, and if so performs a fold step. We do not go into details with this. □

$\mathcal{M}$ is unusual, at least in the Copenhagen tradition, in the sense that it is an on-line technique, but it is rather similar to the partial evaluator considered in [Jon93, Chapter 17].

Now we are ready to discuss the effects of partial evaluation. We proceed in the following order: specialization, compiler generation, pattern matching, elimination of intermediate data structures, theorem proving, problem solving.

Specialization

Partial evaluation can achieve program specialization. Indeed, this is the very purpose of partial evaluation.

As examples, the reader may care to verify that the effect of $\mathcal{M}$ is identical to that of $\mathcal{W}$ on Examples 5.2.1 and 5.2.2 from Section 5.2. Not all partial evaluators can achieve this effect on the first of the two programs, because it involves partially static structures, but $\mathcal{M}$ handles it due to clause (3b) which allows the specialization with respect to a term where the outermost constructor is known.

Compiler generation

Partial evaluators have been established to work successfully in compiler generation ever since [Jon85] and will not be discussed any further here. It is not clear whether the respects in which partial evaluators are usually simpler than the positive supercompiler are essential for successful self-application.

Pattern matching

The partial evaluator $\mathcal{M}$ propagates only simple information: it binds passive arguments to formal parameters when unfolding. As was shown in the preceding section on deforestation, this is not enough to pass the KMP test. That a typical partial evaluator could not pass the KMP test automatically was shown in [Con89].

In the present setting, this phenomenon is reflected by the fact that $\mathcal{M}$ applied to the naive matcher in Definition 8.1.2 and pattern AAB yields the non-improved matcher in Example 9.1.5 that also deforestation yielded.

The traditional way to improve the result of partial evaluation is to modify the source programs. These modifications, called *binding-time improvements*, are semantics-preserving transformations of the source program which enable a partial evaluator to propagate more information and to achieve deeper specialization.

Consel and Danvy showed that partial evaluators can be used to derive specialized KMP matchers by an 'insightful rewriting' of the naive matcher [Con89] to improve its binding time separation. Their rewriting is similar in spirit to the one in the preceding section for the deforestation algorithm.

In the present setting it holds that $\mathcal{M}$ applied to the information propagating pattern matcher in Example 9.1.6 and pattern AAB yields the almost KMP matcher in Example 8.4.1 that also deforestation yielded on the information propagating pattern matcher (and which the positive supercompiler yielded from the naive matcher).

Alternatively, one can modify the rules (3c) and (4c) so as to propagate positive information. The needed change in $\mathcal{M}$ is similar to the change from $\mathcal{S}$ to $\mathcal{W}$. For example, for a call $g\, vv$, the result should be a residual call $g^{\Box}\ v$, where $g^{\Box}$ is defined by g's patterns, and in the i'th clause of $g^{\Box}$ the pattern p_i is substituted for the second formal parameter of g.

vvIf $\mathcal{M}$ is so changed it in fact returns the almost KMP matcher from the naive matcher and the pattern AAB. That the addition of positive information is sufficient to get partial evaluators that can pass the KMP test was mentioned in [Con93], see the quotation to Chapter 8.

elimination of intermediate data structures

Most partial evaluators do not eliminate intermediate data structures. This is due to the handling of nested calls. The handling of nested calls differs from that of deforestation or positive supercompilation as explained below.

Partial evaluators are usually concerned with languages that have a call-by-value semantics, and therefore they simulate call-by-value semantics. The rule for partially evaluating a call $f\, t_1 \ldots t_n$ could be: if all the arguments of f are completely known and evaluate to constants $b_1 \ldots b_n$, then f will be transformed to a constant a'; otherwise the result is a residual call $f'\, x_1 \ldots x_m$ where f' is an optimized version of f taking into account the partially known values.

Reading both "completely known" and "constant" as "passive term" this is the approach taken in $\mathcal{M}$.

Clearly such a strategy does not allow the propagation of constructors in a term like $g_1\, (g_2\, v)$. For instance, $\mathcal{M}$ returns the original program unchanged in Examples 5.1.1 and 5.1.2 from Section 5.1 on elimination of intermediate data structures.

Consider similarly to the situation with nested calls the conditional $g\,(x_1 = x_2 \ \rightarrow\ a_1 \Box a_2)$. Suppose that x_1, x_2 are variables, and b_1, b_2 constants. Then, according to the rule above, we shall have a residual conditional, and around that a residual call to g, even though both branches of the conditional are known. Here it would be desirable for the partial evaluator to check its result to see that it was a conditional with known branches and in such case proceed to $x_1 = x_2 \ \rightarrow\ b'_1 \Box b'_2$ where b'_1, b'_2 are the results of transforming $g\, b_1, g\, b_2$, respectively. This involves manipulating the context of a term explicitly.

To sum up this far, partial evaluation does not eliminate intermediate data structures because constructors are prevented from propagating from the producers out to the consumers. There are three ways of solving this problem.

First, Consel and Danvy have shown that instead of having the partial evaluator manipulate contexts explicitly, one can call-by-value CPS-transform the object program before partially evaluating it to obtain the same effect [Con91].

Second, Bondorf has shown that one can manipulate the context explicitly in an elegant way by writing the specializer itself in CPS style [Bon92].

Third, one could take the formulation of deforestation or positive supercompilation from the present thesis and redefine the notion of contexts e to call-by-value evaluation contexts. This would be a call-by-value transformer which could be called a partial evaluator. This transformer manipulates the context explicitly without being written in CPS style. For instance it moves the context of a conditional into the branches in clauses (4c),(5) and it moves what corresponds to the context of a case expression into the branches in clauses (2),(3a),(3b).

The three approaches seem essentially the same. The situation resembles that of the relationship between call-by-value *interpretation* and call-by-value CPS-translation [Rey72,Plo75]. It would seem, then, that neither of these approaches can achieve the general elimination of intermediate data structures as in deforestation. All three techniques can move contexts to the branches of conditionals, but this is something weaker than having recursive functions propagate constructors by unfolding calls.

EXAMPLE **9.2.5** For example, let us see what the third approach yields on the double-append program. We could define transformation trees and graphs for this algorithm merely by changing the notion of

context to call-by-value evaluation contexts in Definition 4.2.2. Then the transformation tree contains an infinite branch:

$$
\begin{array}{c}
a\ (a\ xs\ ys)\ ts \\
| \\
a\ (Cons\ x(a\ xs\ ys))\ ts \\
| \\
a\ (Cons\ x(Cons\ x'\ (a\ xs\ ys)\ ))\ ts \\
\vdots
\end{array}
$$

At least if the transformer used the same kind of folding scheme as $\mathcal{W}$, the result would be infinite specialization. □

In other words, the general elimination of intermediate data structures seems to rely on the transformer simulating call-by-name evaluation, whereas the CPS-transformation in [Con91] and the partial evaluator in [Bon92] simulate call-by-value transformation.[2] A more close examination of this question remains a piece of future work.

Theorem proving

As has been shown by Julia Lawall, partial evaluation can perform a certain amount of theorem proving [Law93]. Depending on the particular theorems being proven, and the particular method, one can at least to some extent do without positive information progation, as we already saw in the preceding section on deforestation. Specifically, $\mathcal{M}$ yields the same result as deforestation on the term from Example 5.3.1 from Section 5.3 concerning theorem proving. However, its handling of nested calls makes $\mathcal{M}$ a less powerful theorem prover than $\mathcal{S}$. Specifically, $\mathcal{M}$ returns both of the double-append terms $a\ (a\ zs\ ws)\ ts$ and $a\ zs\ (a\ ws\ ts)$ unchanged, and so cannot prove the associativity of append, which $\mathcal{S}$ could.

Problem solving

Our partial evaluator $\mathcal{M}$ returns the same as $\mathcal{S}$ on the program and term from the example in Section 7.2, but the handling of nested calls in general makes partial evaluation poor for achieving the effect of logic programming. We do not go into the details of this question.

Conclusion

Partial evaluation performs as well as the positive supercompiler in situations where the positive information propagation and the call-by-name evaluation is not crucial: specialization and compiler generation. It is weaker than positive supercompilation with respect to elimination of intermediate data structures, specialization of naive pattern matchers, theorem proving, and problem solving.

We end the section by a comment on the need for binding-time analysis in call-by-value transformers. Recall that call-by-value partial evaluators typically do not manipulate the context explicitly. When $\mathcal{M}$ is applied to the term $g_1\ (g_2\ v)$, the decision whether or not to unfold g_1 depends on whether g_2 can be calculated or not. So either the partial evaluator must see whether the result of transforming the argument of g_1 yielded a value, or it must have annotations inserted in the program before partially evaluating it, safely estimating such information. In $\mathcal{M}$, the approach is the former: to always check whether the result is a passive term or not. In the Copenhagen tradition, the binding-time information is computed prior to partial evaluation, and the partial evaluator does not examine the result of transforming arguments to see whether they became values or residual calls.

When $\mathcal{M}$ encounters the term $e[g\ v]$, where g is defined by patterns, g cannot be unfolded right away since there is not enough information to decide which clause of g should be chosen. The results is to first

[2] This conjecture is supported by experimental results conducted by a fellow student, Kristian Nielsen, who has found that call-by-value CPS-translation of programs with intermediate data structures often leads to infinite specialization with the Similix partial evaluator, whereas call-by-name CPS-translation followed by partial evaluation often achieves a deforestation-like effect.

transform the term into $e[g^{\square}\ v]$, and then $\mathcal{M}$ looks at the first surrounding function in e, that is: the residual call is placed *inside* the context. When $\mathcal{W}$ encounters the term $e[g\ v]$, the *entire term* $e[g\ v]$ is replaced by a call to a residual function $g^{\square}$ defined by g's patterns and whose right hand sides initially have form $e[t^{g,c_i}\{v^{g,c_i} := p_i^g\}]$.

This explains why $\mathcal{W}$ neither relies on a binding-time analysis nor has to see whether terms contain residual calls; $\mathcal{W}$ simply never encounters a term with a residual call!

9.3 Generalized partial computation (GPC)

GPC was introduced by Futamura and Nogi [Fut88] and later applied to a lazy functional language by Takano [Tak91]. GPC is a powerful transformation method because it assumes the (unlimited) power of a theorem prover, thereby extending partial evaluation as follows.

Whenever an if-construct (or something equivalent) testing whether predicate P holds is encountered during the transformation, P is propagated to the true branch and the predicate $\neg P$ is propagated to the false branch. Also, whenever a test is encountered, a theorem prover sitting on top of the transformer tests whether more than one branch is possible. If only one is possible, only that branch is taken.

Positive supercompilation and GPC are related, but differ in the propagation of information. While GPC propagates positive and negative information about arbitrary predicates requiring a theorem prover, positive supercompilation propagates only positive information arising from equality tests and pattern matchings.

Takano described generalized partial computation in more detail, using the lazy functional language from the original deforestation paper [Wad88]. There is one rule which is of particular interest for our purposes.

$$\begin{array}{l} G[\![\ \mathbf{case}\ v\ \mathbf{of}\ p_1 : t_1 \mid \ldots \mid p_n : t_n\ ]\!]E \Rightarrow \\ \qquad \mathbf{case}\ v\ \mathbf{of}\ p_1 : G[\![\ t_1\ ]\!]E_1 \mid \ldots \mid p_n : G[\![\ t_n\ ]\!]E_n \\ \mathbf{where} \\ E_i = E \cup \{v \leftrightarrow p_i\} \end{array}$$

The E's are sets of equalities (sets of predicates) which are used in the manner described in more general terms in [Fut88]; concretely, they represent positive information arising from pattern matching. The algorithm actually uses a mixture of substitution based and environment representation of information. There is no notion of negative information in the language of [Tak91] because it has no else-construct or otherwise-construct.

A related substitution based version is:

$$\begin{array}{l} G[\![\ \mathbf{case}\ v\ \mathbf{of}\ p_1 : t_1 \mid \ldots \mid p_n : t_n\ ]\!]E \Rightarrow \\ \qquad \mathbf{case}\ v\ \mathbf{of}\ p_1 : G[\![\ t_1\{v := p_1\}\ ]\!]E \mid \ldots \mid p_n : G[\![\ t_n\{v := p_n\}\ ]\!]E \end{array}$$

So we see here that Takano's formulation of GPC incorporates positive information propagation in a style very similar to that in the positive supercompiler.

We shall not get into details with the handling of nested calls in GPC. One can imagine either a call-by-name simulation or a call-by-value simulation. Although stated for a lazy language, the formulation of GPC in [Tak91] is not as powerful as deforestation or supercompilation in eliminating intermediate data structures.

Concretely the rule

$$\begin{array}{l} \mathcal{S}[\![\ \mathbf{case}\ f\ t'_1 \ldots t'_m\ \mathbf{of}\ p_1 : t_1 \mid \ldots \mid p_n : t_n\ ]\!] \Rightarrow \\ \qquad \mathcal{S}[\![\ \mathbf{case}\ t^f\{v_i^f := t'_i\}_{i=1}^m\ \mathbf{of}\ p_1 : t_1 \mid \ldots \mid p_n : t_n\ ]\!] \end{array}$$

from deforestation [Wad88] is taken in the form

$$\begin{array}{l} G[\![\ \mathbf{case}\ f\ t'_1 \ldots t'_m\ \mathbf{of}\ p_1 : t_1 \mid \ldots \mid p_n : t_n\ ]\!] \Rightarrow \\ \qquad \mathbf{case}\ e\ \mathbf{of}\ p_1 : G[\![\ t_1\ ]\!]E_1 \mid \ldots \mid p_n : G[\![\ t_n\ ]\!]E_n \\ \mathbf{where} \\ \qquad G[\![\ f\ t'_1 \ldots t'_m\ ]\!] \Rightarrow e \\ \qquad E_i = E \cup \{e \leftrightarrow p_i\} \end{array}$$

which does not eliminate intermediate structures constructed by f. Compared to the former rule, the latter rule eases the problem of ensuring termination which is an important issue in [Tak91].

9.4 Interpretation and partial evaluation of Prolog Programs

We shall not discuss whether Prolog interpreters or partial evaluators can achieve one or more of the effects in Chapter 5. Let it suffice to say that we have observed that the actions of the positive supercompiler on a functional program in some cases is very close to the building of an SLD-tree for a similar Prolog program.

A few words concerning the KMP test from Chapter 8 are appropriate. An automatic derivation of the KMP style pattern matcher from the naive tail-recursive program has been reported in [Smi91]. Strictly speaking, the partial evaluator in [Smi91] performs generalized partial computation. It uses substitution based positive information propagation on the result of matchings and true branches of equalities, and it uses a kind of environment based negative information propagation on the result of false branches of equalities, so-called *disequality constraints.*

The nested naive matcher in Prolog is:

$$
\begin{array}{l}
match(P,T) :- prefix(P,T).\\
match(P,[X|R]) :- match(P,R).\\
prefix([],X).\\
prefix([HP|TP],[HP|TS]) :- prefix(TP,TS).
\end{array}
$$

When the partial evaluator in [Smi91] is applied to this general matcher with fixed pattern, no efficiency is gained. The problem is that all the information accumulated in the attempt to satisfy $prefix(P,T)$ is rolled back when the evaluator backtracks to the second clause for $match$. However, Smith's system does pass the KMP test if the program is rewritten to a program in the style of the tail-recursive matcher.

9.5 Relation to Turchin's supercompiler

In this section we briefly review the connections between the positive supercompiler and the supercompiler described by Turchin. The latter can achieve all the effects with which we are concerned so the present section only compares the details of the algorithms, not their effects.

In the works of Turchin, the supercompiler is always described as a metaalgorithm which constructs graphs from which programs can be described, just like we did for the supercompiler in Section 4.5. As for the explanations in Sections 3.4 and 3.5, Turchin always uses the latter; as a matter of fact, the relevance of the former is explicitly denied, see the quotation to Chapter 3. Section 3.4 suggests that the two views are in fact complementary.

Of course, the correctness of Turchin's supercompilation ultimately relies on the correctness of certain transformations on terms. These transformations were first stated in [Tur72] and have, as far as the author knows, only appeared in English in [Tur80a] Chapter 3. They comprise 5 *algorithmic* equivalences, concerning manipulation of the *program*, *i.e.* reversing the order of clauses, deleting clauses *etc.* and 2 *functional* equivalences, concerning transformation of programs, *i.e.* unfolding *etc.*

It is sometimes said that the semantic foundation of supercompilation is unclear, and as a motivation for this is given that the supercompiler does not preserve the exact semantics: it can extend the domain of functions. There are basically three ways this can happen, all explained in [Tur80a].

1. In Refal multiple and nested patterns are allowed, and a function is allowed to have clauses where the patterns are overlapping—the order of the clauses in the program decides which clause is applied. Among the algorithmic equivalences is a rule allowing the addition of a new clause for a function g to be added at the end to the existing clauses for g. This rule trivially extends the domain.

   For instance, to the program $f\ One\ y \leftarrow One$ we can add at the end the clause $f\ x\ y \leftarrow x$ to make f the total function which always returns its first element.

2. Refal is understood as a call-by-value language, but the supercompiler simulates call-by-name evaluation. This means that it can throw away non-terminating computations and hence extend the domain.

   For instance, assuming that f always returns its first element, we can, using the functional equivalences, turn the nowhere defined function $h\,x \leftarrow f\,x\,(h\,x)$ into the total identity function $h'\,x \leftarrow x$.

3. Finally, the functional rules perform a kind of backwards substitution when unfolding which can extend the domain of functions.

   Consider for instance the following program.

$$\begin{array}{l} f\ (Succ\ x) \leftarrow g\ x \\ f\ y \leftarrow y \\ g\ Zero \leftarrow Zero \end{array}$$

   Here f is clearly undefined on $Succ\ (Succ\ Zero)$. Transforming f by instantiating x and unfolding the call to g leads to the following transformed program:

$$\begin{array}{l} f\ (Succ\ Zero) \leftarrow Zero \\ f\ y \leftarrow y \end{array}$$

   Now f is total and behaves as the identity except on $Succ\ Zero$.

   The problem is that the backwards substitution can prevent matchings that lead to calls to functions with illegal arguments. This implies that subsequent, possibly more defined, clauses will be used instead.

It may be noted that none of these phenomena occur with the positive supercompiler, see Chapter 10. While we have seen that the usual effects of supercompilation can be achieved by the positive supercompiler, it should be noted that in the case of problem solving it was very convenient to have backwards substitution, so at least in this respect, Turchin's supercompiler performs better than $\mathcal{W}$.

Apart from what has already been said, the main difference between the positive supercompiler and Turchin's is that whereas the former propagates only positive information, the latter also propagates negative information in the form of restrictions *e.g.* $v \neq w$.

How this is done has been explained very clearly by Glück and Klimov [Glu93a]. In their work, both positive and negative information is propagated by bindings $v \mapsto t$ in two environments, one of bindings that hold, and one of bindings that do not hold.

It may be instructive to elaborate a bit on this. Let us, for a moment, think of $\mathcal{W}$ as the generalization of a *rewrite* interpreter: when it unfolds a function call it replaces the call by the body of the called function and *substitutes* the current arguments into the term being interpreted. Alternatively, one can think of an *environment based* interpreter which creates bindings of the formal parameters to the actual arguments. Correspondingly, one could imagine an environment based version of $\mathcal{W}$. With respect to positive information, this is basically what the supercompiler in [Glu93a] is.

Let us see this in some detail. In the language in [Glu93a], called *S-graph*, all equality tests (performed at run-time, not just occurring syntactically) must be on atoms (0-ary constructors.) This means that the information that an equality test is assumed to fail (during transformation) amounts to saying that a variable is not equal to another variable or atom. Thus both positive *and* negative information arising from equality tests is easy to maintain, using two environment lists of positive and negative bindings.

In S-graph there are no definitions by patterns. Instead there are only *atoms* and one *constructor*, **CONS**, along with a construct ($\mathbf{if}\ (\mathbf{CONS?}\,x\,h\,t)\,t_1\,t_2$). This construct tests whether x is a cons-cell, and if so proceeds with evaluation of t_1 with variable h and t bound to the head and tail of x, respectively; otherwise evaluation proceeds with t_2. Here there is both positive and negative information as the result. Roughly, the binding $x \mapsto \mathbf{CONS}\,h\,t$ is added to the positive environment's information in the true branch and to the negative environment's in the false branch. The action in the true branch is very similar to the action of the case rule considered at the end of Section 3.6.

Thus, the positive supercompiler and the supercompiler in [Glu93a] are identical with respect to the propagation of positive information, except that the former uses substitution and the latter environments.

There is an advantage of the latter over the former pertaining to the propagation of negative information: one can maintain negative information as bindings which do not hold; such information can apparently not be represented by substitutions.

There does not seem to be any significant difference between using environments or susbstitution for positive information.[3]

9.6 Conclusion

We have considered deforestation, partial evaluation, generalized partial computation, supercompilation, positive supercompilation, and interpretation and partial evaluation of Prolog.

Deforestation and partial evaluation are similar in that they propagate neither positive nor negative information, only known argument values. Positive supercompilation and generalized partial computation, as described by Takano, are similar in that the both propagate positive, but not negative, information. Supercompilation as described by Turchin, or Glück and Klimov, is similar to generalized partial computation as described by Futamura and Nogi in that both propagate both positive and negative information.

The amount of information propagation is significant: transformers with positive information propagation, *e.g.* positive supercompilation, generalized partial computation, and supercompilation as described by Turchin, or Glück and Klimov, can pass the KMP test. For transformers without positive information propagation, *e.g.* partial evaluation and deforestation, the object program must be rewritten to explicitly manipulate the positive information arising from test in order for the transformer to be able to pass the KMP test. Also, transformers with positive information propagation perform better in problem solving and theorem proving than those without.

It will be seen though that the problem of ensuring termination of these transformers is harder than for those with simpler information propagation.

Deforestation, positive supercompilation and supercompilation as described by Turchin is similar in the handling of nested calls. Partial evaluation and a call-by-value version of generalized partial computation would also be similar in the handling of nested calls.

The handling of nested calls is significant: deforestation, positive supercompilation and supercompilation as described by Turchin can eliminate intermediate data structures and turn multiple-pass programs into one-pass programs. This cannot be done, at least not to the same extent, by partial evaluation and call-by-value generalized partial computation.

We can summarize the main results in the following table, where "cbv" and "cbn" refer to the transformer's simulation of evaluation.

	cbv	cbn
no info.	partial evaluation	deforestation
pos.info.		positive supercompilation
pos. and neg. info	gpc (Futamura)	supercompilation (Turchin)

Here we have left out Takano's version of gpc, since it does not give a "true" call-by-name treatment of nested calls although it is stated for a call-by-name language, and we have left out Glück and Klimov's simple supercompiler since it deals with a flat language. Finally, it should be noted that Turchin's supercompiler is for a call-by-value language, but simulates call-by-name evaluation, and so it has been placed under call-by-name.

[3]In self-application of partial evaluation one does binding-time analysis; such an analysis gives better results for the environment-based version because it gives better separation of static and dynamic data.

Part III

Correctness

Chapter 10

Preservation of operational semantics

If this sentence is true then Santa Claus exists.

J. Barkley Rosser.[1]

If we transform the body of a function in some program using $\mathcal{W}$ and $\mathcal{W}$ terminates, then we would like the new function to be equivalent, in some sense, to the original. Section 10.1 develops two notions of term equality, one strictly stronger than the other. Section 10.2 shows that $t = \mathcal{W}[\![\, t \,]\!]$ holds for the weaker definition, where $\mathcal{W}$ denotes the driving and folding elements of the positive supercompiler. Section 10.3 shows that the pre- and postphases of the positive supercompiler, *viz.* generalizing and postunfolding, preserves semantics in the same sense. All of these proofs can be extended to the stronger notion at the expense of some complications, which we do not get into details with. Section 10.4 summarizes work on related correctness theorems.

10.1 A discussion on equal terms

Informally, two terms are equal if, for every application they compute the same result. Here we must be precise as to what is meant by an application and what is meant by a result.

Example **10.1.1** (Adopted from [Run89].) Consider the functions i and i'

$$\begin{array}{lcl} i\; ns & \leftarrow & l\; p\; ns \\ l\; [\,]\; x & \leftarrow & [\,] \\ l\; (h : t)\; x & \leftarrow & h : l\; x\; t \\ p & \leftarrow & C : p \end{array}$$

$$\begin{array}{lcl} i'\; [\,] & \leftarrow & [\,] \\ i'\; (x : xs) & \leftarrow & C : x : i'\; xs \end{array}$$

Are i and i' equal? Well, it is easy to see that for any constant b, it holds that $\mathcal{I}[\![\, i\; b \,]\!] \Rightarrow b'$ and $\mathcal{I}[\![\, i'\; b \,]\!] \Rightarrow b'$ for some constant b'. However, when the argument contains non-terminating computations, the two functions differ. For instance, let Ω be the 0-ary function defined as $\Omega \leftarrow \Omega$. Clearly, both interpretation trees $\mathcal{T}_{\mathcal{I}}[\![\, i\; \Omega \,]\!]$ and $\mathcal{T}_{\mathcal{I}}[\![\, i'\; \Omega \,]\!]$ are infinite, but there is a small difference: the former has a node with a term with the outermost constructor C. This means that if we run the programs on the Miranda system, then the first prints out the constructor C and then enters an infinite loop, whereas the second enters the infinite loop without ever printing out anything. □

[1] See: Raymond Smullyan, *What Is the Name of This Book?* Penguin Books, 1978, p215.

From this example three definitions of equality on terms t_1, t_2 arise, which we first review informally, and then express in notation. (i) if t_1, t_2 are both ground, then they are *groundly equal* if they compute the same (finite or infinite) result. (ii) t_1, t_2 are *weakly equal* if they compute the same (finite or infinite) result when we substitute constants for variables, provided that for every v, the constant substituted for v in t_1 is the same as that substituted for v in t_2. (iii) t_1, t_2 are *strongly equal* if, whenever we substitute computations for variables, both terms compute the same (finite or infinite) result provided that for every v, the computation substituted for v in t_1 gives the same (finite or infinite) result as that substituted for v in t_2.

We first introduce some notions that allows us to compare the results of infinite computations.

DEFINITION **10.1.2** Given a (possibly infinite) interpretation tree I. For a node N define the set $\mathcal{R}(N)$ by induction as follows. (i) if N contains $c\, t_1 \dots t_k$ and N has children $N_1 \dots N_k$, then $\mathcal{R}(N) = \{\bot\} \cup \{cx_1 \dots x_k | x_i \in \mathcal{R}(N_i)\}$. (ii) if N contains $ct_1 \dots t_k$ and N has no children, then $\mathcal{R}(N) = \{\bot\} \cup \{ct_1 \dots t_k\}$.

(iii) otherwise N has one child and $\mathcal{R}(N) = \{\bot\} \cup \mathcal{R}(N_1)$. Now define the *value set* of I as $\mathcal{R}(M)$ where M is the root of I.

If I, I' are two interpretation trees with the same value sets we write $I \simeq I'$. □

Generalized induction is described in detail in [Acz77]. The reader should understand that something is element in the value set of an interpretation tree if it follows by a finite number of applications of $(i) - (iii)$ in the preceding definition.

Informally $I \simeq I'$ for two interpretation trees, if the set of finite approximations of the two results are the same.

The following proposition gives the connection to the value of an interpretation tree I, when I is finite.

PROPOSITION **10.1.3** *Define $|\bot| = 0$, and let I be a finite interpretation tree. The largest element of the value set of I is the same as the value of I (according to the measure in Section 2.2).* □

PROOF: Obvious. □

We now define our notions of equality.

DEFINITION **10.1.4** (Equality) For two ground terms t, t' define $t =_g t'$ (t and t' are groundly equal) to mean: $\mathcal{T}_\mathcal{I}[\![\, t \,]\!] \simeq \mathcal{T}_\mathcal{I}[\![\, t' \,]\!]$.

For two terms t, t' possibly containing variables define $t =_w t'$ (t and t' are weakly equal) to mean: $t\theta =_g t'\theta$ for all constant substitutions θ.

Define two ground substitutions θ, θ' to be *similar* if $v\theta =_g v\theta'$ for all variables v. For two terms t, t' possibly containing variables define $t =_s t'$ (t and t' are strongly equal) to mean: $t\theta =_g t'\theta'$ for all similar substitutions θ, θ'. □

The following proposition states the relationship between the different notions of termination.

PROPOSITION **10.1.5** (i) *if t, t' are ground then $t =_s t'$ iff $t =_w t'$ iff $t =_g t'$.* (ii) *$t =_s t'$ implies $t =_w t'$.* (iii) *there are terms which are weakly, but not strongly, equal.* □

PROOF: (i) and (ii) are obvious. The two terms $i\ ns$ and $i'\ ns$ provide an example of terms that are weakly, but not strongly, equal. □

It should be noted that even weak equality is a fairly strong property. In Section 9.5 we shall see that Turchin's supercompiler can extend the domain of functions in the sense that it may be the case that the original function loops infinitely on some constant b and the residual function returns a value when applied to b. Obviously, this possibility is ruled out even if the output of the transformer is weakly equal to the input. Strong equality in addition states that the *strictness properties* of the two terms are the same in the sense that applying one term to an infinite computation yields the same (finite or infinite) result as applying the other term to the same infinite computation.

We shall henceforth be concerned only with weak equality, although the author would claim that the results proved below hold for strong equality too. These extended results rely, however, on certain stronger results than those required to prove the results for weak equality, which the author has not proved rigorously.

10.2 Driving and folding preserves operational semantics

We show that the output of $\mathcal{W}$ is weakly equal to the input, provided that $\mathcal{W}$ terminates and the input are $M_{1/2}$ terms and programs.

The following lemma is used in the induction steps of the overall theorem and may clarify the induction principle used.

LEMMA **10.2.1** *Given ground terms t, t' and programs p, p'. Let $I = \mathcal{T}_{\mathcal{I}}[\![\, t \,]\!]$ using the program p, and $I' = \mathcal{T}_{\mathcal{I}}[\![\, t' \,]\!]$ using the program p'. Assume that the root of I has children $N_1 \dots N_k$ and that the root of I' has children $N'_1 \dots N'_k$. Let I_i, I'_i be the trees with root N_i, N'_i, respectively. Assume that $I_i \simeq I'_i$ for $i = 1 \dots k$. Assume finally that either none or both of t, t' have an outermost constructor, and if they have one that it is the same. Then $I \simeq I'$.* □

PROOF: Proof by induction on the definition of value set above, see [Acz77] if necessary. □

PROPOSITION **10.2.2** *For all t, if $\mathcal{W}[\![\, t \,]\!]$ terminates then $t =_w \mathcal{W}[\![\, t \,]\!]$, where the left hand side uses definitions from the original program and the right hand side uses the new definitions computed by $\mathcal{W}$.*
□

PROOF: By induction on the definition of $\mathcal{W}$ (we proceed in considerable detail, but see [Acz77] if necessary.) In each case we are to show for the term t occurring as argument to $\mathcal{W}$ on the left hand side of $\Rightarrow$ that $t =_w \mathcal{W}[\![\, t \,]\!]$, where t uses the original definitions and $\mathcal{W}[\![\, t \,]\!]$ uses the new definitions. That is: that for any ground substitution θ that $t\theta =_g \mathcal{W}[\![\, t \,]\!]\theta$ *i.e.* that $\mathcal{T}_{\mathcal{I}}[\![\, t\theta \,]\!] \simeq \mathcal{T}_{\mathcal{I}}[\![\, \mathcal{W}[\![\, t \,]\!]\theta \,]\!]$.

We prove this by showing that $\mathcal{N}_{\mathcal{I}}[\![\, t\theta \,]\!] = [t_1 \dots t_n], \mathcal{N}_{\mathcal{I}}[\![\, \mathcal{W}[\![\, t \,]\!]\theta \,]\!] = [t'_1 \dots t'_n]$ where $\mathcal{T}_{\mathcal{I}}[\![\, t_i \,]\!] \simeq \mathcal{T}_{\mathcal{I}}[\![\, t'_i \,]\!]$ and apply the preceding lemma.

In each case the induction hypothesis says that for all the applications $\mathcal{W}[\![\, t \,]\!]$ on the right hand side of $\Rightarrow$, $t =_w \mathcal{W}[\![\, t \,]\!]$ using original definitions for t and new ones for $\mathcal{W}[\![\, t \,]\!]$, *i.e.* that for all ground substitutions σ, $t\sigma =_g \mathcal{W}[\![\, t \,]\!]\sigma$, *i.e.* $\mathcal{T}_{\mathcal{I}}[\![\, t\sigma \,]\!] \simeq \mathcal{T}_{\mathcal{I}}[\![\, \mathcal{W}[\![\, t \,]\!]\sigma \,]\!]$.

So, given ground substitution θ.

Clause (0). We must show that

$$\mathcal{T}_{\mathcal{I}}[\![\, v\theta \,]\!] \simeq \mathcal{T}_{\mathcal{I}}[\![\, v\theta \,]\!]$$

This is true by the definition of $\simeq$.

Clause (1a). This case is similar to clause (0).

Clause (1b). The induction hypothesis states

$$\mathcal{T}_{\mathcal{I}}[\![\, t_i\theta \,]\!] \simeq \mathcal{T}_{\mathcal{I}}[\![\, \mathcal{W}[\![\, t_i \,]\!]\theta \,]\!]$$

and we must show

$$\mathcal{T}_{\mathcal{I}}[\![\, (c\, t_1 \dots t_n)\theta \,]\!] \simeq \mathcal{T}_{\mathcal{I}}[\![\, (c\, \mathcal{W}[\![\, t_1 \,]\!] \dots \mathcal{W}[\![\, t_n \,]\!])\theta \,]\!]$$

We have:

$$\begin{aligned} \mathcal{N}_{\mathcal{I}}[\![\, (c\, t_1 \dots t_n)\theta \,]\!] &= \mathcal{N}_{\mathcal{I}}[\![\, c\, (t_1\theta) \dots (t_n\theta) \,]\!] \\ &= [t_1\theta, \dots, t_n\theta] \end{aligned}$$

and

$$\begin{aligned} \mathcal{N}_{\mathcal{I}}[\![\, (c\, \mathcal{W}[\![\, t_1 \,]\!] \dots \mathcal{W}[\![\, t_n \,]\!])\theta \,]\!] &= \mathcal{N}_{\mathcal{I}}[\![\, c\, (\mathcal{W}[\![\, t_1 \,]\!]\theta) \dots (\mathcal{W}[\![\, t_n \,]\!]\theta) \,]\!] \\ &= [\mathcal{W}[\![\, t_1 \,]\!]\theta, \dots, \mathcal{W}[\![\, t_n \,]\!]\theta] \end{aligned}$$

and Lemma 10.2.1 applies.

Clause (2). The induction hypothesis states that

$$\mathcal{T}_{\mathcal{I}}[\![\ (e[t^f\{v_i^f := t_i\}_{i=1}^n])\theta\ ]\!] \simeq \mathcal{T}_{\mathcal{I}}[\![\ \mathcal{W}[\![\ e[t^f\{v_i^f := t_i\}_{i=1}^n]\ ]\!]\theta\ ]\!]$$

We are to prove

$$\mathcal{T}_{\mathcal{I}}[\![\ (e[f\, t_1 \ldots t_n])\theta\ ]\!] \simeq \mathcal{T}_{\mathcal{I}}[\![\ (f^\square\ u_1 \ldots u_k)\theta\ ]\!]$$

where

$$f^\square\ u_1 \ldots u_k \leftarrow \mathcal{W}[\![\ e[t^f\{v_i^f := t_i\}_{i=1}^n]\ ]\!]$$

We have:

$$\begin{array}{rcl} \mathcal{N}_{\mathcal{I}}[\![\ (e[f\, t_1 \ldots t_n])\theta\ ]\!] & = & \mathcal{N}_{\mathcal{I}}[\![\ (e\theta)[(f\, t_1 \ldots t_n)\theta]\ ]\!] \\ & = & \mathcal{N}_{\mathcal{I}}[\![\ (e\theta)[f\ (t_1\theta) \ldots (t_n\theta)]\ ]\!] \\ & = & [(e\theta)[t^f\{v_i^f := t_i\theta\}_{i=1}^n]] \\ & = & [(e\theta)[(t^f\{v_i^f := t_i\}_{i=1}^n)\theta]] \\ & = & [(e[t^f\{v_i^f := t_i\}_{i=1}^n])\theta] \end{array}$$

and

$$\begin{array}{rcl} \mathcal{N}_{\mathcal{I}}[\![\ (f^\square\ u_1 \ldots u_k)\theta\ ]\!] & = & \mathcal{N}_{\mathcal{I}}[\![\ f^\square\ (u_1\theta) \ldots (u_k\theta)\ ]\!] \\ & = & [\mathcal{W}[\![\ e[t^f\{v_i^f := t_i\}_{i=1}^n]\ ]\!]\{u_j := u_j\theta\}_{j=1}^k] \\ & = & [\mathcal{W}[\![\ e[t^f\{v_i^f := t_i\}_{i=1}^n]\ ]\!]\theta] \end{array}$$

and Lemma 10.2.1 applies.

Clause (3a). Similar to clause (2).

Clause (3b). The induction hypothesis states that for any θ^*

$$\mathcal{T}_{\mathcal{I}}[\![\ ((e[t^{g,c_j}\{v^{g,c_j} := t_i\}_{i=1}^n])\{v := p_j\})\theta_*\ ]\!] \simeq \mathcal{T}_{\mathcal{I}}[\![\ \mathcal{W}[\![\ (e[t^{g,c_j}\{v^{g,c_j} := t_i\}_{i=1}^n])\{v := p_j\}\ ]\!]\theta_*\ ]\!]$$

We are to prove

$$\mathcal{T}_{\mathcal{I}}[\![\ (e[g\ v\, t_1 \ldots t_n])\theta\ ]\!] \simeq \mathcal{T}_{\mathcal{I}}[\![\ (g^\square\ v u_1 \ldots u_k)\theta\ ]\!]$$

where

$$g^\square\ p_j\ u_1 \ldots u_k \leftarrow \mathcal{W}[\![\ e[t^{c_j}\{v_i^{g,c_j} := t_i\}_{i=1}^n]\{v := p_j\}\ ]\!]$$

where p_j, t^c are renamings of the corresponding pattern and body of g and $FV(p_j) = \{x_1 \ldots x_m\}$ is disjoint with $e[g\ v\, t_1 \ldots t_n]$.

Let $v\theta \equiv c_j\ b_1 \ldots b_m$, $\theta_* = \{x_k := b_k\}_{k=1}^m \circ \theta$. We now have:

$$\begin{array}{rcl} \mathcal{N}_{\mathcal{I}}[\![\ (e[g\ v\, t_1 \ldots t_n])\theta\ ]\!] & = & \mathcal{N}_{\mathcal{I}}[\![\ (e\theta)[g\ (c\, b_1 \ldots b_m)\ (t_1\theta) \ldots (t_n\theta)])\ ]\!] \\ & = & [(e\theta)[t^{g,c_j}\{v_i^{g,c_j} := t_i\theta\}_{i=1}^n\{v_{k+n}^{g,c_j} := b_k\}_{k=1}^m]] \\ & = & [(e\theta)[t^{c_j}\{v^{g,c_j} := t_i\theta\}_{i=1}^n\{x_k := b_k\}_{k=1}^m]] \\ & = & [(e\theta_*)[((t^{c_j}\{v^{g,c_j} := t_i\}_{i=1}^n)\{v := p_j\})\theta_*] \\ & = & [((e[t^{c_j}\{v^{g,c_j} := t_i\}_{i=1}^n])\{v := p_j\})\theta_*] \end{array}$$

and

$$\begin{array}{rcl} \mathcal{N}_{\mathcal{I}}[\![\ (g^\square\ v\ u_1 \ldots u_k)\theta\ ]\!] & = & \mathcal{N}_{\mathcal{I}}[\![\ (g^\square\ (c_j\ b_1 \ldots b_m)\ (u_1\theta) \ldots (u_k\theta)\ ]\!] \\ & = & [\mathcal{W}[\![\ (e[t^{c_j}\{v^{g,c_j} := t_i\}_{i=1}^n])\{v := p_j\}\ ]\!]\{x_j := b_j\}_{k=1}^m\{u_i := (u_i\theta)\}_{i=1}^n] \\ & = & [\mathcal{W}[\![\ (e[t^{c_j}\{v^{g,c_j} := t_i\}_{i=1}^n])\{v := p_j\}\ ]\!]\theta_*] \end{array}$$

and Lemma 10.2.1 applies.

Clause (4a). We have that $b \equiv b$ are ground, and the induction hypothesis states that

$$\mathcal{T}_{\mathcal{I}}[\![\ (e[t])\theta\ ]\!] \simeq \mathcal{T}_{\mathcal{I}}[\![\ \mathcal{W}[\![\ t\ ]\!]\theta\ ]\!]$$

We are to prove

$$\mathcal{T}_{\mathcal{I}}[\![\ (e[b = b\ \rightarrow\ t \mathbin{\square} t])\theta\ ]\!] \simeq \mathcal{T}_{\mathcal{I}}[\![\ \mathcal{W}[\![\ t\ ]\!]\theta\ ]\!]$$

We have

$$\begin{array}{rcl} \mathcal{N}_{\mathcal{I}}[\![\, (e[b = b' \;\rightarrow\; t \,\square\, t'])\theta \,]\!] & = & \mathcal{N}_{\mathcal{I}}[\![\, (e\theta)[b = b' \;\rightarrow\; t\theta \,\square\, t'\theta] \,]\!] \\ & = & [(e\theta)[t\theta]] \\ & = & [(e[t])\theta] \end{array}$$

And a minor modification of Lemma 10.2.1 applies.

Clause (4b). Similar to clause (4a).

Clause (4c). The induction hypothesis states that

$$\mathcal{T}_{\mathcal{I}}[\![\, t\theta \,]\!] \simeq \mathcal{T}_{\mathcal{I}}[\![\, \mathcal{W}[\![\, (e[t])\sigma \,]\!]\theta \,]\!]$$

We are to prove

$$\mathcal{T}_{\mathcal{I}}[\![\, (e[b = b' \;\rightarrow\; t \,\square\, t'])\theta \,]\!] \simeq \mathcal{T}_{\mathcal{I}}[\![\, (b = b' \;\rightarrow\; \mathcal{W}[\![\, (e[t])\sigma \,]\!] \,\square\, \mathcal{W}e[t'])\theta \,]\!]$$

where $\sigma := MGU(b, b')$, with $\sigma = \{v_i := t_i\}_{i=1}^{m}, 0 \leq m$ or $\sigma = fail$. We consider two cases the cases (i) $b\theta \equiv b'\theta$, (ii) $b\theta \not\equiv b'\theta$.

Case (i) . Similarly to previous cases we have

$$\mathcal{N}_{\mathcal{I}}[\![\, (e[b = b' \;\rightarrow\; t \,\square\, t'])\theta \,]\!] \;=\; [(e[t]\theta)]$$

and

$$\mathcal{N}_{\mathcal{I}}[\![\, (b = b' \;\rightarrow\; \mathcal{W}[\![\, (e[t])\sigma \,]\!] \,\square\, \mathcal{W}e[t'])\theta \,]\!] \;=\; [\mathcal{W}[\![\, (e[t]\sigma) \,]\!]\theta]$$

Now, if only we can show that $(e[t])\theta \equiv ((e[t])\sigma)\theta$, then the induction hypothesis yields the desired.

Since $b\theta \equiv b'\theta$, b, b' are unifiable, and σ is an MGU. Since θ is a unifier there exists a σ' such that for all t, $t\theta \equiv (t\sigma)\sigma'$. Since σ is chosen to be idempotent with respect to composition, it holds that

$$((e[t])\sigma)\theta \equiv (((e[t])\sigma)\sigma)\sigma' \equiv ((e[t])\sigma)\sigma' \equiv (e[t])\theta$$

as desired.

Case (ii) . This case poses no problems.

Clause (5). This clause is not in $M_{1/2}$. □

REMARK **10.2.3** Note that in every case except (4a),(4b) the number of steps we have to compute in the original and resiudal program to get the result are the same. We return to this in the next chapter. □

10.3 Generalizing and postunfolding preserves semantics

That generalizations do not disturb the semantics of a program and term is the contents of the following proposition.

PROPOSITION **10.3.1** *For all* e, t, v, $e(t) =_w$ *let* $v = t$ *in* $e(v)$. □

PROOF: Obvious since the latter reduces to the former in one step of $\mathcal{I}$. □

That postunfolding does not disturb the semantics of a program and term is the contents of the following proposition.

PROPOSITION **10.3.2** *For all* $e, t_1, \ldots, t_n, t^f$, $e[f\, t_1 \ldots t_n] =_w e[t^f\{v_i^f := t_i\}_{i=1}^{n}]$ □

PROOF: Obvious since the latter reduces to the former in one step of $\mathcal{I}$. □

10.4 Related work

Rigorous proofs of preservation of semantics are rare in the functional programming community. For instance, it seems to be commonly accepted that the deforestation algorithm preserves semantics, although we have not seen a rigorous proof of that fact.

It is observed in [Bur77] that unrestricted folding can also change semantics by introducing nontermination. The standard example is when a definition $f\ x \leftarrow f'\ x$ is folded with itself to $f\ x \leftarrow f\ x$. That problem does not arise here, since we only fold calls to newly introduced functions.

It is observed in [Run89] that instantiations in the fold/unfold framework can change the semantics. For instance, the second of the two programs in Example 10.1.1 can be derived from the first by manual fold/unfold methods. However, this does not, as we have shown, happen in $\mathcal{W}$. In fact, from the first of the programs $\mathcal{W}$ (and postunfolding) produces the following program which indeed is strongly equal to the input.

$$\begin{array}{lcl} i'\ ns & \leftarrow & C : j\ ns \\ j\ [] & \leftarrow & [] \\ j\ (x : xs) & \leftarrow & x : i'\ xs \end{array}$$

The point is that $\mathcal{W}$ does not perform premature instantiations.

Proofs of preservation of semantics, in a certain sense, for the transformation rules underlying Turchin's supercompiler are given in [Tur80a]. As already mentioned in Section 9.5, it is observed in [Tur80a] that the supercompiler can extend the domain of functions, thereby failing to satisfy the desired equivalence above; this is partly because the language Refal is call-by-value whereas the supercompiler for Refal simulates call-by-name evaluation. As was also mentioned in Section 9.5 these phenomena do not occur in the context of the positive supercompiler.

In the context of partial evaluation, the first correctness proof known to the author is that for λ-mix [Gom91]. More recently, the question has been addresed by others [Wan93,Pal93].

Chapter 11

On Efficiency and Size of Residual Programs

If I had had more time
I could have written you a shorter letter.

Blaise Pascal.

The applications of positive supercompilations in part II were concerned with the achievement of various effects. Two of these pertain to the improvement of the efficency of programs: specialization, in particular generation of efficient specialized matchers, and elimination of intermediate data structures. In Chapter 5 we charaterized syntactically the output of $\mathcal{W}$ to show that all constants and intermediate data structures were eliminated, thereby suggesting that the output program is more efficient than the input program. In this chapter we take an abstract approach to the notion of efficiency in terms of the number of steps executed by the rewrite interpreter. We shall also consider informally a notion of efficiency related to lazy evaluation.

Section 11.1 describes what it is in the positive supercompiler that reduces the number of steps in the output program compared to the input program. It also defines run-time complexity in terms of number of steps taken by $\mathcal{I}$ and shows that the positive supercompiler including postunfolding always yields only linear speedups, in a certain sense. The development in this section is inspired by a similar development for partial evaluation in [Jon93, Sections 6.1.1 and 6.2], although the present proof is somewhat more concrete.

Measuring efficiency by a call-by-name interpreter misses the point of sharing. Section 11.2 discusses the efficiency of the output of $\mathcal{W}$ in terms of a lazy evaluator. Section 11.3 discusses on-line onfolding versus postunfolding. Section 11.4 is concerned with the question of the size of residual programs. It should be noted that these sections add nothing new to $\mathcal{W}$; they describe certain well-known problems and solutions that one might adopt.

The last three sections are inspired by [Ses88a].

11.1 Linearity of speedups

There are certain automatic, termination-safe program transformations which can change a program with exponential run-time to a program with linear run-time. For instance, Chin's *tupling* transformation [Chi93b,Chi93c,Chi93d] turns the naive, exponential Fibonacci function into the well-known, more clever, linear Fibonacci function.

The question was posed in [Jon90] whether partial evaluators using only the mechanisms of *definition, folding, unfolding,* and *instantiation* could achieve such effects of *superlinear* speedups. The question was

answered negatively in [And92], see [Jon93, Section 6.2].

We have seen that the positive supercompiler is more powerful than partial evaluation, and it is therefore natural to ask the same question for the positive supercompiler. This section answers the question negatively.

The remainder of the section proceeds as follows. First we make precise the notions of run-time and speedup. Then we give a slightly modified version of the positive supercompiler, equivalent to the original. Finally we give precise correspondences between the run-times of the original and residual program, showing in particular that all speedups are linear.

DEFINITION **11.1.1** Let $\mathcal{I}', \mathcal{T}_{\mathcal{I}'}$ be as $\mathcal{I}, \mathcal{T}_{\mathcal{I}}$ except that clause (1a) and the condition in (1b) is left out.

Let the *alternative* positive supercompiler be the following. The driving part is the same as in the positive supercompiler, except that clauses (4a),(4b) are left out and the restriction that b or b' be non-ground is left out from clause (4c). The folding scheme is the same as in the positive supercompiler. The postunfolding phase is the same as that of the positive supercompiler except that all terms of form $b = b' \;\rightarrow\; t \,\Box\, t'$ where b, b' are ground are reduced. □

DEFINITION **11.1.2** Let t, p be such that $\mathcal{I}'[\![\, t \,]\!]$ terminates. Define $\mathcal{C}[\![\, t \,]\!]$, the *run-time* of t, to be the number of nodes in the interpretation tree $\mathcal{T}_{\mathcal{I}'}[\![\, t \,]\!]$. □

REMARK **11.1.3** We have already mentioned several times the intention that every step of $\mathcal{N}_{\mathcal{I}}$ represents a step of $\mathcal{I}$, that is: that the number of steps executed by $\mathcal{I}$ on a ground term t to reach the result is the number of nodes in $\mathcal{T}_{\mathcal{I}}[\![\, t \,]\!]$. The intention with $\mathcal{I}'$ and $\mathcal{T}_{\mathcal{I}'}$ is the same. This justifies the definition of run-time. □

PROPOSITION **11.1.4** *For any term t and program p, let t', p' be the result of applying the transformation phase and postprocessing phase of the positive supercompiler, and t'', p'' the result of applying the transformation phase and postprocessing phase of the alternative positive supercompiler. Then $t \equiv t', p \equiv p'$.* □

PROOF: Obvious. □

PROPOSITION **11.1.5** *For a $M_{1/2}$ term t and program p, assume that $\mathcal{W}'[\![\, t \,]\!]$ (the transformation phase) terminates with term and program t', p'. For every ground substitution θ, $\mathcal{C}[\![\, t\theta \,]\!] = \mathcal{C}[\![\, t'\theta \,]\!]$.* □

PROOF: Induction on the definition of $\mathcal{W}'$ like the proof of Proposition 10.2.2. □

Informally, the proposition is true because the result in every step returned by $\mathcal{W}'$ has the same form as the argument. The result of transforming a conditional is a conditional, the result of transforming an f-function is an f-function. (The result of transforming a g-function may be an f-function.)

The proposition shows that the run-time of any application of the residual term to constants is *exactly the same* as the run-time of the original term. This should not lead the reader to believe that positive supercompilation is no good; far from it. For instance, all intermediate data structures are gone, and such structures take up space and garbage collection time in actual implementations. Also, the postunfolding phase reduces away some of the steps that the interpreter would otherwise have to execute of residual programs. As we saw in Chapter 8, this is in fact what gives the good complexity of specialized matchers.

PROPOSITION **11.1.6** *Let t, p be $M_{1/2}$ treeless term and program, and suppose that the postunfolding phase of the alternative positive supercompiler yields t', p'. Then there is an integer constant a such that for all ground substitutions θ, $a\mathcal{C}[\![\, t'\theta \,]\!] \geq \mathcal{C}[\![\, t\theta \,]\!]$.* □

PROOF: In the original program and term, call conditionals that become postunfolded *u-conditionals*, and similarly we have *u-calls* to f-functions which become post-unfolded. Let a be the number of conditionals and f-calls unfolded.

Now consider a "run" of the original program $\mathcal{N}_{\mathcal{I}'}[\![t\theta]\!]$ and of the new program $\mathcal{N}_{\mathcal{I}'}[\![t'\theta]\!]$. Since we are dealing with $M_{1/2}$ treeless terms and programs, and since the run substitutes only constants for variables, the form of the terms in the run of the original program is:

$$\begin{array}{lll} t & ::= & c\, t_1 \ldots t_n \mid f\, b_1 \ldots b_n \mid g\, b_0 \ldots b_n \mid b = b' \rightarrow t \,\square\, t' \\ b & ::= & c\, b_1 \ldots b_n \end{array}$$

The original interpretation tree can be turned into a number of disjoint sequences as follows. (i) the sequence from the root to the first branching, if any, and to the end node otherwise, is a sequence. If there were a branching in a node N with children $N_1 \ldots N_k$ recursively compute for every subtree with root N_i a set of sequences (these do not contain the sequence from the root as a prefix).

Similarly we can compute a number of sequences for the interpretation tree for the residual program. Now, because $\mathcal{I}'$ always goes to the children of a constructor, the number of sequences in the two trees are the same. It therefore suffices to show that for every seqeunce of length n in the interpretation tree for the new program, the corresponding sequence in the original tree is at most $a \cdot n$ long. This amounts to showing that any sequence in the original tree can pass through at most a consecutive terms that are either u-conditionals or u-calls.

This can be proven from the postunfolding scheme and the grammar above. □

REMARK **11.1.7** The reasoning here is essentially the same as the one in [Jon93, section 6.2]. The main difference is that we do not consider the abstract notion of *the number of steps in which the transformer terminates*, but the more concrete number of postunfolded conditionals and f-functions. □

THEOREM **11.1.8** *(Linearity of speedups.) For any $M_{1/2}$ term and program t, p the result of applying the transformation phase and the postunfolding phase of the positive supercompiler yields, if it terminates, term and program t', p' such that there is an integer constant a such that for all ground substitutions θ, $a\mathcal{C}[\![t'\theta]\!] \geq \mathcal{C}[\![t\theta]\!]$.* □

PROOF: Compose Propositions 11.1.4,11.1.5, and 11.1.6. □

We close the section with an amusing result.

PROPOSITION **11.1.9** *(Idempotence of $\mathcal{W}$.) If for term t and program p $\mathcal{W}$ terminates with t', p', then the result of running $\mathcal{W}$ on t', p' is t', p'.* □

PROOF: The output of $\mathcal{W}$ is M_1 treeless term and program. Now prove by induction on the definition of $\mathcal{W}$ that $\mathcal{W}$ is the identity on M_1 treeless terms and programs. □

11.2 The problem of duplicated computation

The output of $\mathcal{W}$ may be *less* efficient than the output under a lazy evaluation semantics. This is the subject of the present section.

We first describe the problem in positive supercompilation and a framework for solutions. Then we describe the problem in partial evaluations and the solutions that have been adopted in that field. We close the section with considerations on another, related problem in partial evaluation and another, related effect in positive supercompilation.

The problem in lazy semantics and a framework for solutions

The following example is essentially taken from [Ses88a].

EXAMPLE **11.2.1**

$$\begin{array}{lcl} & & g\ z \\ g\ Nil & = & Leaf\ L \\ g\ (Cons\ x\ xs) & = & f\ (g\ xs) \\ f\ w & = & Branch\ w\ w \end{array}$$

This rather contrived program turns a list of length n into a tree of depth n. Each leaf contains a 0-ary constructor L. The result of running $\mathcal{W}$ on this program is:

$$\begin{array}{lcl} & & g'z \\ g'\ Nil & = & Leaf\ L \\ g'\ (Cons\ x\ xs) & = & Branch\ (g'\ xs)\ (g'\ xs) \end{array}$$

Now, under a call-by-value semantics, the former program has run-time linear in the length of the list, while the latter has run-time exponential in the length of the list. Under a call-by-name semantics like that of our simple interpreter from Section 2.4, both programs have running-time exponential in the length of the list. However, under a lazy implementation of the call-by-name semantics, using *e.g.* graph reduction, the run-times are as in the case of call-by-value. This means that the output of $\mathcal{W}$ is grossly inefficient compared to the original. Obviously, the problem is that the computation of $g\ xs$ has been duplicated; or, equivalently, that the *sharing* of the calls to $g\ xs$ has been lost.

To prevent the duplication of the recursive call of g to itself we generalize f's argument, see Section 3.3.

$$\begin{array}{lcl} & & g\ z \\ g\ Nil & = & Leaf\ L \\ g\ (Cons\ x\ xs) & = & \text{let } v = g\ xs \text{ in } f\ v \\ f\ w & = & Branch\ w\ w \end{array}$$

Given this program as input $\mathcal{W}$ returns

$$\begin{array}{lcl} & & g\ z \\ g\ Nil & = & Leaf\ L \\ g\ (Cons\ x\ xs) & = & \text{let } v = g\ xs \text{ in } Branch\ v\ v \end{array}$$

□

How do we discover when to generalize, as we generalized f's argument above? Well, in this case, the problems arise from the fact that f is *non-linear* in w, *i.e.* has *multiple* occurrences of w in its body, and is called with an argument containing a function call.

Actually, it is not hard to see that *if* the problem occurs, then there must be some function h non-linear in a variable u which is at some point called with an argument containing a call. It is also easy to see that agreeing that h's argument be generalized always solves the problem.

However, it may be the case that there is a non-linear function involved and yet no problem arises.

EXAMPLE **11.2.2** Consider the following function.

$$f\ x\ y\ z \leftarrow x = y\ \rightarrow\ (g\ z)\ \square\ (h\ z)$$

Here, only one of the two occurrences of z will be needed. In this case it is obvious that there is no risk of duplication of computation even if arguments for z contain calls. More complicated examples exist. □

Let us sum up the analysis conducted so far. The supercompiler simulates call-by-name evaluation, and it may happen that the supercompiler encounters a term $f\ t$ where t contains a function call. At that point the calls in t are not unfolded because we are simulating call-by-name. If t is duplicated it may happen that several of the copies of t give rise to the same function call in the residual program, and so we have duplicated a computation, like in the first example above. And the lazy evaluator would not duplicate the call, it would simply keep multiple references to one call and reduce that when required by one of the references.

An ambitious solution to the problem may conduct a *computation duplication risk analysis* of the source program p, discovering whether the problem will appear during transformation, and generalize accordingly. Below we sketch previous techniques of this kind for related program transformers.

Solutions for related transformers

A related problem of computation duplication occurs in partial evaluation. As mentioned in Section 9.2, partial evaluators such as Mix [Jon85,Jon87a] and Similix [Bon90b] usually simulate call-by-value evaluation. So one might think that whenever the partial evaluator encounters a term $f\ t$, then t does not contain any function calls, and hence no computation duplication can occur. This is however not the case. It may happen that calls in t lack information about the arguments, *e.g.* a call $g\ v$ where v is defined by patterns, and such calls cannot be unfolded, and must be left residual. So the partial evaluator may encounter terms $f\ t$ with t containing a call that will not subsequently be unfolded during partial evaluation. If f is non-linear the same problem as that described for $\mathcal{W}$ above can occur.

A static analysis to discover this problem for the partial evaluator *Mix* was described in [Ses88a] under the name *duplication analysis*. The technique works as follows. If there is a possibility that the partial evaluator will encounter a call $f\ t_1 \ldots t_{i-1}\ t\ t_{i+1} \ldots t_n$ where t contains a function call and where f is non-linear in its i'th variable, then special care must be taken. The information concerning the presence of calls in t is computed by a *call abstract interpretation*. The "special care" that must be taken consists in annotating the call to f to be residual. During partial evaluation, the call to f will be replaced by a call to f' where f' is a version of f specialized to the static arguments. The argument t, containing a residual call, is obviously dynamic. This is slightly different from the generalization technique described above. While the let may be considered as a residual call, any remaining arguments containing calls or variables ("dynamic arguments") are in fact bound to the formal parameters of the function. So generalization does not prevent the unfolding of the non-linear function, only the binding of the argument corresponding to the non-linear variable.

Note incidentally that in the partial evaluator $\mathcal{M}$ described in Section 9.2 calls were only unfolded if *all* arguments were passive terms, *i.e.* terms that do not contain computations, regardless of whether the function was or was not linear in parameters corresponding to arguments containing function calls. So $\mathcal{M}$ does not duplicate computations.

For the partial evaluator Similix there is a related analysis, called *abstract occurrence counting analysis*, which checks whether let-expressions can be unfolded at no risk of computation duplication [Bon90b]. In Similix all functions are transformed into the form $f\ v \leftarrow \text{let}\ v' = v\ \text{in}\ f\ v'$ so that all function calls can be unfolded without risk of computation duplication; only let-expressions need special treatment. This framework resembles the generalization framework: the technique in Similix allways insert let-expressions, but then unfolds the ones which are found by the abstract occurrence counting analysis to be unnecessary; the generalization technique above only inserts let-expressions which are found (by some analysis) to be necessary, and never unfolds them.

In deforestation the original approach by Wadler was to restrict application of deforestation to *treeless* programs, which in particular are linear [Wad88,Fer88]. Chin was the first to show how one could apply deforestation to all first-order programs [Chi90,Chi92b,Chi93a]. The problem discussed in this subsection is dealt with roughly by generalizing an argument of a function whenever the parameter has multiple occurrences in the body of the function. A more sophisticated technique has been described by G.W. Hamilton under the name of *usage count analysis* [Ham92a,Ham92b].

In Turchin's papers on supercompilation the solution is rather different. Recall that Turchin's supercompiler, like $\mathcal{W}$, simulates call-by-name. If, during the course of transformation, a term ft is encountered where f is non-linear and t contains a function call, the supercompiler suspends the call to f and transforms t first. [Tur86b, p317].[1] In simple cases, as *e.g.* if the call in t has form $f\ c$ where c does not contain calls and the right hand side of f does not contain calls, it is clear that after transforming t to t', transformation of the overall term $f\ t'$ can continue; what should happen in the case where t' contains calls is not clear from [Tur86b] p317.

[1] This is in harmony with the fundamental *on-line* paradigm underlying the whole supercompiler project. We shall see that techniques for ensuring termination are also applied on-line in Turchin's papers. From the philosophy of the project, on-line techniques do seem the most natural. A human does not sit down before applying a tool of reasoning to a new problem to consider all problems that may occur during the application of the tool; the problems are observed and solved during application of the tool. In applications of tools to other tasks than reasoning, *e.g.* the task of building a factory, this is of course no longer true.

Another, related problem in call-by-value transformers

It should be noted that the problem treated in this subsection is merely a problem of *inefficiency*. In transformers which simulate call-by-value, which partial evaluators usually are, there is also a problem of guaranteeing that the output of the transformer does not terminate more often than the original; this can happen if the unfolding of a function discards a non-terminating computation, which in turn may occur when the body of a function does not use one of its formal parameters. In Mix this problem was ignored ([Ses88a] p501). In Similix the solution was to check not only whether a variable does or does not occur more than once, but in fact to check that it occurs exactly once [Bon90b]; "Call unfolding should neither duplicate nor discard computation."

If it were not because $\mathcal{W}$ is designed carefully to avoid this problem, a similar problem could occur with conditionals in $\mathcal{W}$. Recall that conditionals are strict in variables in the test, *i.e.* for $t_1 = t_2 \rightarrow t_3 \square t_4$ the terms t_1, t_2 will be evaluated to constants. It is very tempting to transform a function like $f\ v \leftarrow v = v \rightarrow t \square t'$ to $f\ v \leftarrow t$. However, the latter right hand side terminates more often then the original. This is why b, b' are required to be ground in clause (4a),(4b) of $\mathcal{W}$.

$\mathcal{W}$ can eliminate the need for sharing

Example 11.2.1 showed that $\mathcal{W}$ can lose sharing. But there are also examples where $\mathcal{W}$ eliminates multiple occurrences of variables and thereby eliminates the *need* for sharing.

For instance, the function *match* in the tail-recursive general matcher is non-linear in both p and s, see Definition 8.1.2. This means that if it is called with a computation as argument for, say, s, then this computation must be shared by the different occurrences of s. However, in the residual program the multiple occurrences of s are gone. For instance, the right hand side of *match* with pattern AAB yields the term $loop_{AAB}\ s$ instead of $loop\ AAB\ s\ AAB\ s$, see Example 8.4.1. The multiple occurrences of s have been turned into just one occurrence, and so there is no need for sharing, and the lazy evaluator may save some bookkeeping.

11.3 The problem of excessive residual definitions

We first describe the problem of finding out when to introduce residual functions for the positive supercompiler. Then we describe solutions for certain partial evaluators.

The problem and the solution from Section 3.2

We noted in Section 3.2 that a residual f-function can be superfluous. This happens when the term which the residual f-function records is never encountered again. Ideally we should only introduce residual f-functions for terms that will be encountered more than once (all g-functions are necessary because M_1 does not have nested patterns or case-expressions.) This would prevent the same term from being encountered an infinite number of times. But we cannot know ahead whether a term will be encountered again.

However, *after* the transformation it is easy to figure out which functions were necessary. A very simple strategy is to put a mark on a residual function when a fold to that function is performed during transformation. After the transformation, all calls to residual functions that did not get a mark are unfolded. Since they did not get a mark, no fold was ever performed to any of them and therefore this post processing cannot proceed infinitely. This is what our postunfolding scheme does.

Note incidentally that the postunfolding phase does not duplicate computations in the style of the preceding section. In our context, postprocessing cannot duplicate compututations because of Proposition 5.1.4, which specifically states that the residual term and each residual function will have variables as arguments in all function calls, and this property is preserved by every step of the postunfolding phase.

We could also imagine postunfolding the let's. This may, however, duplicate computations as in the preceding section. To solve this problem, we could before postunfolding let's, perform the *computation duplication risk analysis* once more, or we could apply simpler on-line techniques.

Now it seems appropriate to address the question: does our conservative way of introducing residual functions affect the amount of inefficiency that $\mathcal{W}$ removes; that is, could we have obtained a more efficient program if we had introduced fewer residual functions.

Perhaps slightly surprisingly, the answer is *no.* To explain the point let us recall how problems arise in a naive call-by-value transformer, such as $\mathcal{M}$ from Section 9.2, when transforming the term $g_1\ (g_2\ x)$, where g_1, g_2 are defined by patterns. Since g_2 has insufficient information, it will not be unfolded by $\mathcal{M}$; we could also imagine that g_2 has sufficient information but were made residual for other reasons. The action taken by $\mathcal{M}$ is to replace the call $g_2\ x$ by a call $g'\ x$ where g' is an optimized version of f. This means however that the call to g_1 cannot be unfolded since we do not know what outermost constructor $g'\ x$ will produce, so the overall result is $g''\ (g'\ x)$ where g'' is an optimized version of g_2.

In contrast with this, $\mathcal{W}$ operates as follows. The result of transforming the term $g_1\ (g_2\ x)$ is $h\ x$ where h is defined by the samme patterns as g_2, and where the right hand side corresponding to the j'th clause will be the result of transforming $g\ t^{g_2,c_j}\{v_1^{g_2,c_j} := x\}$. If t^{g_2,c_j} has an outermost constructor the transformation algorithm can take advantage of it now, even though we introduced a residual function. The point is that the residual function is not introduced in the redex.

For efficiency of $\mathcal{W}$ it is of course desirable that the set of residual functions introduced during transformation be as small as possible, because this set is searched for candidates for folding back. This problem has recently been investigated in partial evaluation of functional programs [Mal93] and partial evaluation of logic programs [Pre93]. Chin's [Chi93a] formulation of the deforestation algorithm introduces fewer residual functions during transformation, by employing simple optimizations; for instance, it is not necessary to introduce residual functions for terms with no calls to recursive functions. Of course, such improvements make the algorithm harder to understand and reason about.

Postunfolding in related transformers

Finally, we mention postunfolding phases in related transformers.

Both Mix and Similix have a postunfolding phase. In Mix the postunfolding phase checks, before unfolding $f\ t$, whether t contains a call and f is non-linear, and if so does not unfold the call. In Similix the problem does not occur because of the let-expressions introduced before transformation.

As was the case during transformation, it must also be ensured that the postunfolding terminates. If there are two calls to the same function in the residual program, then Similix will not unfold them; in particular, recursive functions are not unfolded, and hence unfolding terminates. Mix analyzes the call graph to ensure that no recursive functions are unfolded. The means of ensuring termination in Mix and Similix during ordinary transformation are discussed in Chapter 13.

11.4 The problem of duplicated code

Example **11.4.1** Consider the following variant of the program from Section 11.2.

$$\begin{array}{lcl} & & g\ (Cons\ A\ (Cons\ A\ (Cons\ A\ Nil)))\ y \\ g\ Nil\ y & = & y \\ g\ (Cons\ x\ xs)\ y & = & f\ (g\ xs\ z) \\ f\ w & = & B\ w\ w \end{array}$$

This term and program is transformed into the term

$$B\ (B\ (B\ z\ z)\ (B\ z\ z))\ (B\ (B\ z\ z)\ (B\ z\ z))$$

As is remarked in [Ses88a], the size of this term is exponential in the size of the original term

$$g\ (Cons\ A\ (Cons\ A\ (Cons\ A\ Nil)))\ y$$

To prevent such growth, we can generalize as in Section 11.2, transforming the program into:

$$
\begin{array}{lcl}
 & & g\ (Cons\ A\ (Cons\ A\ (Cons\ A\ Nil)))\ u \\
g\ Nil\ y & = & y \\
g\ (Cons\ x\ xs)\ y & = & \text{let } v = g\ xs\ z \text{ in } f\ v \\
f\ w & = & B\ w\ w
\end{array}
$$

on which $\mathcal{W}$ yields the term:

$$\text{let } x = (\text{let } y = (\text{let } z = u \text{ in } B\ z\ z) \text{ in } B\ y\ y) \text{ in } B\ x\ x$$

the size of which is only linear in the size of the original program. □

Whether this problem is really a problem is debatable. Wadler and Ferguson argue that exponential growth is unlikely in practice [Fer88] p45. Turchin also has means of preventing growth, similar to the generalization technique described above, but the places to generalize are decided by the user [Tur80b] p454. In $\mathcal{W}$ no action is taken to prevent the problem.

Chapter 12

Introduction to the Problem of Ensuring Termination of $\mathcal{W}$

> It may seem strange that the problem of generalization
> is raised in the context of partial evaluation.
> Indeed, partial evaluation is mostly used for [...] specialization,
> and this is something opposite to generalization.
>
> *Valentin F. Turchin.*[1]

There are often two problems of infinity in fold/unfold based transformers: generation of infinitely many functions in the residual program and infinite unfolding of calls in a term being transformed, see *e.g.* [Ses88a]. The first occurs when the transformer reaches infinitely many *different* terms, whereas the second happens when the transformer unfolds infinitely many steps. The main difference of the two is that in the former case it is not a problem that the same term is encountered over and over again, whereas in the latter case that *is* a problem.

As can be seen from the definition of $\mathcal{W}$, the latter problem, infinite unfolding, never occurs during application of $\mathcal{W}$. This is because $\mathcal{W}$ always unfolds exactly one step and records *every* step of transformation in a global log, see Section 11.3.

The first two sections serve as motivation for subsequent developments, and should be considered as informal. Section 12.1 shows three canonical ways $\mathcal{W}$ can fail to terminate and in each case finds a generalization (see Section 3.3) on the program, ensuring that application of $\mathcal{W}$ to the new program terminates. Section 12.2 describes some common $\mathcal{W}$-termination patterns.

The last four sections state some more general results rigorously. Section 12.3 gives a proposition stating precisely the termination problem in terms of transformation trees. Section 12.4 gives a syntactic characterization of infinite sets of terms. The three canonical ways of non-$\mathcal{W}$-termination are instances of the general pattern formulated in this section. Section 12.5 shows that the problem of deciding for arbitrary term and program whether $\mathcal{W}$ terminates is recursively unsolvable. Section 12.6 shows that there are recursive functions for which application of $\mathcal{W}$ to any $M_{1/2}$ formulation fails to terminate.

[1] In [Tur88]. The comment apparently originally comes from a referee report by Olivier Danvy (personal communication with Danvy).

12.1 The canonical non-$\mathcal{W}$-termination patterns

The problem of the Accumulating Parameter

EXAMPLE **12.1.1** (The Accumulating Parameter.)

$$\begin{array}{lcl} & & r\,l \\ r\;xs & \leftarrow & rr\;xs\;Nil \\ rr\;Nil\;ys & \leftarrow & ys \\ rr\;(Cons\;z\;zs)\;ys & \leftarrow & rr\;zs\;(Cons\;z\;ys) \end{array}$$

The r function returns its argument list reversed. $\mathcal{W}$ does not terminate on this program and term. The problem is that $\mathcal{W}$ encounters the terms $rr\;l\;Nil$, $rr\;zs\;(Cons\;z_1\;Nil)$, $rr\;zs\;(Cons\;z_2\;(Cons\;z_1\;Nil))$, *etc.* which become progressively larger.

Since the formal parameter ys of rr is bound to progressively larger terms, Chin [Chi90] calls x an *accumulating parameter.*[2] We might also in the spirit of Chin call rr a *bad consumer* of its ys argument, because rr does not consume the value bound to ys as quickly as it is built up in the calls to rr.

Note that each of the problematic terms that are bound to ys is a subterm of the term which is subsequently bound to ys.

It would seem that we can solve the problem if we could, somehow, make sure that $\mathcal{W}$ could not tell the difference between the terms Nil, $Cons\;z_1\;Nil$, $Cons\;z_2\;(Cons\;z_1\;Nil)$, *etc.* This can be achieved by generalizing the second argument in all calls to rr:

$$\begin{array}{lcl} & & r\,l \\ r\;xs & \leftarrow & \text{let } v = Nil \text{ in } rr\;xs\;v \\ rr\;Nil\;ys & \leftarrow & ys \\ rr\;(Cons\;z\;zs)\;ys & \leftarrow & \text{let } v = Cons\;z\;ys \text{ in } rr\;zs\;v \end{array}$$

Applying $\mathcal{W}$ to this program yields the program unchanged. □

The problem of the Obstructing Function Call

EXAMPLE **12.1.2** (The Obstructing Function Call.)

$$\begin{array}{lcl} & & r\,l \\ r\;Nil & \leftarrow & Nil \\ r\;(Cons\;z\;zs) & \leftarrow & a\;(r\;zs)\;z \\ a\;Nil\;y & \leftarrow & Cons\;y\;Nil \\ a\;(Cons\;x\;xs)\;y & \leftarrow & Cons\;x\;(a\;xs\;y) \end{array}$$

The r function again reverses its argument, this time by first reversing the tail and then appending the head to this (the a function puts the element y in the end of its first argument.)

Now the problem is that $\mathcal{W}$ encounters the terms $r\,l$, $a\;(r\;zs)\;z_1$, $a\;(a\;(r\;zs)\;z_2)\;z_1$, *etc.*

We call each of the calls to r in the redex position an *obstructing function call*, since they prevent the surrounding term from ever being transformed.[3] We might also in the spirit of Chin call r a *bad producer*, because it will never evaluate to a term with an outermost constructor that the surrounding a could consume.

Note that each of the problematic terms that $\mathcal{W}$ encounters appears in the redex position of the subsequent problematic term.

It would seem that we can solve the problem if we could, somehow, make sure that $\mathcal{W}$ could not tell the difference between the terms that occur in the redex position. This can be achieved by generalizing

[2] The same name is usually used for the programming style rr is written in.
[3] We differ slightly from the terminology of Chin here.

the recursive call to r:

$$
\begin{array}{lcl}
 & & r\ l \\
r\ Nil & \leftarrow & Nil \\
r\ (Cons\ z\ zs) & \leftarrow & \text{let } v = r\ zs \text{ in } a\ v\ z \\
a\ Nil\ y & \leftarrow & Cons\ y\ Nil \\
a\ (Cons\ x\ xs)\ y & \leftarrow & Cons\ x\ (a\ xs\ y)
\end{array}
$$

Applying $\mathcal{W}$ to this term and program again yields the program unchanged.[4] □

The problem of the Accumulating Side effect

The two problems in the preceding two sections are quite well-known in the literature on deforestation, as can be seen in the works of Chin [Chi90,Chi92b,Chi93a], Hamilton [Ham91,Ham92b], and the author [Sor93a,Sor93b,Sor94a]. But we saw in part II that positive supercompilation is stronger than deforestation in some respects. It is therefore natural to expect that new problems in ensuring termination pop up. The problem described in this section will be seen to be *the* new problem that pops up.

EXAMPLE **12.1.3** (The Accumulating Side effect.)

$$
\begin{array}{lcl}
 & & f\ v\ v \\
f\ Nil\ ys & \leftarrow & ys \\
f\ (Cons\ x\ xs)\ ys & \leftarrow & f\ xs\ ys
\end{array}
$$

Here f is not intended to be an interesting function, merely to provide a simple illustration of a problem that occurs in more complicated contexts.

$\mathcal{W}$ does not terminate on this program and term. The problem is that $\mathcal{W}$ encounters the progressively larger terms $f\ v\ v$, $f\ xs_1\ (Cons\ x_1\ xs_1\ )$, $f\ xs_2\ (Cons\ x_1\ (Cons\ x_2\ xs_2\ )\ )$, *etc.*

Note that f does not have an accumulating parameter by itself. The instantiation of the first of f's arguments forces the instantiation of the second argument as well, and it is for this reason that f encounters progressively larger terms.

Since the second parameter y of f is bound to progressively larger terms by the instantiation of the first, we call y an *accumulating side effect.*

It would seem that we can solve the problem if we could, somehow, make sure that $\mathcal{W}$ could not tell the difference between the terms v, $Cons\ x_1\ xs_1$, $Cons\ x_1\ (Cons\ x_2\ xs_2)$, *etc.* This can be achieved by generalizing the first argument in the first call to f in the term:

$$
\begin{array}{lcl}
 & & \text{let } u = v \text{ in } f\ u\ v \\
f\ Nil\ ys & \leftarrow & ys \\
f\ (Cons\ x\ xs)\ ys & \leftarrow & f\ xs\ ys
\end{array}
$$

Applying $\mathcal{W}$ to this term and program yields the term and program unchanged. □

Conclusion

Let us sum up the three examples. In the case of the accumulating parameter or side effect, we have a term $e[r]$, where $r \equiv ht_1 \ldots t_n$, and $e[r]$, in a number of steps, is transformed into $e[r']$ where $r' \equiv ht'_1 \ldots t'_n$ and some t'_i is a superterm of t_i for some i.

In this case we should generalize h's i'th argument. That is, everywhere in the program we should transform terms of form $e(ht_1 \ldots t_{i-1} t_i t_{i+1} \ldots t_n)$ into: let $v = t_i$ in $e(ht_1 \ldots t_{i-1}\, v\, t_{i+1} \ldots t_n)$. Given the very cautious way $\mathcal{W}$ handles let-constructs this will prevent $\mathcal{W}$ from encountering $e(ht_1 \ldots t_{i-1} t t_{i+1} \ldots t_n)$ for arbitrarily large t.

In the case of the obstructing function call, we have a term $e[r]$, where $r \equiv ht_1 \ldots t_n$, and r in a number of steps is transformed into $e[e'[r]]$[5] for some $e' \neq []$. In this case we should *generalize calls to* h, that is,

[4] Both the original and the residual program traverse their input twice, but $\mathcal{W}$ cannot change that.

[5] As follows from the definition of contexts, this term is read as follows. Given the contexts e, e' and the redex r. Substitute r for the hole in e' yielding a term t. Then substitute this term for the hole in e.

everywhere in the program we should transform terms of form $e(h\,t_1 \dots t_n)$ into let $v = h\,t_1 \dots t_n$ in $e(v)$. Given the very cautious way $\mathcal{W}$ handles let-constructs this will prevent $\mathcal{W}$ from encountering $e[h\,t_1 \dots t_n]$ with h occurring arbitrarily deeply nested within e.

REMARK **12.1.4** This technique is an *off-line* technique in two senses. First, the analysis to figure out where to generalize (to be discussed later) is conducted before application of $\mathcal{W}$. Second, and not entirely standard, the actions to prevent infinite specialization are also carried out before applying $\mathcal{W}$. In deforestation it is common to use an analysis to figure out where to generalize, and then put annotations on the program. The extended deforestation algorithm then observes these annotations and generalizes during transformation when the annotations say so. Such *on-line* generalizations do essentially the same as our generalizations above, but during transformation.

The use of not only off-line analyses but also off-line generalizations was suggested to the author by Glück. The idea not only simplifies the subsequent development, as can be seen by a comparison of Chapter 14 with [Sor93a], but also allows several new points to be made. □

12.2 $\mathcal{W}$-Termination patterns

Section 12.1 was concerned with non-$\mathcal{W}$-termination patterns. This section is concerned with the opposite, $\mathcal{W}$-termination patterns, in an informal way.

We have seen in the preceding sections that $\mathcal{W}$ can fail to terminate because things grow unboundedly. Therefore a simple pattern to ensure $\mathcal{W}$-termination is: whenever a term $e[r]$ in a number of steps is transformed into $e'[r']$ where r and r' are calls to the same function (a loop) then $e' \leq e$ and $r' \leq r$ by some well-founded ordering (*e.g.* term size ordering.) We call this the *non-increasing* criterion.

However it may happen that from one call of r to the next, some arguments increase in size while others decrease. This gives rise to the following less conservative criterion, which we call the *decreasing* criterion: for every function h there exist indexes $i_1 \dots i_k$ such that whenever a term $e[r]$ in a number of steps is transformed into $e'[r']$ where r and r' are calls to h then $e' \leq e$, and either (1) $r' \leq r$ (no argument has increased); or (2) $r' > r$, but one of the arguments $t_{i_1} \dots t_{i_k}$ have decreased and none of the other of *these* arguments have increased. The idea can be extended to take growing contexts into account.

Finally, it may happen that from one call of a function to the next call of the same function, some arguments have grown whereas the others are unchanged, and yet transformation still terminates. This gives rise to the last of our $\mathcal{W}$-transformation patterns, which we call the *bounded* criterion. Let (p, t_0) be the input to $\mathcal{W}$, let k be the size of the largest argument in t_0 that does not contain a function call, and let m be the largest number of function calls in t_0 or a right hand side in p. The criterion then is: for all terms t encountered by $\mathcal{W}$, t contains at most m function calls and the largest argument that does not contain a function call is less than k. Less conservative bounds on the number of function calls can be devised.

Variations of these three criteria are common in the literature. Chin [Chi90,Chi92b,Chi93a] and the author [Sor93a,Sor93b,Sor94a] use a non-increasing criterion for deforestation. Holst [Hol91], Jones *et al.* [Jon93] use a decreasing criterion for partial evaluation. Jones [Jon88a,Jon88b] uses a combined decreasing and bounded criterion for partial evaluation. We return to these techniques in Chapter 13.

REMARK **12.2.1** (Call-by-value/call-by-name, on-line/off-line.) Static analyses to test the decreasing criterion (which subsumes the non-increasing criterion) are hard to formulate for lazy languages because it is hard to predict which calls will be evaluated, but have been successfully formulated for eager languages.

Static analyses for the non-increasing criterion have been successfully formulated for lazy languages.

As will be seen in Section 14.2, static analyses for the bounded criterion for realistic examples are very complicated to formulate for both lazy and eager languages.

On-line analyses, *i.e.* tests of the condition during transformation, are hard to formulate for the decreasing criterion for a number of reasons, pointed out in [Jon88a]. On-line analyses for at least simple variations of the non-increasing criterion are conceptually simple, but still seem impractical. On-line tests of the bounded criterion are simple and efficient, as will be discussed in Section 14.3. □

12.3 Quasi-finiteness

In this section we show how the problem of ensuring termination of $\mathcal{W}$ can be reduced to a certain simpler condition on transformation trees.

DEFINITION **12.3.1** A tree or graph is quasi-finite MVR iff only finitely many terms in the tree or graph are not identical MVR. □

PROPOSITION **12.3.2** *$\mathcal{W}[\![t_0]\!]$ terminates iff the transformation tree $\mathcal{T}_{\mathcal{W}}[\![t_0]\!]$ is quasi-finite MVR.* □

PROOF: For brevity, let $T = \mathcal{T}_{\mathcal{W}}[\![t_0]\!], G = \mathcal{G}_{\mathcal{W}}[\![t_0]\!]$. We are to show that G is finite iff T is quasi-finite MVR, see Section 4.2. First observe that (*) the *set* of terms occurring in T and G are the same.

Now the left to right direction follows: if G is finite then G is quasi-finite MVR, and by (*) so is T.

As for the right to left direction, we show the contrapositive: if G is infinite, then T is not quasi-finite MVR. So assume that G is infinite. It suffices to show that G is not quasi-finite MVR, because then by (*) neither is T.

Since G is infinite there exists by König's Tree Lemma an infinite sequence of nodes in G containing $t_1, t_2 \ldots$. Consider for a term t the lexicographically ordered measure (m, n) where m is the number of conditionals in t and n is $|t|$, see Section 2.2. Clearly each application of all clauses of $\mathcal{N}_{\mathcal{T}}$ except (2),(3a),(3b) strictly decreases this measure, and so there must be an infinite number of terms $t_{i_1}, t_{i_2}, \ldots$ which are function calls. But then no two of these terms can be identical MVR, for in such a case the branch would have been terminated. So the branch is not quasi-finite MVR, and we have completed the proof. □

So to check that (i) $\mathcal{W}[\![t_0]\!]$ terminates with a program p we need only check that (ii) the tree $\mathcal{T}_{\mathcal{W}}[\![t_0]\!]$ with p is quasi-finite MVR. And to transform t, p into t', p' so that (i) holds, we need only transform t, p into t', p' so that (ii) holds.

12.4 A general characterization of Non-$\mathcal{W}$-termination

Section 12.1 showed the three canonical ways in which programs and terms can be non-$\mathcal{W}$-terminating. This section describes more generally and precisely the appearance of the set of terms that $\mathcal{W}$ encounters when it loops infinitely.

The most simple, *but incorrect*, characterization of infinite sequences of terms encountered by $\mathcal{W}$ in the case of non-termination would be as follows. Let $\sqsubseteq$ denote subterm ordering, *i.e.* $s_1 \sqsubseteq s_2$ iff s_1 is a subterm of s_2 (or if s_1 is a subterm of a term that is identical MVR to s_2). If the transformation tree $\mathcal{T}_{\mathcal{W}}[\![t_0]\!]$ contains an infinite branch $s_1, s_2 \ldots$ then there is an infinite subsequence $s_{i_1}, s_{i_2} \ldots$ such that $s_1 \sqsubseteq s_2 \sqsubseteq \ldots$. We saw already in the case of the accumulating parameter that this characterization is incorrect; in that example $\mathcal{W}$ encountered the terms $rr\ l\ Nil$, $rr\ zs\ (Cons\ z_1\ Nil)$, $rr\ zs\ (Cons\ z_2\ (Cons\ z_1\ Nil))$, *etc.* where no term is a subterm of a subsequent term. Of course, if we replace the subterm ordering with the term size ordering, *i.e.* $s_1 \sqsubseteq s_2$ iff $|s_1| \leq |s_2|$, then the characterization is correct, but rather coarse-grained.

Below we develop a fine-grained characterization. The following definition, theorem, and corollary are adopted from [Der87], although we provide the original references.

DEFINITION **12.4.1** Let h range over f-function names, g-function names and constructor names, and v over variable names. Define the *homeomorphic embedding relation* $\trianglelefteq$ on M_1 terms as follows.

$$v \trianglelefteq v$$

$$\frac{s \trianglelefteq t}{s \trianglelefteq h\, t_1 \ldots t_{i-1}\, t\, t_{i+1} \ldots t_n}$$

$$\frac{s_1 \trianglelefteq t_1 \ldots s_n \trianglelefteq t_n}{h\, s_1 \ldots s_n \trianglelefteq h\, t_1 \ldots t_n}$$

□

The following remark throws some light on $\trianglelefteq$.

REMARK **12.4.2** (*i*) the third rule in particular states that $h \trianglelefteq h$ for 0-ary constructors and functions.

(*ii*) let a one-step *deletion* be the operation of deleting a function symbol along with all its arguments but one from a term. For instance, all the one-step deletions of $f\,(g\,s_0\,s_1\,s_2)\,t$ are t, $g\,s_0\,s_1\,s_2$, $f\,s_0\,t$, $f\,s_1\,t$, $f\,s_2\,t$. It then holds that $s \trianglelefteq t$ iff s can be obtained by repeatedly applying one-step deletions (zero or more times) to t.

(*iii*) suppose that $s \trianglelefteq t$. Then $|s| \leq |t|$, and $|s| = |t|$ iff $s \equiv t$. □

The following theorem is due to Higman [Hig52] (Higman's Lemma) and in a more general form to Kruskal [Kru60] (Kruskal's Tree Theorem.) Both results have beautiful proofs due to Nash-Williams [Nas63].

THEOREM **12.4.3** *Let $t_1, t_2, \ldots$ be an infinite sequence of M_1 terms over a finite set of function names, constructor names, and variable names. Then there exists $i < j$ such that $t_i \trianglelefteq t_j$.* □

Applying the infinite version of Ramsey's Theorem [Ram30] one can prove:

COROLLARY **12.4.4** *Let $t_1, t_2, \ldots$ be an infinite sequence of M_1 terms over a finite set of function names, constructor names, and variable names. Then there exists an infinite subsequence $t_{i_1}, t_{i_2} \ldots$ so that $t_{i_1} \trianglelefteq t_{i_2} \trianglelefteq \ldots$.* □

We now apply these classics to the present problem of characterizing non-$\mathcal{W}$-termination.

DEFINITION **12.4.5** If $t_1 \trianglelefteq t_2$ but not $t_1 \equiv t_2$, we write $t_1 \lhd t_2$. □

REMARK **12.4.6** By Remark 12.4.2, $t_1 \lhd t_2$ implies $|t_1| < |t_2|$. Of course, the inverse implication does not generally hold. □

COROLLARY **12.4.7** *(p, t_0) is non-$\mathcal{W}$-terminating iff the transformation tree $\mathcal{T}_{\mathcal{W}}[\![\, t_0 \,]\!]$ contains an infinite branch $t_1, t_2, \ldots$ with an infinite subsequence $t_{i_1}, t_{i_2}, \ldots$ such that $t_{i_1} \lhd t_{i_2} \lhd \ldots$.* □

PROOF: First the right to left direction. By Remark 12.4.6, $\mathcal{T}_{\mathcal{W}}$ cannot be quasi-finite MVR. The result now follows from Proposition 12.3.2.

Now to the left to right direction. Since (p, t_0) is non-$\mathcal{W}$-terminating, $\mathcal{T}_{\mathcal{W}}[\![\, t_0 \,]\!]$ is not quasi-finite MVR by Proposition 12.3.2. This means that $\mathcal{T}_{\mathcal{W}}[\![\, t_0 \,]\!]$ contains an infinite branch $t_1, t_2, \ldots$ where no terms are identical MVR. By Corollary 12.4.4 the branch contains a subsequence $t_{i_1}, t_{i_2}, \ldots$ such that $t_{i_1} \trianglelefteq t_{i_2} \trianglelefteq \ldots$. Since no terms in the branch are identical (not even MVR) it must hold that in fact $t_{i_1} \lhd t_{i_2} \lhd \ldots$. □

This gives a precise and concrete characterization of transformation trees corresponding to non-$\mathcal{W}$-terminating (p, t_0). The reader may like to check that the three canonical patterns of non-$\mathcal{W}$-termination are all instances of this general result.

12.5 Recursive Unsolvability of $\mathcal{W}$-termination

This section shows that the problem of deciding whether $\mathcal{W}[\![\, t \,]\!]$ in the context of p terminates is recursively unsolvable. The idea is that for ground t, a procedure deciding wheter $\mathcal{W}[\![\, t \,]\!]$ terminates would yield a procedure deciding $\mathcal{I}[\![\, t \,]\!]$ terminates, but the latter problem is recursively unsolvable.

DEFINITION **12.5.1** The *$\mathcal{W}$-halting problem* is the problem with input parameters p, t of deciding whether (p, t) is $\mathcal{W}$-terminating. The *$\mathcal{I}$-halting problem* is the problem with input parameters p, t, θ, where θ is a constant substitution, of deciding whether $(p, t\theta)$ is $\mathcal{I}$-terminating. □

PROPOSITION **12.5.2** *The $\mathcal{I}$-termination problem is recursively unsolvable.* □

PROOF: It is easy to see that using the encoding $\lceil n \rceil \equiv Succ^n\, Zero$, M_1 can express every partial recursive function f of k variables by a program p and a call $f\,x_1 \ldots x_k$, where application of f to numbers $n_1 \ldots n_k$ is represented by the constant substitution $\{x_i := \lceil n_i \rceil\}_{i=1}^k$.

The problem with input parameter a partial recursive function f of deciding whether f is defined with all arguments 0 is known to be recursively unsolvable, and therefore so is the $\mathcal{I}$-termination problem. □

The following shows that $\mathcal{W}$ includes the power of an interpreter.

PROPOSITION **12.5.3** *(i) for a ground term t,* $\mathcal{T_I}[\![\, t \,]\!] = \mathcal{T_W}[\![\, t \,]\!]$. *(ii) for any t,* $\mathcal{T_W}[\![\, t \,]\!]$ *is infinite iff* $\mathcal{G_W}[\![\, t \,]\!]$ *is either infinite or contains a cycle.* □

PROOF: (i) follows by induction on t. (ii) follows from the folding scheme of $\mathcal{G_W}$. □

THEOREM **12.5.4** *The* $\mathcal{W}$*-halting problem is recursively unsolvable.* □

PROOF: Assume otherwise that the $\mathcal{W}$-halting problem were decidable, *i.e.* that we had a procedure telling us whether $\mathcal{G_W}[\![\, t \,]\!]$ is finite.

Applying the preceding proposition and decidability of cycle detection in finite graphs, we then obtain a procedure for testing $\mathcal{I}$-termination of (p, t, θ), where $t_0 \equiv t\theta$ is ground, by splitting into the following cases. (i) $\mathcal{G_W}[\![\, t_0 \,]\!]$ is infinite, then so is $\mathcal{T_I}[\![\, t_0 \,]\!]$. (ii) $\mathcal{G_W}[\![\, t_0 \,]\!]$ is finite and contains a cycle, then $\mathcal{T_I}[\![\, t_0 \,]\!]$ is infinite. (iii) $\mathcal{G_W}[\![\, t_0 \,]\!]$ is finite and contains no cycle, then $\mathcal{T_I}[\![\, t_0 \,]\!]$ is finite. □

12.6 On the need for generalizations in principle

In this section we show that there are certain recursive functions for which $\mathcal{W}$ will loop infinitely on any $M_{1/2}$ formulation. This shows that there are recursive functions which inherently needs generalization in order for $\mathcal{W}$ to terminate.

We shall use an argument resembling the diagonal argument usually used to prove that there are recursive functions which are not primitive recursive. The following proposition, which is a preparation for this argument, shows that given $M_{1/2}$ term and program t, p one can formulate a recursive function of one variable m giving an upper bound on the result of any run of t when m is the size of the largest input to t.

PROPOSITION **12.6.1** *Let* p, t_0 *be* $M_{1/2}$*-treeless. Then there exists a constant K satisfying the following property. Let* θ *be a constant substitution, v be the variable among* $FV(t_0)$ *with largest* $|v\theta|$, $m = |v\theta|$. *Then either* $\mathcal{I}[\![\, t_0\theta \,]\!]$ *does not terminate, or* $\mathcal{I}[\![\, t_0\theta \,]\!]$ *terminates with a constant b with* $|b| \leq K^m(Km)$. □

PROOF: Given $M_{1/2}$ treeless p, t_0 and a constant substitution θ. Among the variables in t_0, let v be the one for which $|v\theta|$ is largest, and let $m = |v\theta|$. Let k be the maximum of 1 and the largest number of arguments to a constructor in t_0 or a right hand side of p. Among t_0 and the right hand sides of p, let t' be the term for which $|t'|$ is largest, and let $n = |t'|$. Let for brevity $I = \mathcal{T_I}[\![\, t_0\theta \,]\!]$.

Assume that I is finite, *i.e.* that $\mathcal{I}[\![\, t_0\theta \,]\!]$ terminates. We find the constant K by proving a number of steps.

(i) for any term t in a node N in I it holds that every argument b in a function call in t has $|b| \leq m$. *proof:* induction on the distance of N from the root; in the induction step apply the fact that in any right hands side of p, all calls have variable arguments.

(ii) for any term t in a node in I, $|t| \leq mn$. *Proof:* Induction as above; in the induction step apply (i).

(iii) for any term t in I and every subterm $c\, t_1 \ldots t_n$ of t, if one of the t_i's are active, then the constructor c occurs in t_0 or p. *Proof:* Induction as above. In the base case the assertion follows from the fact that θ is a constant substitution. In the induction step use the fact that only constants are substituted for formal parameters.

(iv) there at most k^{mn} nodes in I. *Proof:* Since I is finite, and there are at most mn different terms in I by (ii), the longest path from the root to a leaf visits at most mn nodes. Since every node has at most k children by (iii), there can be at most $k^{mn} - 1$ nodes in I.

(v) the constant b returned by $\mathcal{I}[\![\, t_0\theta \,]\!]$ has $|b| \leq k^{mn}mn$. *Proof:* recall from Section 4.1 how the value returned by $\mathcal{I}$ can be recovered from I. In the result every non-leaf node adds at most one constructor to the result, whereas leaf nodes add the whole term that they contain to the result. Since all terms have positive size, the overall result b is at most as big as the total number of nodes multiplied by the size of the largest term occurring in I. By $(ii), (iv)$, this number is no larger than $k^{mn}mn$.

(vi) the constant K can be chosen as k^n. *Proof:* by (v). Note that K is independent of θ. □

Proposition **12.6.2** *The recursive function $f(m) = (m+1)^{m+3}$ cannot be computed by any $M_{1/2}$-treeless term and program using the encoding $\lceil m \rceil \equiv Succ^m\ Zero$.* □

Proof: Let p be an $M_{1/2}$-treeless program and let t be an $M_{1/2}$-treeless term with one free variable x. Note that for any number n, $|\lceil n \rceil| = n + 1$.

Let the result returned by $\mathcal{I}[\![\, t\{x := \lceil K \rceil\} \,]\!]$ be b. We are to show that b cannot be $\lceil f(K) \rceil$. We have

$$|b| \leq K^{|\lceil K \rceil|}(K\,|\lceil K \rceil|) = K^{K+1}K(K+1) \leq (K+1)^{K+3}$$

But then it cannot be the case that $b \equiv \lceil f(K) \rceil$, for

$$|\lceil f(K) \rceil| = \left|\lceil (K+1)^{K+3} \rceil\right| = (K+1)^{K+3} + 1$$

That is: the result returned by $\mathcal{I}$ is too small to be right. □

There are certain (abstract) functions which can be given different concrete $M_{1/2}$ formulations, where $\mathcal{W}$ will terminate for one but not for the other. A trivial example is the identity on lists, $id\ xs \leftarrow xs$, which does not terminate in the formulation $id'\ xs \leftarrow r\ (r\ xs)$, where r is one of our reverse functions, because of the problems with reverse, see Section 12.1.

One might wonder whether all non-termination problems are caused by such "unfortunate" choice of formulations, *i.e.* whether for any function there is an $M_{1/2}$ formulation on which $\mathcal{W}$ terminates. The preceding proposition shows that the answer to this question is *no*. Recall from Chapter 10 that the output of $\mathcal{W}$ is equivalent to the input. Specifically, if a program p and a term t with n free variables represents the partial recursive function f of n variables, then so does the term and program returned by $\mathcal{W}$. If all partial recursive functions had a formulation as an $M_{1/2}$ term and program t, p for which $\mathcal{W}$ terminated, then it would also have a formulation as an $M_{1/2}$-treeless term and program obtained by applying $\mathcal{W}$ to t, p, but this contradicts the preceding proposition.

The underlying reason for this phenomenon is that the class of $M_{1/2}$-treeless terms and programs is too small to represent all recursive functions, or, equivalently, that $\mathcal{W}$ is to powerful for all recursive functions to have an $M_{1/2}$ formulation on which $\mathcal{W}$ stops.

Chapter 13

Termination of Related Transformers

Algoritmer der ikke standser
siger mig ikke noget.

Nils Andersen.

This chapter reviews existing techniques for automatically guaranteeing termination of transformers. The chapter is intended as a tool box that can be applied in the next chapter on automatic techniques for ensuring termination of $\mathcal{W}$.

Section 13.1 describes methods that ensure termination of the deforestation algorithm. Section 13.2 describes methods that ensure termination partial evaluation of strict functional programs. Section 13.3 describes Turchin's previous means of ensuring termination of the supercompiler. Section 13.4 describes techniques for proving termination of logic programs. Section 13.5 describes methods that ensure termination of transformers for logic programs.

One field is conspicuous by its absence: rewrite systems. Indeed, there is a significant body of literature on termination of rewrite systems; a survey is given by Dershowitz [Der87]. However, these techniques are most often intended for *manually* proving the termination of a given rewrite system. The author is aware of a three papers on automatic means of proving termination for rewrite systems, but has yet not investigated them in any detail.

The exposition is to a large extent based on examples.

13.1 Deforestation

There are basically two means of ensuring termination of deforestation.

The first consists in requiring that the object programs and terms be written in specific formats for which termination can be guaranteed. This is the approach in Wadler's own work [Wad88,Fer88] where deforestation is restricted to what we have called linear M_0-treeless programs in Section 5.1. It is also essentially the approach in the recent work [Gil93], requiring structures to be explicitly built and destructed, and in [She93] which is concerned with programs in the formats of the "Bananas, Lenses, ..." paper [Mei91].

The second consists in annotating the object programs and modifying the deforestation algorithm so as to take the annotations into account. This approach, inspired by the so-called *blazed* deforestation algorithm in [Wad88], was started by Chin [Chi90,Chi92b,Chi93a], and later taken up by Hamilton and Jones [Ham91] and the author [Sor93a,Sor93b]. A more recent work by the author recasts the latter technique in terms of pre-deforestation generalizations [Sor94a].

Below we review Wadler's technique and show how it was extended by Chin and the author. Although all results in the original papers are concerned with M_0 we state the results for $M_{1/2}$. The author has verified that the proofs are simple modifications of the original proofs. Also, in Chin's technique one annotates the program and applies to it the *extended* deforestation algorithm which takes annotations into account. We explain the technique in terms of pre-transformation generalizations as outlined in Section 12.1. The two approaches are essentially equivalent.

Wadler's technique

The major result in [Fer88] is:(See Section 5.1 for definitions of treeless terms)[1]

THEOREM **13.1.1** *(Deforestation Theorem.) For a term t and an $M_{1/2}$-treeless program p, $\mathcal{S}[\![t]\!]$ terminates.* □

Since an $M_{1/2}$-treeless program does not build any intermediate data structures, all the constructing and destructing is caused by nested calls in t.

To ensure non-degradation of efficiency, programs are required to be linear. In fact, linearity is taken as part of the definition of treelessness in [Fer88].

It does not generally hold that $\mathcal{W}$ terminates if applied to a term t and an $M_{1/2}$-treeless program, as the last example in Section 12.1 shows. However, applied to a linear term and linear M_0-treeless program, $\mathcal{W}$ terminates. This follows already from Proposition 9.1.7.

Chin's extension

Wadler's technique was extended by Chin so as to apply to all first-order programs [Chi90,Chi92b,Chi93a]. In this subsection we describe a much simplified version of Chin's technique capturing, in the author's opinion, the essence of the full technique.

The basic observation is that $M_{1/2}$-treeless programs impose *one* restriction on programs: that no nested calls be present. Given an arbitrary $M_{1/2}$ term and program, Chin's technique annotates non-treeless subterms (non-variable function call arguments). An $M_{1/2}$ term annotated this way is essentially the same as an $M_{1/2}$-treeless term.

However, in the present setting we do not consider annotations. Instead, given an arbitrary $M_{1/2}$ term, repeatedly generalize non-variable arguments, considering also terms t_1, t_2 in $t_1 = t_2 \rightarrow t' \,\square\, t''$ as arguments; see Section 3.3. We call this repeated operation *generalization of non-variable arguments.* This operation yields an M_1-treeless term, essentially the same as an annotated $M_{1/2}$ term, which, in turn, is essentially the same as an $M_{1/2}$-treeless term.

In terms of our notion of treelessness, Chin's *Extended Deforestation Theorem* [Chi90] states:

THEOREM **13.1.2** *(Extended deforestation theorem.) Let t, p be M_1-treeless term and program. Then $\mathcal{S}[\![t]\!]$ terminates in the context of p.* □

So, to deforest an arbitrary term and program, just generalize non-variable arguments in the term and in the right hand sides of the program arriving at an M_1-treeless term and program, then termination is guaranteed by Theorem 13.1.2. As described here, Chin's method finds that no generalizations are required iff the term and program are $M_{1/2}$-treeless. As mentioned, Chin has numerous extensions to the basic idea. Below we introduce one.

DEFINITION **13.1.3** Define a term to be M_1-*constant treeless* if it adheres to the following grammar:

$$\begin{array}{lcl} t & ::= & v \mid c\, t_1 \ldots t_n \mid f\, a_1 \ldots a_n \mid g\, a_0\, a_1 \ldots a_n \mid b_1 = b_2 \rightarrow t_1 \,\square\, t_2 \mid \text{let } v = t_1 \text{ in } t_2 \\ b & ::= & v \mid c\, b_1 \ldots b_n \\ a & ::= & v \mid d \\ d & ::= & c\, d_1 \ldots d_n \end{array}$$

A program is M_1-constant treeless if all right hand sides in the program are M_1-constant treeless. □

[1] It is actually required that t be a so-called *function term*, but the present extended version is folklore in the community.

The extension compared to M_1-treeless terms is that now functions can have constant arguments, that is: terms containing only construtors, not variables or anything else.

THEOREM **13.1.4** *(Extended deforestation theorem with constants.) Let p be an M_1-constant treeless program, and t an arbitrary term. Then $\mathcal{S}[\![\, t \,]\!]$ terminates.* □

So to deforest an arbitrary term and program we should only generalize those non-variable arguments which are non-constant.

In terms of the terminology in Section 12.2, Chin's technique resides within the *non-increasing* framework.

A semantics-based technique

Chin's method is *syntactic* in nature; it does not attempt to take the flow of a program into account. Although the technique seems to work well on some examples, one can conceive situations where minor details of the program outsmart it.

Inspired by a dataflow analysis for lazy (higher-order) programs [Jon87b], the author has devised a more fine-grained analysis [Sor93a,Sor93b]. Like Chin's method, the technique calculates annotations and applies an extended deforestation algorithm, but a recent work [Sor94a] uses the idea of pre-transformation generalizations, which we stick to here. The technique is too complicated to fit in a subsection, so we shall settle for an example.

Given a term t and a program p the idea is to compute a *tree grammar*, *i.e.* a grammar where the right hand sides of productions are terms containing non-terminals. The grammar approximates the set of terms that $\mathcal{W}$ encounters when applied to t and p. One then looks whether infinitely many different terms are derivable from the grammar, and if so, generalizes accordingly.

EXAMPLE **13.1.5** Recall the program in Section 12.1 which suffered from the problem of the accumulating parameter.

$$\begin{array}{lll} & & r\;l \\ r\;xs & \leftarrow & rr\;xs\;Nil \\ rr\;Nil\;ys & \leftarrow & ys \\ rr\;(Cons\;z\;zs)\;ys & \leftarrow & rr\;zs\;(Cons\;z\;ys) \end{array}$$

For this program the following grammar will be computed.

$$\begin{array}{lllllll} N^0 & \to & r\;\bullet \mid N^r \mid \bullet \mid N^{ys} & \quad & N^r & \to & rr\;N^{xs}\;Nil \mid N^{rr,Nil} \mid N^{rr,Cons} \\ N^{xs} & \to & \bullet & & N^{rr,Nil} & \to & N^{ys} \\ N^{ys} & \to & Nil \mid Cons\;\bullet\;N^{ys} & & N^{rr,Cons} & \to & rr\;\bullet\;(Cons\;\bullet\;N^{ys}) \mid N^{rr,Nil} \mid N^{rr,Cons} \end{array}$$

In the grammar there is a nonterminal N^f for each f-function in the program, a nonterminal $N^{g,c}$ for each clause of the definition of every g-function in the program, a nonterminal N^v for every variable in the program and finally a start nonterminal N^0. (Nonterminals with no productions are not shown.)

Reading $\bullet$ as "any variable," the idea is that if $\mathcal{W}[\![\, t_0 \,]\!]$ encounters t then t is derivable from N^0 in the grammar, and if v is bound to some term t during transformation then t is derivable from N^v. There is a related idea for nonterminals corresponding to functions.

Recall that the problem of the Accumulating Parameter in Section 12.1 was that rr was called with the progressively larger arguments Nil, $Cons\;z_1\;Nil$, $Cons\;z_2\;(Cons\;z_1\;Nil)$, *etc.* The formal parameter of rr is ys, and in fact these terms are derivable from N^{ys} in this grammar. Also recall that we noted that each problematic term was a subterm of the subsequent problematic term. This is reflected by the production $N^{ys} \to Cons\;\bullet\;N^{ys}$.

The problem of the Accumulating Parameter is generally reflected by presence of cyclic derivations $N^v \to \ldots \to e(N^v)$. In preventing $\mathcal{S}$ from looping, the idea is to generalize every variable v for which $N^v \to \ldots \to e(N^v)$ where $e \neq ()$. For the above grammar this yields the generalization that we found manually in Section 12.1. □

The whole technique is very much inspired by the phenomena of the accumulating parameter and the obstructing function call. For treeless programs the technique finds that no generalizations are required.

In terms of the terminology in Section 12.2, this technique also remains within the non-increasing framework.

13.2 Partial evaluation of functional programs

In the first implementation of a self-applicable partial evaluator, *Mix*, the user had to supply annotations, *binding-time annotations*, [Jon85]. The process was subsequently automated by employing a static analysis to compute annotations [Jon87a,Ses88a] ensuring termination of call unfolding, provided the program did not already contain a "potential infinite loop" (a loop depending only on static information.) Moreover, infinite specialization could still occur. A technique to ensure termination of specialization, provided that the object program did not already contain a potential loop was described in [Jon88a].

Later the techniques from Mix have been improved by Holst [Hol91]. His ideas were simplified and recast in an imperative setting in [Jon93, chapter 14]

Similix uses a quite different automatic strategy [Bon90b] which chooses *dynamic conitionals* as specialization points; all calls are unfolded. The strategy does not ensure finite specialization, and only ensures termination of call-unfolding provided that the object program does not contain a potential loop.

Below we sketch the techniques employed in the Mix project [Ses88a,Jon88a] and the extensions by Holst [Hol91].

Structural induction condition in Mix

In [Ses88a] the *structural induction condition* was used. The paper only considers direct recursive calls

$$f\ s_1 \ldots s_n\ d_1 \ldots d_m \leftarrow e(f\ a_1 \ldots a_n\ b_1 \ldots b_m)$$

where $s_1 \ldots s_n$ are the static parameters and the remaining are dynamic.

A static parameter s_i is *inductive* if the expression a_i computes a value which is a proper substructure of a_i. Examples include $\mathbf{car}(s_i), \mathbf{cdr}(s_i)$. A call satisfies the structural induction condition if there is at least one inductive parameter, and the remaining static parameters are unchanged or inductive in the recursive call.

Calls with only static variables and calls that satisfy the structural induction condition ar unfolded, all other is made residual. The technique relies crucially on the language being strict, so that evaluation of a_i is guaranteed.

The following example from [Ses88a] shows that the technique does not guarantee termination of the method. The technique was found to give good experimental results.

EXAMPLE **13.2.1** Consider the function

$$\begin{array}{lcl} g\ x\ [] & \leftarrow & x \\ g\ x\ (z : zs) & \leftarrow & g\ (A : x)\ zs \end{array}$$

with $x = []$ and z unknown. Clearly g's first argument is static while the second is dynamic. Since the first argument is not inductive, the recursive call to g are made residual. But this means that versions of g specialized to $x = []$, $x = [A]$, $x = [A, A]$, *etc.* are generated. The problem is that the technique should make the first argument to g dynamic rather than just making the call residual. □

Sestoft's technique is within the *decreasing* framework.

Re-examination of basic principles

The technique in the paper by Jones proceeds as follows. Given an initial binding-time analysis for the function f to be specialized stating which arguments to f will be known and unknown. Assign ? to all other variables than those of f (which are classified S/D.) Now reclassify arguments classified as ? as follows.

- *Dynamic dependencies.* All arguments that depend on dynamic arguments are reclassified as dynamic. This is the usual *congruence condition.*
- *Static domination.* All arguments definitely never larger that some program constant or static input are reclassified as static.
- *Dynamic construction.* Arguments built from static arguments under dynamic control are reclassified as dynamic, and arguments built only from constants and static arguments under static control are reclassified as static.

Variations of the method were found to give good results [Jon88b]. This method is a combined decreasing and bounded technique.

Holst's technique

Like in the analyses for deforestation, Holst also identifies accumulating parameters, in his terminology *in situ increasing* parameters. Holst also identifies *in situ decreasing* parameters. Such a parameter has the property that if a call of the function in a number of steps leads to a new call to the same function, then the argument will have decreased. If a function has an increasing parameter, but also has a decreasing parameter, and all calls to the function have a constructor term as argument for the decreasing parameter, then there is no need for annotations. For instance, for the program

$$\begin{array}{lcl} & & f\ (Succ\ (Succ\ Zero))\ Zero \\ f\ (Succ\ c)\ y & \leftarrow & f\ x\ (Succ\ y) \\ f\ Zero\ y & \leftarrow & y \end{array}$$

transformation will terminate although the techniques for deforestation will suggest that the second argument of f is accumulating. One might say that the second argument *is* accumulating, but only a finite number of times, bounded by the first argument.

To decide which parameters are increasing and decreasing, Holst uses an abstract interpretation which relies crucially on the programming language being strict. To handle nested calls the technique also relates the sizes of a function's argument to the size of the result returned by the function (see also the subsection about off-line techniques in Section 13.4). This technique is a decreasing technique.

13.3 Supercompilation

Turchin has described an on-line technique to ensure termination of his supercompiler [Tur88]. His supercompiler basically keeps track of all the terms that has been encountered. If a term is encountered which is a passive instance (see Definition 2.4.4) of a previously encountered term, a fold step is performed. If a term t is encountered such that there is a previously encountered term t' such that both t and t' are passive instances of some term, then the most specific generalization t'' is computed and the transformation is rolled back to t' which is replaced by t'' and the transformation proceeds from there. The handling of nested calls is more complicated.

The analyses in deforestation and partial evaluation attempt to predict the later situation *before* the transformation, and annotate the program at points where generalization appears necessary. Being on-line Turchin's technique has, in some senses, more precise information, but it does not seem to pay attention to decreasing parameters, and so his technique is an increasing technique.

Turchin has also used the idea of using grammars to approximate the terms that the supercompiler will encounter, see [Tur80a] (Section 5.4.) He uses the grammar approximation to get better transformation in the case of nested function calls, but apparently not to ensure termination.

13.4 Termination of logic programs

A recent, extensive survey on termination of logic programs is provided by De Schreye and Decorte [DeS93]. The methods can, as usual, be divided into two camps: on-line and off-line techniques.

On-line techniques

On-line techniques seek to cut off branches in the SLD-tree which cannot possibly be successful. Ideally the method should prune all infnite branches and thus reduce an infinite search space to a finite one with only success or finite failure branches, but the problem is, of course, undecidable in general.

Early on-line techniques were concerned with simple checks such as whether the clause in a node is an instance of a clause from a later node [Cov85a,Cov85b,Poo85]. Later techniques for ensuring termination of partial deduction use more complicated measures; see the next section. As a simple example of an on-line technique consider the following from [Cov85a].

EXAMPLE **13.4.1**

$$b(a,b).$$
$$b(b,c).$$
$$b(c,d).$$
$$b(X,Y) :- b(X,Z), b(Z,Y).$$

In Prolog, the goal $b(X,Y)$ yields the answers $b(a,b), b(b,c), b(c,d)$ directly, the answer (a,c) by one application of the fourth rule, and (a,d) by two applications of the fourth rule, and then enters an infinite branch of the SLD(NF) tree with successive goals (composite goals enclosed in square brackets) $b(X,Y)$, $[b(X,Z), b(Z,Y)]$, $b(b,Y)$, $[b(b,Z), b(Z,Y)]$, $b(c,Y)$, $[b(c,Z), b(Z,Y)]$, $b(d,Y)$, $[b(d,Z), b(Z,Y)]$, $[b(d,Y1), b(Y1,Z), b(Z,Y)]$, $[b(d,Y2), b(Y2,Y1), b(Y1,Z), b(Z,Y)]$, *etc.* This process never stops because the atomic goal $b(d,Yi)$ gives rise to a goal $b(d,Yj), b(Yj,Yi)$ in which the former goal appears as a subgoal. Pruning all branches in which a subgoal directly gives rise to a new goal containing a renaming of the former goal, yields a finite SLD(NF) tree. □

The criterion for pruning can be modified in various ways, and has been so, in particular because the simple technique above can change the success set of Prolog programs. A related criterion is to check whether a goal $g(a_1 \ldots a_n)$ gives rise to a new goal containing $g(b_1 \ldots b_n)$ where the b_i's are instances of the corresponding a_i's. This resembles Turchin's technique.

Off-line techniques

Off-line techniques try to guarantee termination of a goal (in a program) before the goal is triggered [Ull88,Plu90]. These techniques are rather complicated compared to the on-line techniques. We shall consider an example from [Plu90].

EXAMPLE **13.4.2**

$$append([], YS, YS).$$
$$append([X|XS], YS, [X|ZS]) :- append(XS, YS, ZS).$$

$$perm([], []).$$
$$perm(L, [H|T]) :- append(V, [H|U], L), append(V, U, W), perm(W, T).$$

The predicate *perm* is true if the second argument is a permutation of the first. We shall assume that *perm* is invoked with first argument known. The first clause of *perm* is obvious. The second states that any permutation of L can be obtained by taking an element H out of L, permuting the remaining list yielding T, and putting H in front of that.

The basic idea in proving termination of goals is similar to that employed in [Ses88a,Hol91]: to show that *completely ground* arguments are *smaller* in recursive calls. To show that *perm* terminates we must show that the two calls to *append* terminate and that W in the recursive call to *perm* is ground and smaller than whatever ground term L *append* originally was called with.

First, *append* invoked with ground third argument terminates. This is shown by induction on the size l (list length) of the third argument. In the case $l = 0$, the first clause is applied and the assertion is obvious; in the case $l > 0$, the goal terminates if the recursive goal terminates, but the size of the third argument in the recursive call is $l - 1$ and the assertion follows from the induction hypothesis. Similarly *append* with first argument ground terminates.

Second, by induction one shows that *append* invoked with third argument ground returns first and second argument ground, and *append* called with first and second argument ground terminates with third argument ground.

So the goal $append(V, [H|U], L), append(V, U, W)$ terminates with ground W. It now suffices to show that the list length of W is strictly smaller than that of L. To this end, show by induction that whenever $append(A, B, C)$ terminates with ground A, B, C it holds that $l_A + l_B = l_c$, where l_A, l_B, l_C are the list lengths of the first, second, and third argument, respectively, after the satisfaction. Then we can see that after the first call to *append* has been satisfied, $l_V + l_{[H|U]} = l_L$, and after the second call, $l_V + l_U = l_W$. But $l_U + 1 = l_{[H|U]}$, and so $l_W = l_V + l_U = l_V + l_{[H|U]} - 1 = l_L - 1$, as desired. □

Automatic methods for this kind of proof are given in [Ull88,Plu90]. The need for reasoning about groundness is eliminated by assuming that programs are in a certain normal form, and that they are *data driven*, *i.e.* by assuming a given division of predicate arguments into input and output arguments satisfying reasonable conditions. Such normal forms and divisions can be calculated automatically.

The method calculates various inequalities, roughly *axioms* which relate the output arguments of predicates to their input arguments, and *goals* which are required to hold in order to ensure that arguments in recursive calls are decreasing. It is then tested whether the goal inequalities follow from the axiom inequalities. Since the inequalities all relate expressions only involving addition, the test is decidable (Presburger arithmetic, *i.e.* Peano arithmetic excluding terms and axioms concerning multiplication, is decidable, and inequalities involving only addition are just certain formulas in Presburger arithmetic; see [Ull88,Kle52].) Of course the technique can fail to discover termination, since it is restricted to a form of structural recursion.

The variable W in the above example is called a *local variable*. It represents a *computed value* which is passed on as an argument. The Prolog form $g(A, B), h(B, C)$ with A ground, where g instantiates B to some ground term and passes this term to h which instantiates C to another ground term, resembles the functional term $h\ (g\ a)$ where g computes a value which is passed on to h, which returns a value, see Chapter 7.

Recall that for a function defined as $h\ x \leftarrow h\ (g\ x\ )$, the problem in Holst's analysis for termination of partial evaluation was to ensure that the argument in the recursive call to h is at most as big as x, *i.e.* that g returns something smaller than its argument. In the Prolog form this corresponds to the inequality $l_B \leq l_A$.

So the technique for termination is related to the techniques of ensuring termination of partial evaluation of functional programs. Moreover, we have seen that in supercompilation instantiations similar to the ones in Prolog are performed, and the correspondence will be amplified in Section 14.2.

13.5 Transformation of logic programs

We end this chapter by mentioning briefly techniques to ensure termination of transformation of logic programs.

Most Prolog partial evaluators apply *on-line* techniques similar to the loop-checking mechanisms described in the preceding section although the criteria for pruning are more complicated, see *e.g.* [Bru92, Mar93a,Mar93b,Bol93]. Others apply on-line abstract interpretations [Gal88].

In papers concerned with the pattern matching [Smi91,Gal93], no off-line technique that can handle that example is given.

Pettorossi and Proietti have described the *elimination procedure*, essentially a deforestation algorithm for logic programs extended to account for the fact that arguments can be both input and output in Prolog [Pro91]. This algorithm uses annotation techniques very similar to Chin's in deforestation.

Chapter 14

Stopping the positive supercompiler

To be and not to be; that is the answer.

Piet Hein.

This chapter develops methods of ensuring termination of the positive supercompiler for all programs.

Section 14.1 describes a syntactic method inspired by Chin's method from deforestation. The method prevents $\mathcal{W}$ from ever taking advantage of non-linearity of programs; that is: it basically makes $\mathcal{W}$ yield the same result as deforestation would. Specifically, one does not get KMP style specialized matchers. Section 14.2 investigates why $\mathcal{W}$ actually terminates when applied to the general matcher to see what kind of reasoning a termination ensuring technique must perform to discover that there is no termination problem. Section 14.3 describes a simple on-line technique inspired by this analysis. Section 14.4 shows how the two techniques can be combined into one technique that yields good results on all the programs considered in this thesis.

14.1 A simple off-line strategy

Recall what we have called Chin's technique for deforestation. It basically reasons as follows. Termination can be guaranteed for $M_{1/2}$-treeless terms and programs. Given arbitrary term and program t, p we turn these into essentially $M_{1/2}$-treeless term and program by introducing enough local definitions to get an M_1-treeless term and program.

Our idea in the present setting is similar. Termination for positive supercompilation can be guaranteed whenever the result of positive supercompilation is the same as that of deforestation and termination can be guaranteed for deforestation. By Proposition 9.1.7, the results of the two algorithms are the same when both program and term are linear. Just like treelessness can be obtained by introduction of enough local definitions, so can linearity.

This idea is pursued in the following definition, proposition, and corollary.

Definition **14.1.1** (Actual occurrences of a variable in a term.) For a variable v and a term t, let $\mathcal{O}[\![\, t \,]\!]v$ denote the number of *actual occurrences* of v in t.

$$
\begin{array}{lcll}
\mathcal{O}[\![\, u \,]\!]v & = & 1 & \textbf{if } u \equiv v \\
\mathcal{O}[\![\, u \,]\!]v & = & 0 & \textbf{if } u \not\equiv v \\
\mathcal{O}[\![\, c\, t_1 \ldots t_n \,]\!]v & = & \sum_{i=1}^{n} \mathcal{O}[\![\, t_i \,]\!] & \\
\mathcal{O}[\![\, f\, t_1 \ldots t_n \,]\!]v & = & \sum_{i=1}^{n} \mathcal{O}[\![\, t_i \,]\!] & \\
\mathcal{O}[\![\, g\, t_0\, t_1 \ldots t_n \,]\!]v & = & \sum_{i=0}^{n} \mathcal{O}[\![\, t_i \,]\!] & \\
\mathcal{O}[\![\, t_1 = t_2 \;\rightarrow\; t_3 \,\square\, t_4 \,]\!]v & = & (\max_{i=3}^{4} \mathcal{O}[\![\, t_i \,]\!]) + \sum_{i=1}^{2} \mathcal{O}[\![\, t_i \,]\!] & \\
\mathcal{O}[\![\, \text{let } u = t_1 \text{ in } t_2 \,]\!]v & = & \max_{i=1}^{2} \mathcal{O}[\![\, t_i \,]\!] &
\end{array}
$$

□

A term t has at least one actual occurrence of a variable v iff $t \equiv e(v)$ for some e; see Section 3.3. Given an arbitrary term t we can turn it into an equivalent term t' with at most one actual occurrence of every variable by repeatedly applying the following transformation: if $t \equiv e(v)$ and t contains at least two actual occurrences of v then replace t by let $u = v$ in $e(u)$ where u is a fresh variable. We call this repeated operation *generalization of multiple actual occurrences of variables.* Note that this operation does not disturb the treelessness property.

PROPOSITION **14.1.2** *(Down-grading of positive supercompilation to deforestation.) Let p,t be $M_{1/2}$ program and term with at most one actual occurrence of every variable. Then, in the context of p, $\mathcal{W}[\![t]\!]$ and $\mathcal{S}[\![t]\!]$ yield the same term and program, or both loop infinitely.* □

PROOF: Similar to Proposition 9.1.7. □

COROLLARY **14.1.3** *Let p,t be arbitrary $M_{1/2}$ program and term. Let p'',t'' be the program and term obtained as follows. First generalize non-variable, non-constant arguments in t and the right hand sides of p, yielding t',p'. Then generalize multiple actual occurrences of variables in t' and the right hand sides of p', yielding t'',p''. Then $\mathcal{W}[\![t'']\!]$ terminates in the context of p''.* □

PROOF: The result of the transformation is M_1-treeless term and program in which every variable has at most one actual occurrence. On this term and program $\mathcal{W}$ yields the same as $\mathcal{S}$ which terminates. □

This gives us a termination safe procedure for applying the positive supercompiler to arbitrary terms and programs at the expense of only using the deforestation power of it. This is not so bad for several reasons. First, it shows that nothing is lost by going from deforestation to positive supercompilation: we can always transform the program into another program such that the two programs are essentially indistinguishable by deforestation and so that positive supercompilation on the latter program yields the same as deforestation. Second, deforestation is not such a bad algorithm since it can both eliminate intermediate structures and partially evaluate, and perform a certain amount of theorem proving.

Note that the technique is a non-increasing technique.

We now review the effect of the preceding corollary on the example applications from Chapter 5. For the double append example in Section 5.1, the technique makes no generalizations. For the second example in Section 5.1, the technique makes no generalizations. For the first example in Section 5.2 no generalizations are found. For the Ackerman example, starting from the obvious $M_{1/2}$ program with no local definitions, one generalization is found yielding the program in Section 5.2. For the theorem proving example no generalizations are found, provided that we transform the two sides of the equality independently, as described in Section 9.1.

In conclusion, the technique works perfectly when we use $\mathcal{W}$ for things that we would also use deforestation for. It may also be noted that the technique finds exactly the generalizations that we showed in Section 12.1 for the canonical non-termination patterns for $\mathcal{W}$.

However, whenever the program is non-linear in an essential way and $\mathcal{W}$ takes advantage of this fact, the non-linearity is removed by the introduction of local definitions. So the technique does not give good results with the pattern matching example. More precisely, on the flat program specialized to pattern AAB:

$$loop\ AAB\ s\ AAB\ s$$

$$\begin{array}{lcl}
loop\ []\ ss\ op\ os & \leftarrow & True \\
loop\ (p:pp)\ []\ op\ os & \leftarrow & False \\
loop\ (p:pp)\ (s:ss)\ op\ os & \leftarrow & p = s\ \rightarrow\ loop\ pp\ ss\ op\ os\ \Box\ next\ op\ os \\
 & & \\
next\ op\ [] & \leftarrow & False \\
next\ op\ (s:ss) & \leftarrow & loop\ op\ ss\ op\ ss
\end{array}$$

the program we get from the generalization phase is:

$$\text{let } v = s \text{ in } loop\, AAB\; v\; AAB\; s$$

$$\begin{array}{lcl}
loop\, [\,]\; ss\; op\; os & \leftarrow & True \\
loop\, (p : pp)\; [\,]\; op\; os & \leftarrow & False \\
loop\, (p : pp)\; (s : ss)\; op\; os & \leftarrow & p = s \;\rightarrow\; loop\, pp\; ss\; op\; os \;\Box\; next\; op\; os \\
 & & \\
next\; op\; [\,] & \leftarrow & False \\
next\; op\; (s : ss) & \leftarrow & \text{let } v = op \text{ in } loop\; v\; ss\; op\; ss
\end{array}$$

Transforming this with $\mathcal{W}$ clearly does not yield a KMP matcher, since we have lost the coordination between the current string and the original string (top-most let) and similarly between the patterns (lower let). So the technique has introduced generalizations although, as we know, there is no termination problem.

REMARK **14.1.4** Just as Chin's technique for deforestation can be outsmarted by minor syntactic details of the program, so can the present technique; this is a fundamental draw-back of a syntactic method. One can extend the grammar technique for deforestation to positive supercompilation (the author has done this) but the method is rather complicated, and it turns out that with respect to pattern matching and similar examples it suffers from exactly the same problems as the technique above. Since the gain is thus limited to application of the positive supercompiler in the role of deforestation and partial evaluation, the author has decided that the development is not worthwhile in the present report. □

14.2 Why $\mathcal{W}$ terminates on the pattern matcher

In this section we manually prove that $\mathcal{W}$ terminates on the tail-recursive, general pattern matcher with the purpose of investigating whether such a proof may be found automatically by an off-line analysis.

Consider first the nested matcher once again.

$$match\, [d_1 \ldots d_k]\; ss$$

$$\begin{array}{lcl}
match\; p\; s & \leftarrow & prefix\; p\; s = True \;\rightarrow\; True \;\Box\; next\; p\; s \\
 & & \\
next\; p\; [\,] & \leftarrow & False \\
next\; p\; (s : ss) & \leftarrow & match\; p\; ss \\
 & & \\
prefix\; [\,]\; ss & \leftarrow & True \\
prefix\; (p : ps)\; [\,] & \leftarrow & False \\
prefix\; (p : ps)\; (s : ss) & \leftarrow & p = s \;\rightarrow\; prefix\; ps\; ss \;\Box\; False
\end{array}$$

In Chapter 8 we gave an argument in rather abstract terms that $\mathcal{W}$ always terminates on this term and program. Why *does* $\mathcal{W}$ terminate on this program? Well, neither non-increasing nor decreasing techniques suffice. The transformation graph in Chapter 8 for pattern AAB reveals that there is a path from the node containing $loop\, AAB\; u\; AAB\; u$ to a node $loop\, AAB\; (A : s : ss)\; AAB\; (A : s : ss)$. (That transformation graph is for the flat version, but the nested version is similar). No arguments have decreased, but two have *increased.* So in terms of our termination patterns from Section 12.2 the only remaining possibility ia a bounded criterion argument.

The idea in this argument is as follows. Whenever transformation chooses the call to $prefix$ in $match$ as a redex, then $|s| \leq |p|$, where p, s are the first and second argument in the call to $prefix$, respectively. Let us call the string and pattern at this point s_0, p_0 and their lengths $|s_0|, |p_0|$. Now $prefix$ is called recursively a number of times. Compared to the previous call, each new call to $prefix$ has a p argument 1 smaller. In every new call, the s argument is the same as the preceding if the preceding s argument was a variable, otherwise it is 1 smaller. While this is happening, there is a call to $next$ in the context

(in the body of *match*) waiting for the tail-recursive calls to *prefix* to finally chose clause (1) or (2) of *prefix*. When this happens, the argument p in the call to *next* is p_0, but what is s compared to s_0? Well, every time s was decreased by 1 in the recursive calls to *prefix* nothing happens to the s in the context, but when the s argument in the recursive calls to *prefix* has become a variable, every recursive call to *prefix* increases the s in the context by 1. This can happen at most $|p_0| - |s_0|$ times. For there are (at most) $|p_0|$ recursive calls, but in the $|s_0|$ first calls, the s argument is decreased by 1 because it is not a variable. So, when *prefix* is done, the s in the context has length at most $|s_0| + (|p_0| - |s_0|) = |p_0|$, and so after the call from *next* to *match*, the invariant that we started out with is maintained.

Here we manipulated various kinds of information. To see exactly what a supposed static analysis must deduce, let us make the argument precise:

PROPOSITION **14.2.1** *$\mathcal{W}[\![\, loop\,[d_1 \ldots d_k]\, ss\, [d_1 \ldots d_k]\, ss \,]\!]$, where $[d_1 \ldots d_k]$ is a completely known list with k elements, and ss is a variable, terminates in the context of the tail-recursive pattern matcher.* □

PROOF: We consider the following version:

$$\begin{array}{llcl}
(0) & & & loop\,[d_1 \ldots d_k]\, ss\, [d_1 \ldots d_k]\, ss \\
(1) & loop\,[]\, ss\, op\, os & \leftarrow & True \\
(2) & loop\,(p : pp)\,[]\, op\, os & \leftarrow & False \\
(3) & loop\,(p : pp)\,(s : ss)\, op\, os & \leftarrow & p = s \rightarrow loop\, pp\, ss\, op\, os \;\Box\; next\, op\, os \\
 & & & \\
 & next\, op\, [] & \leftarrow & False \\
 & next\, op\, (s : ss) & \leftarrow & loop\, op\, ss\, op\, ss
\end{array}$$

We are to show that no branch in the transformation tree can contain infinitely many terms that are not identical MVR. Every infinite path in the tree, if any, must contain infinitely many nodes that contain a term of form $loop\, t_1\, t_2\, t_3\, t_4$. We imagine that paths between any two of these nodes are contracted so that the tree only contains these nodes with these terms. Clearly this new tree is finite iff the original was.

The idea is that no argument ever exceeds the size of $[d_1 \ldots d_k]$. We use a size measure on terms without function calls slightly different from $|\bullet|$ (it is the list length):

$$\begin{array}{lcl}
\mathcal{L}[\![\, v \,]\!] & = & 0 \\
\mathcal{L}[\![\, C \,]\!] & = & 0 \\
\mathcal{L}[\![\, (t : ts) \,]\!] & = & 1 + \mathcal{L}[\![\, ts \,]\!]
\end{array}$$

where C is any 0-ary constructor. We proceed in 5 steps.

(i) The first and third argument to *loop* are always ground. *Proof:* Induction on the length of the path from the root to the node.

(ii) The fourth argument always has form $t_1 : \ldots t_i : ss$, where $0 \leq i$, and the t's do not contain ss. The second argument always has form $t_1 : \ldots t_j : ss$, $0 \leq i \leq j$ and the t's do not contain ss. *Proof:* Induction on the length of the path from the root to the node. To spell it out: verify that the initial term has the property, and verify for any call $loop\, t_1\, t_2\, t_3\, t_4$ that if this call has the property then the next term of this form arising from a recursive call in the true-branch of clause (3) or from a recursive call from the second clause of *next* via the false-branch of clause (3) also has the property.

(iii) Suppose that a node in some branch contains $loop\, t_1\, t_2\, t_3\, t_4$. Then

$$S \mapsto^* (\mathcal{L}[\![\, t_1 \,]\!], \mathcal{L}[\![\, t_2 \,]\!], \mathcal{L}[\![\, t_3 \,]\!], \mathcal{L}[\![\, t_4 \,]\!])$$

where $\mapsto^*$ is the transitive, reflexive closure of the following non-deterministic rewrite relation on quadruples of non-negative integers defined by pattern matching.

$$\begin{array}{llcl}
(\alpha) & S & \mapsto & (k, 0, k, 0) \\
(\beta) & (1 + n_{pp}, 0, n_{op}, n_{os}) & \mapsto & (n_{pp}, 0, n_{op}, n_{os} + 1) \\
(\gamma) & (1 + n_{pp}, 0, n_{op}, n_{os}) & \mapsto & (n_{op}, n_{os}, n_{op}, n_{os}) \\
(\delta) & (1 + n_{pp}, 1 + n_{ss}, n_{op}, n_{os}) & \mapsto & (n_{pp}, n_{ss}, n_{op}, n_{os}) \\
(\epsilon) & (1 + n_{pp}, 1 + n_{ss}, n_{op}, 1 + n_{ost}) & \mapsto & (n_{op}, n_{ost}, n_{op}, n_{ost})
\end{array}$$

Proof: By induction on the length l of the path from the root of the transformation tree to the node containing $loop\, t_1\, t_2\, t_3\, t_4$. For $l = 0$ the assertion follows by rule (α) of $\mapsto$, since the list $[d_1, \ldots, d_k]$ has length k. Now suppose that the length is $l > 0$; the induction hypothesis says that the parent to our node contains $loop\, t'_1\, t'_2\, t'_3\, t'_4$ with $S \mapsto^* (\mathcal{L}[\![t'_1]\!], \mathcal{L}[\![t'_2]\!], \mathcal{L}[\![t'_3]\!], \mathcal{L}[\![t'_4]\!])$.

In the step from $loop\, t'_1\, t'_2\, t'_3\, t'_4$ to $loop\, t_1\, t_2\, t_3\, t_4$ rule (3) for $loop$ was used, so $t'_1 \equiv (t_p : t_{pp})$ for some ground t_p, t_{pp}.

Now there are four possibilities.

A) t'_2 is a variable and the true branch in (3) was used. By (ii) we then have $t'_2 \equiv ss$. Then ss is instantiated to $s : ss$ the tail of which is used as first argument in the recursive call, so $\mathcal{L}[\![t_1]\!]+1 = \mathcal{L}[\![t'_1]\!]$, $\mathcal{L}[\![t_3]\!] = \mathcal{L}[\![t'_3]\!]$, $\mathcal{L}[\![t_4]\!] = \mathcal{L}[\![t'_4]\!]$. Further, by (ii) we exactly one occurrence of ss in t'_4, and this is also instantiated to $(s : ss)$, so $\mathcal{L}[\![t_2]\!] = \mathcal{L}[\![t'_2]\!] + 1$. Now it follows by rule (β) of $\mapsto$ that

$$(\mathcal{L}[\![t'_1]\!], \mathcal{L}[\![t'_2]\!], \mathcal{L}[\![t'_3]\!], \mathcal{L}[\![t'_4]\!]) \mapsto (\mathcal{L}[\![t_1]\!], \mathcal{L}[\![t_2]\!], \mathcal{L}[\![t_3]\!], \mathcal{L}[\![t_4]\!])$$

yielding the desired result.

B) t'_2 is a variable and the false branch in (3) was used. In this case t'_2 is also ss, and ss has exactly one occurrence in t'_4, and both become instantiated to $(s : ss)$, increasing the fourth argument by 1; $next$ is then called with the third and fourth argument, decreases the fourth argument by 2 and calls $loop$ again. It is easy to see that rule (γ) of $\mapsto$ does the job.

C) t'_2 is not a variable and the true branch in (3) was used. This case is similar to A) except that the fourth argument does not grow since nothing is instantiated.

D) t'_2 is not a variable and the false branch in (3) was used. This case is similar to B) with the same difference as between A) and C).

This concludes the proof of (iii) .

(iv) In every (x_1, x_2, x_3, x_4) reachable from S by $\mapsto$, it holds that $x_1 + x_4 = x_2 + x_3$. *Proof:* Induction in the length of the derivation.

(v) In every (x_1, x_2, x_3, x_4) reachable from S by $\mapsto$ in l steps, it holds that $x_1, x_2, x_4 \leq x_3 \leq k$. *Proof:* x_3 never changes and is initially k, so the latter inequality is obvious. That $x_1 \leq x_3$ is easy to prove by induction on l. Finally, $x_2, x_4 \leq x_3$ is proved by induction on l as follows. If $l = 0$ the assertion holds. For the induction step first note that any sequence must apply rule (β) or (γ) at least once since the second argument is initially 0. Now write the derivation of (x_1, x_2, x_3, x_4) as

$$S \mapsto^* (x''_1, x''_2, x''_3, x''_4) \mapsto (x'_1, x'_2, x'_3, x'_4) \mapsto^* (x_1, x_2, x_3, x_4)$$

where the rewriting from $(x''_1, x''_2, x''_3, x''_4)$ to (x'_1, x'_2, x'_3, x'_4) apply rule (β) or (γ), and the rewritings from (x'_1, x'_2, x'_3, x'_4) to (x_1, x_2, x_3, x_4) apply only rules (δ) and (ϵ).

We consider two cases. 1) If the latter sequence is in 0 steps, then $(x'_1, x'_2, x'_3, x'_4) = (x_1, x_2, x_3, x_4)$, and we consider two subcases. 1a) If rule γ was used then the induction hypothesis for $(x''_1, x''_2, x''_3, x''_4)$ gives the result. 1b) If rule β was used, $x_2 = 0 \leq x_3$. Since $0 \leq x_4$, (iv) shows that $x_1 \leq x_3$.

2) If the sequence of $(\delta), (\gamma)$ rewritings is not empty, the induction hypothesis says that $x'_2, x'_4 \leq x'_3$. Now note that this property is not disturbed by $(\delta) - (\epsilon)$ reductions.

Since all arguments to $loop$ in the terms in the nodes in the transformation tree have finite size, and the terms are built from symbols of a finite alphabet, except for varibles, the transformation tree must be quasi-finite MVR. □

In the proof we are manipulating three kinds of information: 1) *groundness*; 2) *linearity*; 3) *size*. 1) and 2) are manipulated in step (ii) and (iii) of the proof; 3) is manipulated in step (iv) and (v). Let us supose that we have an analysis which gives us 1),2) so that we need only worry about 3).

It is still not easy to see how this proof could be automated. In the case of the nested version, the call structure suggested that we should try to show that $prefix$ terminated and that a certain bounded property was invariant. In the flat case this is far less obvious.[1] Using techniques such as those in [Ull88,Plu90] (see later) one could aim towards a technique that would show that calls $prefix\ p\ s$ always

[1] Plümer [Plu90] points out that his technique, which also manipulates certain boundedness information, works far better on nested programs.

terminate, and do so with $|s| \leq |p|$, and use this to show quasi-termination of g. Such a technique *can* be devised, but it is hard to give general criteria stating: which arguments should be chosen for comparison; when one should require termination of subcalls or just quasi-termination (encountering of only finitely many different states); and in each of these two cases which further requirements must be satisfied.

In conclusion the reason for termination of $\mathcal{W}$ on the matchers is that all arguments are bounded by the original pattern, but devising a static analysis that will discover this for the nested matcher is rather hard, and in the case of the flat version, *very* hard.

In the next section we show a ridiculously simple on-line strategy for flat programs that works for the flat matcher.

14.3 A simple on-line strategy

We have seen that the part of $\mathcal{W}$'s power strictly extending deforestation only comes into work when we have programs with multiple actual occurrences of a variable and this varible becomes instantiated in an equality test or in a pattern matching. When a call such as $e[g v v]$ or conditional such as $e[v = b \rightarrow v \,\square\, t']$ instantiates the second occurrence of v, then that occurrence becomes instantiated to bits and pieces of constants of the original program and this program's inputs. For instance, in the example with the general pattern matcher, the variable ss becomes instantiated to parts of the pattern. A reasonable security check is therefore to ensure that things that become more and more instantitated never exceed the size of the largest constant in the program. This is a simple *bounded* criterion similar to *static domination* described in the second subsection of Section 13.2.

The idea is corrupted by the presence of nested calls, because there is no clear intuition in comparing the size of such calls to constants. Rather than devising some bounded property for the nesting of calls and argument sizes in contexts, we state a version of $\mathcal{W}$, incorporating a bounded check,, for M_1-constant treeless terms and programs, see Definition 13.1.3. Terms and programs in this class contain no nested calls.

DEFINITION **14.3.1** (On-line termination technique for flat programs.) Given program and term p, t. Let M be the size of the largest passive term appearing in the program and term. Let t, b, a range over the syntactic classes in Definition 13.1.3. Define $\mathcal{G}$, the positive supercompiler for constant-treeless programs with on-line bounded-generalization as follows.

$$
\begin{array}{lll}
(0) & \mathcal{G}[\![\, v \,]\!] & \Rightarrow\; v \\
(1a) & \mathcal{G}[\![\, c\, t_1 \ldots t_n \,]\!] & \Rightarrow\; c\, t_1 \ldots t_n \\
 & \quad \textbf{if } \text{all } t_i \text{ are passive} & \\
(1b) & \mathcal{G}[\![\, c\, t_1 \ldots t_n \,]\!] & \Rightarrow\; c\, (\mathcal{G}[\![\, t_1 \,]\!]) \ldots (\mathcal{G}[\![\, t_n \,]\!]) \\
 & \quad \textbf{if } \text{not all } t_i \text{ are passive} & \\
(X) & \mathcal{G}[\![\, h\, a_1 \ldots a_n \,]\!] & \Rightarrow\; \text{let } v_{i_1} = a_{i_1} \ldots v_{i_k} = a_{i_k} \\
 & & \quad \text{in } \mathcal{W}'[\![\, (h\, u_1 \ldots u_n)\{u_{i_j} := v_{i_j}\}_{j=1}^{k}\{u_{i_j} := a_{i_j}\}_{j=k+1}^{n} \,]\!] \\
 & \quad \textbf{where } |a_{i_1}|, \ldots |a_{i_k}| > M & \\
 & \quad \textbf{where } |a_{i_{k+1}}|, \ldots |a_{i_n}| \leq M & \\
(4a) & \mathcal{G}[\![\, b = b' \;\rightarrow\; t \,\square\, t' \,]\!] & \Rightarrow\; \mathcal{G}[\![\, t \,]\!] \\
 & \quad \textbf{if } b, b' \text{ are ground and } b \equiv b' & \\
(4b) & \mathcal{G}[\![\, b = b' \;\rightarrow\; t \,\square\, t' \,]\!] & \Rightarrow\; \mathcal{G}[\![\, t' \,]\!] \\
 & \quad \textbf{if } b, b' \text{ are ground and } b \not\equiv b' & \\
(4c) & \mathcal{G}[\![\, b = b' \;\rightarrow\; t \,\square\, t' \,]\!] & \Rightarrow\; b = b' \;\rightarrow\; \mathcal{G}[\![\, tMGU(b, b') \,]\!] \,\square\, \mathcal{G}[\![\, t' \,]\!] \\
 & \quad \textbf{if } \text{not both } b, b' \text{ are ground} & \\
(5) & \mathcal{G}[\![\, \text{let } v = t \text{ in } t' \,]\!] & \Rightarrow\; \text{let } v = \mathcal{G}[\![\, t \,]\!] \text{ in } \mathcal{G}[\![\, t' \,]\!]
\end{array}
$$

$$
\begin{array}{lll}
(2) & \mathcal{W}'[\![\, f\, a_1 \ldots a_n \,]\!] & \Rightarrow\; f^{\square}\, u_1 \ldots u_k \\
 & \quad \textbf{where} & \\
 & \qquad f^{\square}\, u_1 \ldots u_k \leftarrow \mathcal{G}[\![\, t^f\{v_i^f := a_i\}_{i=1}^{n} \,]\!] & \\
(3a) & \mathcal{W}'[\![\, g\, (c\, a_{n+1} \ldots a_{n+m})\, a_1 \ldots a_n \,]\!] & \Rightarrow\; f^{\square}\, u_1 \ldots u_k \\
 & \quad \textbf{where} & \\
 & \qquad f^{\square}\, u_1 \ldots u_k \leftarrow \mathcal{G}[\![\, t^{g,c}\{v_i^{g,c} := a_i\}_{i=1}^{n+m} \,]\!] & \\
(3b) & \mathcal{W}'[\![\, g\, v\, a_1 \ldots a_n \,]\!] & \Rightarrow\; g^{\square}\, v\, u_1 \ldots u_k \\
 & \quad \textbf{where} & \\
 & \qquad g^{\square}\, p_1\, u_1 \ldots u_k \leftarrow \mathcal{G}[\![\, t^{g,c_1}\{v_i^{g,c_1} := a_i\}_{i=1}^{n}\{v := p_1\} \,]\!] & \\
 & \qquad \vdots & \\
 & \qquad g^{\square}\, p_m\, u_1 \ldots u_k \leftarrow \mathcal{G}[\![\, t^{g,c_m}\{v_i^{g,c_m} := a_i\}_{i=1}^{n}\{v := p_m\} \,]\!] &
\end{array}
$$

□

In all clauses except clause (X) the result is as in the formulation of $\mathcal{W}$. In clause (X), the constants that have exceeded the size M are generalized on-line, and then control is passed to $\mathcal{W}'$ which proceeds like $\mathcal{W}$.

We do not bother to make the notion of termination precise by transformation trees and graphs, but we can give a reasonably precise proposition and proof of termination:

PROPOSITION **14.3.2** *For arbitrary constant-treeless term and program t, p, $\mathcal{G}$ terminates.* □

PROOF: Let M be size of the largest passive term occurring in t or a right hand side in p, let K be the size of the largest pattern of a g-function in p, let N be the size of the largest term among t and the right hand sides of p.

Recall from Section 12.3 the measure (m, n) where m is the number of conditionals in t and n is $|t|$, see Section 2.2. Clearly each step of $\mathcal{G}$ except possibly clause (X) strictly decreases this measure. In clause (X), the next term encountered by $\mathcal{G}$ has measure at most (N, MNK).

But then all terms encountered by $\mathcal{G}$ has at most measure (N, MNK). This means that $\mathcal{G}$ encounters only finitely many different terms MVR, and so the folding scheme will eventually fold all branches of transformation. □

To see that we have actually achieved anything at all, note that applying $\mathcal{G}$ to the flat pattern matcher with pattern AAB yields the almost KMP matcher in Example 8.4.1.

One can also state a version of $\mathcal{G}$ that applies Turchin's on-line generalization technique, which is somewhat more complicated, also in the case of no nested function calls. This version would also be able to derive efficient matchers, but relies, in our terms, of a more powerful postunfolding phase which also unfolds conditionals with tests that can be calculated and which also unfolds g-function calls with sufficiently know pattern arguments. In the output of $\mathcal{G}$ such sources of postunfolding are never present.

14.4 On-line or off-line?

The question is classical in partial evaluation: off-line or on-line binding-time analysis? In that field the main arguments for one or the other is that on-line specializers have more information available and thus can give better specialization, whereas off-line specializers are more suitable for self-application, see [Jon93, Chapter 7].

Here we are only concerned with the question of termination and not self-application or binding-times analysis. We have seen in the preceding sections that there are two patterns we would like to recognize: the non-increasing criterion and the bounded criterion.

This resembles the situation in [Jon88a] with the principles dynamic construction and static domination, respectively. In that paper, Jones argues rather forcefully against an on-line technique. The preference towards off-line techniques is motivated by the difficulties with on-line techniques and self-application in the early Mix project. However it is still true, as argued in [Jon88a], that incorporating an increasing criterion is hard on-line. This is easier off-line.

On the other hand, the preceding section showed that deciding statically whether parameters are bounded by input parameters can also be very hard. On-line this is trivial.

We can get the best of both worlds as follows. First generalize the program to arrive at an M_1-treeless program. Then apply the deforestation algorithm $\mathcal{S}$ to this program. Then apply $\mathcal{G}$ to the output of $\mathcal{S}$; the output is in just the right format!

For instance, applying $\mathcal{S}$ to either of the general matchers and then applying $\mathcal{G}$ to the output yields the almost KMP style matcher, although the passive term M in both cases must be the largest passive term in the program *before applying* deforestation. This is reasonable enough since the original constants are taken apart and frozen in tests by $\mathcal{S}$, so that the subsequent instantiations of $\mathcal{G}$ recover these parts. This seems to be a rather general phenomenon.

So, applying the combined technique in fact works well on all the examples considered in this thesis. The technique also works for examples where the passive term is not ground. For instance, on neither the nested nor the flat matcher with pattern $[x_1, x_1, x_2]$ will the combined technique come up with any generalizations.

The combined technique has several good properties. The off-line technique works well when $\mathcal{W}$ was used in the role of $\mathcal{S}$, and the on-line mechanism in $\mathcal{G}$ works well for eliminating non-linearity. In the combined version each technique is used for exactly what it is good at. Also, the problem of termination of $\mathcal{W}$ has been factored into two simpler problems. We can replace the technique in the first phase by better techniques from deforestation such as that in [Sor94a], and similarly we can replace the technique in the second phase by more powerful techniques.

That the factorization is possible shows that $\mathcal{S}$ does not blur sources of efficiency that $\mathcal{W}$ could otherwise have removed. This means that nothing is lost by deforesting before applying $\mathcal{G}$, except perhaps run-time efficiency of the transformer. However, $\mathcal{G}$ may be implemented efficiently by taking into account the restricted class of programs with which it deals.

Whether the combined technique should be taken seriously or just considered as an interesting factorization of $\mathcal{W}$ providing insight remains to be seen.

Chapter 15

Conclusion

Hey! There is no quotation on Chapter 15.

Olivier Danvy.[1]

The first section gives an overview of the achievements of the thesis. The second section briefly reviews research with similar goals, and the last section outlines directions for further research.

15.1 What have we achieved

We have described the positive supercompiler, a new formulation of the essential principle of driving underlying Turchin's supercompiler, and we have given a number of properties of the positive supercompiler and its relation to other transformers. In more detail:

1. We have described the positive supercompiler, a reformulation of the notion of driving, in simpler and more familar terms than those in which driving is usually described. The positive supercompiler always preserves the call-by-name semantics, and by well-known techniques also the lazy semantics of programs. It has a simple explanation in Burstall-Darlington terms, and often has the same power as Turchin's supercompiler. We have also explained the essence of driving as the propagation of positive information.

   The reformulation means that it becomes easier to answer many questions concerning correctness, termination, *etc.* and leads to a better understanding of how supercompilation achieves its effects and of its relation to other transformers (see below.)

2. Supercompilation is traditionally explained as the construction of certain graphs. We have given a technical development of that approach and explained its connection to the formulation as a Burstall-Darlington style transformer.

3. We have shown examples of the effects that supercompilation is traditionally shown to be capable of: program specialization, elimination of intermediate data structures, and theorem proving.

   Rather than just showing the input-output behaviour we were able to explain precisely what it was in the formulation of the positive supercompiler that achieved the various effects, and we have given technical results stating that the positive supercompiler achieved these effects optimally, in a certain sense.

4. In partial evaluation and supercompilation different ways of expressing transformation, in particular self-application, are traditionally used. We have explained the difference precisely in familiar terms and in a way independent of any programming language.

[1]Danvy's first repsonse to this thesis in personal communication.

We also gave a very simple intuition on the Futamura projections which explains specializers in a simple way as programs which change the functionality of functions.

5. We have given a precise correspondance between logic programming and positive supercompilation, showing that driving, in a certain sense, is similar to interpretation of logic programs.

6. We have given yet another in depth investigation of pattern matching. Due to the simplicity of the formulation of positive supercompiler we were able to prove rigorously that the specialized matchers have the same complexity, in a certain sense, as the matchers output by the Knuth-Morris-Pratt algorithm. This shows that positive information propagation suffices to get Knuth-Morris-Pratt style matchers. We also showed that a simple folding scheme wih looping back to identical configurations in general suffices to ensure termination of positive supercompilation of pattern matchers.

7. We have given an in-depth comparison of supercompilation and related transformers: partial evaluation, deforestation, and generalized partial evaluation. In terms of information propagation we described the differences betweem these transformers, thereby explaining why partial evaluation and deforestation cannot generate Knuth-Morris-Pratt style matchers. We also explained in terms of evaluation order why partial evaluation does not generally eliminate intermediate data structures.

   The two notions of information propagation and evaluation order provide a good handle to classify, compare, and understand various transformers.

8. We have proved that the positive supercompiler preserves the call-by-name semantics of programs; in particular, the residual program has exactly the same termination properties as the original program. The proof includes the postunfolding phase and the prephase performing generalizations.

9. We gave a result stating that positive supercompilation can give only linear speedups, in a certain sense, thereby clarifying the theoretically possible gains from supercompilation. We have also shown that the positive supercompiler never introduces constructs in the residual program that it could improve itself: the positive supercompiler is idempotent.

10. We have given a general syntactic characterication of infinite transformation, and shown three canonical patterns that are instances of this pattern, isolating more concretely the termination problem caused by positive information propagation. We have shown that the termination problem for positive supercompilation is recursively unsolvable. We have also shown that no matter how clever we are in choosing concrete programs to represent abstract functions, the need for generalizations in principle is inevitable.

11. We have explained the difficulties of on-line and off-line methods in connection with various degrees of information propagation and different "evaluation orders" for transformers.

    We have given an off-line and an on-line technique to ensure termination of the positive supercompiler, and a way to combine the two. This method works well when the positive supercompiler works in the role of deforestation and partial evaluation as well as in the generation of efficient matchers. This clarifies what kind of techniques a method to ensure termination and yet not prevent these effects must perform.

15.2 Related work

We shall not mention all works that have results similar to results in one or more chapters in this thesis. The overall purpose of clarifying the essence of supercompilation is also the motivation for the paper by Glück and Klimov [Glu93a]. Many of the insights above are contained in one or more papers by Glück and one or more papers by Turchin.

15.3 Future work

Although several of the insights mentioned in the first section are new and interesting there is still some way to go before the ideas can turn into research papers. The following ideas are the most concrete and promising:

1. The connection between driving and logic programming has been mentioned by Glück in [Glu92a], but has never been investigated in detail. Chapter 7 contains the beginning of such an investigation.

2. Given such an investigation, there are several applications. In logic programming, interpretation and partial evaluation are very similar operations in terms of SLD-trees, and both tasks are very similar to positive supercompilation. A technique succesfully ensuring termination of a partial evaluator for Prolog is, in the author's opinion, likely to be successful for ensuring termination of the supercompiler too, and vice versa.

3. The significance of call-by-name transformation as opposed to call-by-value transformation, outlined in Section 9.2, also deserves a detailed study. The idea of obtaining deforestation by CPS and partial evaluation is "in the air" at the moment, but it is the authors belief that to truly obtain deforestation, the CPS transformation must be call-by-name. The outcome of such an investigation could be a major clarification.

Bibliography

[Abr87] S. Abramsky, C. Hankin. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, London, 1987.

[Acz77] P. Aczel. An Introduction to Inductive Definitions. In *Handbook of Mathematical Logic.* (Ed.) J. Barwise, North-Holland, 1977.

[Aho86] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques and Tools .* Addison-Wesley, 1986.

[Amb87] T. Amble. *Logic Programming And Knowledge Engineering.* Addison-Wesley, 1986.

[And86] N. Andersen. *Approximating Term Rewriting Systems With Tree Grammars.* DIKU-report 86/16, Institute of Datalogy, University of Copenhagen, 1986.

[And92] L. O. Andersen, C. K. Gomard. Speedup Analysis in Partial Evaluation (preliminary results). In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992.* (Technical Report YALEU/DCS/RR-909) pp1-7, 1992.

[Aug85] L. Augustsson. Compiling Lazy Pattern-Matching. In *Conference on Functional Programming and Computer Architecture.* LNCS 201, 1985.

[Bar84] H.P. Barendregt. *The Lambda Calculus, its Syntax and Semantics.* Studies In Logic And The Foundations Of Mathematics, vol 103. Revised Edition. North-Holland, Amsterdam, 1984.

[Bec75] L. Beckman *et al.*. A Partial Evaluator, and its Use as a Programming Tool. In *Artificial Intelligence.* 7(4):319-357,1976.

[Bir77] R. S. Bird. Improving Programs by the Introduction of Recursion. In *Communications of the ACM.* Vol.20, No.11, pp.856-863, 1977.

[Bir80] R. S. Bird. Tabulation Techniques for Recursive Programs. In *Computing Surveys.* Vol. 12, No 4, December 1980.

[Bir80] R. S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. In *Acta Informatica.* 21, pp. 239-250, 1984.

[Bir88] R. S. Bird and P. L. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[Bol93] R. Bol. Loop Checking in Partial Deduction. In *Journal of Logic Programming.* 16 (1 & 2), May 1993.

[Bon90a] A. Bondorf, O. Danvy. *Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data Types.* DIKU-Rapport 90/4, Department of Computer Science, University of Copenhagen, 1990.

[Bon90b] A. Bondorf. *Self-Applicable Partial Evaluation.* Ph.D. thesis, DIKU-Rapport 90/17, Department of Computer Science, University of Copenhagen, 1990.

[Bon91a] A. Bondorf, O. Danvy. Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data Types. In *Science of Comuter Programming.* 16, pp151-195, 1991.

[Bon91b] A. Bondorf. *Similix Manual.* System Version 4.0. DIKU-Rapport 91/9, Department of Computer Science, University of Copenhagen, 1991

[Bon92] A. Bondorf. Improving Binding Times without Explicit CPS-Conversion. In *ACM Lisp and Functional Programming Conference.* San Francisco, California, June 1992.

[Bra86] I. Bratko. *Prolog Programming For Artificial Intelligence.* Addison-Wesley, 1986.

[Bru92] M. Bruynooghe, D. De Schreye, B. Martens. A General Criterion for Avoiding Infinite Unfolding During Partial Deduction. In *New Generation Computing.* 11(1):47-79, 1992.

[Bul88a] M. A. Bulyonkov. A Theoretical Approach to Polyvariant Mixed Computation. In *Partial Evaluation and Mixed Computation.* Eds. A. P. Ershov, D. Bjørner, N. D. Jones, North-Holland 1988.

[Bul88b] M. A. Bulyonkov, A. E. Ershov. How Do Ad-Hoc Compiler Constructs Appear in Universal Mixed Computation Processes?. In *Partial Evaluation and Mixed Computation.* Eds. A. P. Ershov, D. Bjørner, N. D. Jones, North-Holland 1988.

[Bur77] R. M. Burstall, J. Darlington. A Transformation System for Developing Recursive Programs. In *Journal of the ACM.* Vol. 24, No. 1. January 1977.

[Chi90] W.-N. Chin. *Automatic Methods for Program Transformation.* Ph.D. thesis, Imperial College, University of London, July 1990.

[Chi92a] W.-N. Chin. Fully Lazy Higher-Order Removal. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* Yale University, 1992.

[Chi92b] W.-N. Chin. Safe Fusion of Functional Expressions. In *ACM Lisp and Functional Programming Conference.* San Francisco, California, June 1992.

[Chi93a] W.-N. Chin. Safe Fusion of Functional Expressions II: Further Improvements. Accepted for *Journal of Functional programming.* 1994.

[Chi93b] W.-N. Chin. Towards an Automated Tupling Strategy. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* Copenhagen, Denmark, 1993.

[Chi93c] W.-N. Chin. *A Modular Strategy for Combining the Fusion and Tupling Methods.* Unpublished manuscript. 1993.

[Chi93d] W.-N. Chin. Tupling Functions with Multiple Recursion Parameters. In *3rd International Workshop on Static Analysis, Padova, Italy.* Lecture notes in Computer Science, vol 724, 1993.

[Coh83] N. H. Cohen. Eliminating Redundant Recursive Calls. In *ACM Transactions on Programming Languages and Systems.* Vol. 5 No 2, April 1983.

[Con86] R. Constable *et al.*. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[Con89] C. Consel, O. Danvy. Partial Evaluation of Pattern Matching in Strings. In *Information Processing Letters.* Vol.30, No.2, pp.79-86, 1989.

[Con91] C. Consel, O. Danvy. For a Better Support of Static Data Flow. In *Conference on Functional Programming and Computer Architecture, Cambridge, Massachusetts.* (Ed.) John Hughes, Lecture Notes in Computer Science Vol.523, pp495-519, 1991.

[Con93] C. Consel, O. Danvy. Tutorial Notes on Partial Evaluation. In *20th ACM Symposium on Principles of Programming Languages.* Charleston, South Carolina, pp.493-501, ACM Press 1993.

[Cov85a] M. A. Covington. Eliminating Unwanted Loops in Prolog. In *SIGPLAN Notices*. Vol 20, No 1, January, 1985.

[Cov85b] M. A. Covington. A Further Note On Looping in Prolog. In *SIGPLAN Notices*. Vol 20, No 8, August, 1985.

[Dan91] O. Danvy. Semantics-Directed Compilation of Non-Linear Patterns. In *Information Processing Letters*. Vol.37, pp.315-322, March 1991.

[Dar81] J. Darlington. An Experimental Program Transformation and Synthesis System. In *Artificial Intelligence*. 16, 1981.

[Dav82] M. Davis. *Computability and Unsolvability*. Reprint (originally published in 1958) Dover, 1982.

[Der87] Nachum Dershowitz. Termination of Rewriting. In *Journal of Symbolic Computation*. 3, 1987.

[Der90] Nachum Dershowitz, Jean-Pierre Juannaud. Rewrite Systems. In *Handbook of theoretical Computer Science*. 1990.

[DeS93] D. De Schreye, S. Decorte. *Termination of Logic Programs: the Never-Ending Story*. to appear.

[Ede85] E. Eder. Properties of Substitutions and Unifications. In *Journal of Symbolic Computation*. No 1, pp31-46, 1985.

[Ers77] A. P. Ershov. On the Partial Computation Principle. In *Information Processing Letters*. 6,2 pp.38-41,1977.

[Ers78] A. P. Ershov. On the Essence of Compilation. In *Formal Description of Programming Concepts*. (Ed.) E.J. Neuhold, pp.391-420, Noth-Holland, 1978.

[Ers82] A. P. Ershov. Mixed Computation: Potential Applications and Problems for Study. In *Theoretical Computer Science*. 18, pp.41-67, 1983.

[Ers88] A.P. Ershov. Opening Key-Note Speech at Partial Evaluation and Mixed Computation. In *Partial Evaluation and Mixed Computation*. Eds. A. P. Ershov, D. Bjørner, N. D. Jones, North-Holland 1988.

[Fea82] M. S. Feather. A System for Assisting Program Transformation. In *ACM Transaction on Programming Languages and Systems*. 4(1), 1982.

[Fer88] A. B. Ferguson, P. L. Wadler. When will Deforestation Stop?. In *1988 Glasgow Workshop on Functional Programming*. August 1988.

[Fut71] Y. Futamura. Partial Evaluation of Computation Process—an Approach to a Compiler-Compiler. In *Systems, Computers, Controls*. Vol 2. No 5, 1971.

[Fut83] Y. Futamura. Partial Computation of Programs. In *RIMS Symposia on Software Science and Engineering, Kyoto, Japan, 1992*. (Ed.) E. Goto *et al.*, pp.1-35, Lecture Notes in Computer Science Vol 147, 1983.

[Fut88] Y. Futamura, K. Nogi. Generalized Partial Computation. In *Partial Evaluation and Mixed Computation*. Eds. A. P. Ershov, D. Bjørner, N. D. Jones, North-Holland 1988.

[Gal88] J. P. Gallagher, M. Codish, E.Y. Shapiro. Specialisation of Prolog and FCP Programs Using Abstract Interpretation. In *New Generation Computing*. 6:159-186, 1988.

[Gal93] J. P. Gallagher. Tutorial on Specialisation of Logic Programs. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. Copenhagen, Denmark, 1993.

[Gil93] A. Gill, J. Launchbury, S. L. P. Jones. A Short Cut to Deforestation. In *Conference on Functional Programming and Computer Architecture*. Copenhagen, Denmark, 1993.

[Glu89] R. Glück, V. F. Turchin. *Experiments with a Self-Applicable Supercompiler.* Technical Report, City University, New York, 1991.

[Glu90] R. Glück, V. F. Turchin. Application of Metasystem Transition to Function Inversion and Transformation. In *Proceedings of the ISSAC '90, Tokyo, Japan.* pp286-287, ACM press, 1990.

[Glu91a] R. Glück. Towards Multiple Self-Application. In *ACM Symposium on Partial Evaluation and Semantics Based Program Manipulation.* 1991.

[Glu91b] R. Glück. *On the Generation of $S \rightarrow R$-Specializers.* Technical Report, University of Vienna, 1991.

[Glu92a] R. Glück. Projections for Knowledge Based Systems. In *Cybernetics and Systems Research '92.* (Ed.) R. Trappl, pages 535-542, 1992.

[Glu92b] R. Glück. The Requirement of Identical Variety. In *13th International Congress on Cybernetics.* Namur, Belgium, 1992.

[Glu93a] R. Glück, A. Klimov. Occam's Razor in Metacomputation:the Notion of a Perfect Process Tree. In *3rd International Workshop on Static analysis, Padova, Italy.* Lecture Notes in Computer Science vol. 724, 1993.

[Glu94a] R. Glück, J. Jørgensen. Generating Optimizing Specializers. Accepted for *International Conference on Computer Languages (ICCL) '94.* 1993.

[Glu94b] R. Glück, A. Klimov. Metacomputation as a Tool for Formal Linguistic Modelling. Accepted for *Cybernetics and Systems Research '94.* World Scientific: Singapore, 1994.

[Gom90] C. K. Gomard, N. D. Jones. *Compiler Generation By Partial Evaluation: a Case Study.* DIKU-rapport 90/16, Department of Computer Science, University of Copenhagen, 1990.

[Gom91] C. K. Gomard. *Program Analysis Matters.* Ph.D. Thesis, DIKU-rapport 91/14, Department of Computer Science, University of Copenhagen, 1991.

[Gor93] M. J. C. Gordon, T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.

[Ham90] G. W. Hamilton, S. B. Jones. *Compile-Time Garbage Collection by Necessity Analysis.* Technical Report Department of Computing Science and Mathematics University of Stirling, Scotland, 1990.

[Ham91] G. W. Hamilton, S. B. Jones. Extending Deforestation for First Order Functional Programs. In *1991 Glasgow Workshop on Functional Programming.* 1991.

[Ham92a] G. W. Hamilton. *Sharing Analysis of Lazy First Order Functional Programs.* Unpublished manuscript. 1992.

[Ham92b] G. W. Hamilton. *Compile-Time Optimisation of Storage Utilisation for Lazy First Order Functional Programs.* Unpublished manuscript. 1992.

[Ham93] G. W. Hamilton. *Higher Order Deforestation.* Unpublished manuscript. 1993.

[Har78] M. A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, 1978.

[Hig52] G. Higman. Ordering by Divisibility in Abstract Algebras. In *Proc. London Math. Soc.* (3) 2 pp326-336.

[Hol91] C. K. Holst. Finiteness Analysis. In *5th ACM Conference on Functional Programming Languages and Computer Architecture, LNCS 523.* Cambridge, Massachussets, 1991.

[Hud92] P. Hudak *et al.*. Report on the Programming Language Haskell. In *SIGPLAN Notices.* Vol.27, No.5, 1992.

[Hue80a] G. Huet, D. C. Oppen. Equations and Rewrite Rules. A Survey. In *Formal Language Theory-Perspectives and Open Problems.* (Ed.) Ronald V. Book, Academic Press, 1980.

[Hue80b] G. Huet. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. In *Journal of the ACM.* Vol. 27, No 4, pp.797-821, October 1980.

[Hug88] J. Hughes. Backwards Analysis of Functional Programs. In *Partial Evaluation and Mixed Computation.* Eds. A. P. Ershov, D. Bjrner, N. D. Jones, North-Holland, 1988.

[Hug90a] J. Hughes. Why Functional Programming Matters. In *Research topics in Functional Programming.* Ed. D. Turner, Addison-Wesley, 1990.

[Hug90b] J. Hughes. Compile-Time Analysis of Functional Programs. In *Research topics in Functional Programming.* Ed. D. Turner Addison-Wesley, 1990.

[Joh85] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proceedings of the Conference on Functional Programming and Computer Architechture.* Lecture Notes In Computer Science Vol 201, 1985.

[Jon85] N. D. Jones, P. Sestoft, H. Søndergaard. An Experiment in Partial Evaluation: the Generation of a Compiler Generator. In *Rewriting Techniques and Applications, Dijon, France.* (Ed.) J.-P. Jouannaud, Lecture Notes in Computer Science Vol. 202, 1985.

[Jon87a] N. D. Jones, P. Sestoft, H. Søndergaard. *Mix. A Self-applicable Partiel Evaluator for Experiments in Compiler Generation.* DIKU-rapport 87/8, Department of Computer Science, University of Copenhagen, 1987.

[Jon87b] N. D. Jones. *Flow analysis of Lazy higher-order functional programs.* Chapter 15 in [Abr87], 1987.

[Jon88a] N. D. Jones. Automatic Program Specialization: A re-examination from basic principles. In *Partial Evaluation and Mixed Computation.* Eds. A. P. Ershov, D. Bjørner, N. D. Jones, North-Holland, 1988.

[Jon90] N. D. Jones. Partial Evaluation, Self-Application, and Types. In *Automata, Languages, and Programming. 17th International Colloqium, Warwick, England.* (Ed.) M. S. Patterson, Lecture Notes in Computer Science vol. 443, 1990.

[Jon88b] N. D. Jones, T. Andersen. *The Termination Problem in Partial Evaluation.* Unpublished manuscript. 1988.

[Jon93] N. D. Jones, C. K. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[Jor91] J. Jørgensen. Generating a Pattern Matching Compiler by Partial Evaluation. In *Functional Programming, Glasgow 1990.* Ed. S. L. Peyton Jones, G. Hutton and C. K. Holst, pp.177-195, Springer-Verlag 1991.

[Jor92a] J. Jørgensen. Generating a Compiler for a Lazy Language by Partial Evaluation. In *Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Alburquerque, New Mexico.* January,1992.

[Jor92b] J. Jørgensen. *Compiler Generation by Partial Evaluation.* Master's Thesis. DIKU, Department of Computer Science, University of Copenhagen,1992

[Kle52] S. Kleene, Introduction to Metamathematics. Van Nostrand, 1952.

[Klo87] J. W. Klop. Term Rewriteing Systems: a Tutorial. In *Bulletin of the European Association for Theoretical Computer Science.* Nr.32, pp.143-183.

[Knu77] D. E. Knuth, J. H. Morris, V. R. Pratt. Fast Pattern Matching in Strings. In *SIAM Journal on Computing.* Vol.6, No.2, pp.323-350, 1977.

[Kom82] J. Komorowski. Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Therory and Implementation in the Case of Prolog. In *Ninth ACM Symposium on Principles of Programming Languages.* Alberquerque, New Mexisco, pages 255-267,1982.

[Kru60] J. B. Kruskal. Well-quasi-ordering, the Tree Theorem, and Vazsonyi's Conjecture. In *Trans. Amer. Math. Soc.* 95 pp210-225,1960.

[Lau91] J. Launchbury. A Strongly-Typed Self-Applicable Partial Evaluator. In *5th ACM Conference on Functional Programming Languages and Computer Architecture, LNCS 523.* Cambridge, Massachussets, 1991.

[Law93] J. Lawall. Proofs by Structural Induction using Partial Evaluation. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* Copenhagen, Denmark, 1993.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* North-Holland, New York, 1987.

[Llo91] J. W. Lloyd, J.C. Shepherdson. Partial Evaluation in Logic Programming. In *Journal of Logic Programming.* 11 (3, 4), October/Novermber 1991.

[Lom64] L. A. Lombardi. Incremental Computation. In *Advances in Computers.* (Ed.) F.L.Alt, M.Rubinoff, vol. 8, pages 247-333, Academic Press,1964.

[Lom67] L. A. Lombardi, B.Raphael. Lisp as the Language for a Incremental Computer. In *The Programming Language Lisp: Its Operation and Applications.* (Ed.) E.C.Berkeley, D.G.Bobrow, pages 204-219, MIT Press,1967.

[Mal93] K. Malmkær. Towards Efficient Partial Evaluation. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* Copenhagen, Denmark, 1993.

[Mar93a] B. Martens, D. De Schreye. *Advanced Techniques in Finite Unfolding.* Technical reprt CW 182, Dep. Computerwetenschappen, K.U.Leuven, October, 1993.

[Mar93b] B. Martens, D. De Schreye, T. Horváth. Sound and Complete Partial Deduction with Unfolding Based on Well-Founded Measures. Accepted for *Theoretical Computer Science.* Vol 122, 1994.

[Mar92] S. Marlow, P. L. Wadler. Deforestation for higher-order functions. In *Functional Programming, Glasgow 1992.* Ed. J. Launchbury, Workshops in Computing, 1992.

[Mei91] E. Meijer, M. Fokkinga, R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *5th ACM Conference on Functional Programming Languages and Computer Architecture, LNCS 523.* Cambridge, Massachusetts, 1991.

[Men87] E. Mendelson. *Introduction to Mathematical Logic.* Wadsworth 1987.

[Mil78] R. Milner. A therory of type polymorphism in programming. In *Journal of Computer and System Sciences.* 17, 1978.

[Mil90] R. Milner, M. Tofte, R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Mog88] T. Mogensen. Partially Static Structures in a Self-applicable Partial Evaluator. In *Partial Evaluation and Mixed Computation.* Eds. A. P. Ershov, D. Bjørner, N. D. Jones, North-Holland, 1988.

[Mog91] T. Mogensen. *Variants of Unfold/fold Strategies.* Unpublished manuscript. (In Danish.) 1991.

[Mos79] P. Mosses. *SIS - Semantics Implementation System, Reference manual and User Guide.* DAIMI MD-30, University of Aarhus, Denmark, 1979.

[Nas63] C. St. J. A. Nash-Williams. On Well-quasi-ordering finite trees. In *Proc. Cambridge Phil. Soc.* 59 pp833-835, 1963.

[Pal93] J. Palsberg. Correctness of Binding-time Analysis. In *Journal of Functional Programming.* Vol.3 part 3, July 1993.

[Plo75] G. D. Plotkin. Call-by-Name, Call-by-Value and the λ.-Calculus. In *Theoretical Computer Science.* 1, 1975.

[Poo85] D. Poole, R. Goebe. On Eliminating Loops in Prolog. In *SIGPLAN Notices.* Vol 20, No 8, August, 1985.

[Pre93] S. Prestwich. Online Partial Deduction of Large Programs. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* Copenhagen, Denmark, 1993.

[Pro90] M. Proietti, A. Pettorossi. Synthesis of Eureka Predicates for Develping Logic Programs. In *Proceedings of ESOP '90, LNCS 432.* pp305-325, Copenhagen, Denmark, 1990.

[Pro91] M. Proietti, A. Pettorossi. Unfolding - Definition - Folding, in this order for avoiding unnecessary variables in logic programs. In *Proceedings of PLILP '91, LNCS 528.* pp347-358, Passau, Germany, 1991.

[Plu90] L. Plümer. *Termination Proofs for Logic Programs.* Lecture Notes in Artificial Intelligence Vol 446, 1990.

[Ram30] F. P. Ramsey. On a Problem of Formal Logic. In *Proc. London Math. Soc.* (2) 20, pp264-286, 1930.

[Ree86] J. Rees, W. Clinger. Revised report^3 on the algorithmic Language Scheme. In *SIGPLAN Notices.* Vol 21, No 12, pp37-79, December, 1986.

[Rey72] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the 25'th ACM National Conference.* ACM 1972.

[Rom91] A. Romanenko. Inversion and Metacomputation. In *Symposium on Partial Evaluation and Sematics-based Program Manipulation.* Yale University, USA, pp12-22, 1991.

[Run89] C. Runciman, M. Firth, N. Jagger. Transformation in a Non-Strict Language: An approach to instantiation. In *1989 Glasgow Functional Programming Workshop.* 1989.

[Sei93] H. Seidl. *Approximating Functional Programs in Polynomial Time.* Unpublished manuscript. 1993.

[Ses88a] P. Sestoft. Automatic Call Unfolding in a Partial Evaluator. In *Partial Evaluation and Mixed Computation.* Eds. A. P. Ershov, D. Bjrner, N. D. Jones, North-Holland, 1988.

[Ses88b] P. Sestoft, A. V. Zamulin. Annotated Bibliography on Partial Evaluaiton and Mixed Computation. In *Partial Evaluation and Mixed Computation.* Eds. A. P. Ershov, D. Bjrner, N. D. Jones, North-Holland, 1988.

[She93] T. Sheard, L. Fegaras. A Fold for All Seasons. In *Conference on Functional Programming and Computer Architecture.* Copenhagen, Denmark, 1993.

[Smi91] D. A. Smith. Partial Evaluation of Pattern Matching in Constraint Logic Programming Languages. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* Ed. N. D. Jones, P. Hudak, pp.62-71, ACM Press 1991.

[Sor93a] M. H. Sørensen. *A New Means of Ensuring Termination of Deforestation.* Student Project 93-8-3, DIKU; Department of Computer Science, University of Copenhagen, 1993.

[Sor93b] M. H. Sørensen. A New Means of Ensuring Termination with an Application to Logic Programming. In *Workshop of the Global Compilation Workshop at ILPS '93.* Available as Penn State University Technical Report, 1993.

[Sor94a] M. H. Sørensen. A Grammar-Based Data-Flow Analysis to Stop Deforestation. Accepted for *Colloqium on Trees and Algebra in Programming (CAAP) '94.* To appear as Lecture Notes in Computer Science, 1994.

[Sor94b] M. H. Sørensen, Robert Glück, Neil D Jones. Towards Unifying Deforestation, Supercompilation, Partial Evaluation, and Generalized Partial Computation. Accepted for *European Symposium On Programming (ESOP) '94.* To appear as Lecture Notes in Computer Science, 1994.

[Tak91] A. Takano. Generalized Partial Computation for a Lazy Functional Language. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* Ed. N. D. Jones, P. Hudak, pp.1-11, ACM Press 1991.

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms.* SIAM,1983.

[Tur85] D. A. Turner. Miranda: A Non-strict Functional Language with Plymorphic Types. In *Conference on Functional Programming and Computer Architecture.* LNCS 201, 1985.

[Tur90] D. Turner. An Overview of Miranda. In *Research topics in Functional Programming.* Ed. D. Turner, Addison-Wesley, 1990.

[Tur72] V. F. Turchin. Equivalent Transformation of Recursive Functions Defined in the Language REFAL (in Russian). In *Trudy Vsesoyuzn. Simpos Teoria Yazykov i Metody Progr.* pages 31-42, Alushta-Kiev, 1972.

[Tur74] V. F. Turchin. Equivalent Transformation of REFAL programs (in Russian). In *Avtomaizirovannaya Sistema Upravleniye Stroitelstvom.* Trudy TsNIPIASS, GOSSTROY, pages 36-68, Moscow, 1974.

[Tur77a] V. F. Turchin. Basic Refal and Its Implementation on Computers. In *GOSSTROI SSSR, TsNIPIASS.* 1977.

[Tur77b] V. F. Turchin. *The Phenomenon of Science.* Columbia University Press, 1977.

[Tur79] V. F. Turchin. A Supercompiler System Based on the Lnaguage Refal. In *SIGPLAN Notices.* 14(2), 1979.

[Tur80a] V. F. Turchin. *The Language REFAL—The Theory of Compilation and Metasystem Analysis.* Courant Computer Science Report 20, 1980.

[Tur80b] V. F. Turchin. Semantic Definitions in Refal and Automatic Production of Compilers. In *Semantics-Directed Compiler Generation, Aarhis, Denmark.* (Ed.) Neil D. Jones, Lecture Notes in Computer Science vol. 94, pp. 645-657, 1980.

[Tur80c] V. F. Turchin. The Use of Metasystem Transition in Theorem Proving and Program Optimization. In *Automata, Languages and Programming, Seventh ICALP, Noordwijkerhout, The Netherlands.* (Ed.) J.W. de Bakker, J. van Leuwen, Lecture Notes in Computer Science vol. 85, pp645-657, 1980.

[Tur82] V. F. Turchin, R. M. Nirenberg, D. V. Turchin. Experiments with a Supercompiler. In *ACM Symposium on Lisp and Functional Programming, Pittsburgh, Pennsylvania.* pp 47-55, New york, 1982.

[Tur86a] V. F. Turchin. *Refal: A Language for Linguistic Cybernetics.* Technical Report, City University, New York, 1986.

[Tur86b] V. F. Turchin. The Concept of a Supercompiler. In *ACM Transactions on Programming Languages and Systems*. Vol. 8, No. 3, pp. 292- 325, 1986.

[Tur86c] V. F. Turchin. Program Transformation by Supercompilation. In *Programs as Data Objects, Copenhagen, Denmark*. (Eds.) H. Ganzinger, Neil D. Jones, Lecture Notes in Computer Science, vol. 217, pp 257-281, 1986.

[Tur87] V. F. Turchin. A Constructive Interpretation of the Full Set Theory. In *Journal of Symbolic Logic*. Volume 52, Number 1, March 1987.

[Tur88] V. F. Turchin. The Algorithm of Generalization in the Supercompiler. In *Partial Evaluation and Mixed Computation*. Eds. A. P. Ershov, D. Bjrner, N. D. Jones North-Holland, 1988.

[Tur93] V. F. Turchin. Program Transformation by Metasystem Transitions. In *Journal of Functional Programming*. 1993.

[Tro88] A.S. Troelstra, D. van Dalen. *Constructivism in Mathematics, an introduction, vol.1.* Studies In Logic And The Foundations Of Mathematics, vol 121. North-Holland, Amsterdam, 1988.

[Ull88] J. D. Ullman, A. Van Gelder. Effecient Tests for Top-Down Termination of Logical Rules. In *Journal of the ACM*. Vol 35, No 2, April 1988.

[Wad84] P. L. Wadler. Listlessness is Better than Lazyness: Lazy Evaluation and Garbage Collection at Compile-time. In *ACM Symposium on Lisp and Functional Programming*. Austin, Texas, 1984.

[Wad85] P. L. Wadler. Listlessness is Better than Lazyness II: Composing Listless Functions. In *Workshop on Programs as Data objects*. LNCS 217, Copenhagen, 1985.

[Wad87a] P. L. Wadler. Views: A Way for Pattern-Matching to Cohabit with Data Abstraction. In *14th Conference on Principles of Programming Languages*. 1987.

[Wad87b] P. L. Wadler. Efficient compilation of pattern-matching. In *The implementation of Functional Programming Languages*. Ed. S. L. Peyton Jones, Prentice-Hall, 1987.

[Wad88] P. L. Wadler. Deforestation: Transforming Programs to Eliminate Trees. In *European Symposium On programming (ESOP)*. Nancy, France, 1988.

[Wan93] M. Wand. Specifying the Correctness of Binding-time Analysis. In *Journal of Functional Programming*. Vol.3 part 3, July 1993.

[Wei91] D. Weise, R. Conybeare, E. Ruf, S. Seligman. Automatic Online Partial Evaluation. In *5th ACM Conference on Functional Programming Languages and Computer Architecture*. Cambridge, Massachusetts, 1991.