Program and Data Specialization
Principles, Applications, and Self-Application

Karoline Malmkjær
Datalogisk Institut, Københavns Universitet

August 1989

Contents

1 Introduction **3**

1.1 Outline 4

1.2 Terminology 4

1.3 Background 4

2 Mixed Computation **5**

2.1 Introduction 5

2.2 Definitions 5

2.3 Some Terminology 6

2.4 Binding Times and Binding Time Analysis 6

2.5 Mixed Computation as a General Computation Principle 7

3 Program Specialisation **8**

3.1 Introduction 8

3.2 Theory 8

3.3 Algorithms 10

3.3.1 Polyvariant specialisation 10

3.3.2 Unfolding 11

4 Data Specialisation **13**

4.1 Introduction 13

4.2 Definition 13

4.3 Algorithms for Data Specialisation 13

4.3.1 Some trivial algorithms 13

4.3.2 The relation to partial evaluation 14

4.4 Barzdins's and Bulyonkov's algorithm 14

4.4.1 Evaluation of the algorithm 19

4.5 Extending the algorithm of Barzdins and Bulyonkov 20

4.5.1 Binding time analysis 21

4.5.2 The algorithm for tail-recursive first-order recursive equations 21

4.5.3 An algorithm for first-order recursive equations 23

4.6 Evaluation of the Extensions 24

4.6.1 Conditionals 24

4.6.2 The Structure of the Residual Data 25

4.7 Adding unfolding 25

4.8 A depth-first algorithm 26

4.9 Towards a Formal Derivation of Data Specialisation Algorithms 27

5 Application to String Matching **29**
5.1 Introduction . . . 29
5.1.1 Notation . . . 29
5.2 Definitions . . . 30
5.2.1 The Knuth-Morris-Pratt matching algorithm for static pattern strings . . . 31
5.2.2 Definitions related to a static subject string . . . 32
5.3 Specialising the naive algorithm . . . 34
5.3.1 A naive, quadratic, pattern matching algorithm . . . 34
5.3.2 Analysis of the quadratic result . . . 35
5.3.3 Suggestions for improvement . . . 35
5.4 Obtaining the Knuth-Morris-Pratt Algorithm . . . 36
5.4.1 Rewriting the algorithm . . . 37
5.5 Obtaining Suffix Trees and Automata . . . 39
5.5.1 Rewriting the algorithm . . . 39
5.5.2 Results of specialisation . . . 41
5.5.3 How can we get small, linear residual results? . . . 41
5.6 How to "Specialise Well" . . . 46
5.7 Extending the Specialiser . . . 46
5.8 Conclusion . . . 46

6 Sequential Decomposition **50**
6.1 Introduction . . . 50
6.2 Definitions . . . 50
6.3 Sequential Decomposition by Data Specialisation . . . 51
6.4 Sequential Decomposition for Multi-Phase Compiling . . . 52

7 Applications of Sequential Decomposition **54**
7.1 Introduction . . . 54
7.2 Partially Evaluating the Data Specialisation Algorithm . . . 54
7.2.1 Problems . . . 54
7.3 Sequential Decomposition of a String Matching Algorithm . . . 55
7.4 Program Specialising the Data Specialisation Program . . . 56
7.5 Sequential Decomposition for Compiler Generation . . . 56
7.6 Towards a Compiler-Based Design of Residual Data Structures . . . 56

8 Related Work **59**
8.1 Hand Transformations from Interpreter to Compiler . . . 59

9 Further Work **61**

10 Conclusion **62**

A Implementation of the Imperative Algorithm **66**
A.1 Main parts of the graph generating program . . . 66
A.2 The program generating program . . . 68

B Semantics of the First-Order Recursive Equations **70**

C Implementation of the Algorithm for First-order Recursive Equations 72

C.1 The graph generator 72

C.1.1 Data structures 72

C.1.2 The initialisation 72

C.1.3 The main loop 73

C.1.4 Treatment of expressions 73

C.1.5 Treatment of procedure application 74

C.2 The Program Generator 75

D Specialising String Matching Algorithms 77

Preface

This report is a master's thesis ("speciale") written by Karoline Malmkjær as the final part of the cand. scient education at the Computer Science Department of the University of Copenhagen (Datalogisk Institut ved Københavns Universitet).

Chapter 1

Introduction

Many problems are expressible as simpler cases of more general problems. When solving problems by computer, this may help us to find solutions. If the more general problem has a solution, we may be able to find a solution to the simpler case as a special case of the solution to a more general problem.

A particularly clear example of this is *specialisation*:

Many computer programs take several arguments. It often happens that some of the arguments are known at an earlier time than others. It may also happen that the program will be executed several times with the same values for some of the arguments.

It is possible to take advantage of these situations by constructing (by hand or machine) special programs integrating the constant data or by preprocessing the constant/earlier known data to speed up the subsequent computation.

The most well-known example of this is the use of compilers to obtain faster run times than if the source program had to be interpreted. Conceptually, this can be understood as specialising an interpreter with respect to the source program. Even in other cases the term "compilation" is often used to describe the integration of a program with some of its data.

Other examples include

- String matching, where we want to look for for a pattern string in several subject strings, *e.g.*, to find a sentence in one of several text files, or conversely look for several patterns in a constant string (in the most general case, this includes searching in a data base).
- Image generation, *e.g.*, ray tracing, where we must generate pictures of the same scene from different viewpoints.
- Parsing, where we want to match several strings against the same grammer.

This report investigates principles describing preprocessing and compilation as specialisation of more general algorithms.

The most general formulation is that of *mixed computation* [Ershov 77]. Special cases of this include *program specialisation*, also known as *partial evaluation* [Lombardi & Raphael 64] and the new principle of *data specialisation* [Barzdins & Bulyonkov 88].

In this thesis an algorithm for data specialisation is implemented and applied to the string matching problem. The principles of using data specialisation for compiler generation are outlined and the adequacy of the presented algorithm for this application is discussed.

1.1 Outline

We will introduce self-applicable partial evaluation and some of the related problems, *e.g.*, concerning binding time analysis and unfoldability annotations.

Thus familiarity with self-applicable partial evaluators is not a prerequisite to understand the report, but many of the problems encountered here will be very similar to well-known problems in program specialisation. For this reason we will often refer to existing literature rather than describe these problems in detail.

Chapters 2 and 3 introduce mixed computation and program specialisation, providing the background for the rest of the report.

Chapter 4 presents the principle of data specialisation, the main topic of the report. Some algorithms are given and the possibility of deriving an algorithm from a semantic specification is discussed.

Chapter 5 shows and compares application of program specialisation and data specialisation to the generation of string matching algorithms.

In chapter 6 the principle of sequential decomposition is introduced and it it seen how a sequential decomposition algorithm can be obtained using program specialisation of a data specialiser and how sequential decomposition can be used to generate a multi-pass compiler.

Chapter 7 describes the sequential decomposition algorithm based on a data specialisation algorithm from chapter 4 and illustrates it with one of the string matching algorithms from chapter 5. Then the design of data specialisation algorithms for the multi-pass compiler application is discussed.

Some related works, particularly in compiler generation, are presented in chapter 8. Chapter 9 suggests some further directions of work and chapter 10 is the conclusion.

1.2 Terminology

For concepts related to programs and programming languages we will mainly use the terminology employed in the Mix papers [Jones et al. 89].

We will use the term "function" only for the abstract, mathematical object and the term "procedure" for textual descriptions of algorithms as they appear in programs. Thus "the function add1" refers to the function computed by the procedure `add1` and is identical to the function $f(x) = x + 1$.

1.3 Background

Partial evaluation was first mentioned in [Lombardi & Raphael 64].

The use of self-applicable partial evaluation for compiler generation was described in [Futamura 71]. The first reported self-applicable partial evaluator was the Mix system [Jones et al. 1985] [Jones et al. 89].

The idea of data specialisation and its use to generate sequential composition and multi-phase compilers comes from [Barzdins & Bulyonkov 88], which also gives an algorithm for data specialisation (described in chapter 4).

The extension of this algorithm and the implementations were done for this report.

[Consel & Danvy 89] uses program specialisation to obtain compiled matching algorithms like in [Knuth, Morris & Pratt 77]. With Olivier Danvy, the author has generalised the principles involved to obtain matching algorithms comparable to [Weiner 73]. For this report, these results have been extended to obtain directed acyclic words graphs [Blumer et al. 85]. The data specialiser developed has been used to generate the actual data structures and corresponding matchers, rather than compiled versions, for all cases.

Chapter 2

Mixed Computation

2.1 Introduction

The concept of mixed computation was introduced in [Ershov 77] and [Ershov 78] as a framework to generalise underlying principles found in compiler construction. It was soon related to partial evaluation [Lombardi & Raphael 64] which is a special case of mixed computation and with the work of Futamura [Futamura 71] and Turchin [Turchin 74] [Turchin 81].

This section gives the definition of mixed computation and points out the special cases that have inspired most of the work done in the area. Particularly the concepts of input division and binding time are introduced.

2.2 Definitions

We will use the following framework of programs and programming languages (from [Jones et al. 89]).

We assume that we have a domain D containing the values that we need, *e.g.*, natural numbers and character strings (including program texts). This domain is further assumed to contain its n-ary products for all $n \in I\!N$:

$$[\,,\ldots,\,] : D^n \to D$$

A programming language L is then defined as a partial function:

$$L : D \overset{\sim}{\to} D \overset{\sim}{\to} D$$

Thus a legal program l in the language L is mapped to a partial input-output function:

$$L\,l : D \overset{\sim}{\to} D$$

(here $\overset{\sim}{\to}$ is used to denote partial functions).

Definition 1 *([Ershov 77] and [Ershov 78]).* **Mixed computation** *for a programming language L is a two-part algorithm MIX = (MIX$_p$, MIX$_d$) written in a (possibly different) language M. For any program l in L and input d, MIX generates an intermediate program* $l' = M\,MIX_p\,(l, d)$ *and intermediate output* $d' = M\,MIX_d\,(l, d)$*. The algorithm MIX is correct if*

$$L\,l\,d = L\,l'\,d'$$

□

Thus correct mixed computation consists of any two algorithms that take a program and its input and generate a new program and some new input so that, when the new program is run on the new input, it gives the same result as the original program on the original input. It is simple to prove that there exists a mixed computation algorithm:

Let M be the λ-calculus. Let $MIX_{trivial} = (\lambda l.\, l, \lambda d.\, d)$.

Then $MIX_{trivial}$ is a correct mixed computation.

[Ershov 78] gives an example of a non-trivial mixed computation for a small imperative language. The algorithm proceeds by executing some of the statements and terms of l and "suspending" the rest. The decision to suspend one part of the computation then forcibly suspends those other parts that depends on the result of the suspended part.

As an important special case [Ershov 78] mentions the case where the input can be divided in two parts, $d = (d_s, d_d)$. If all statements and terms depending on d_d are suspended and all occurrences of d_s are integrated into l', then MIX_d can be defined simply by $M\; MIX_d\, [l, [d_s, d_d]] = d_d$. MIX_p, on the other hand, does not depend on d_d. This case has later been known as *partial evaluation* or *program specialisation* (see chapter 3), since such a MIX_p-algorithm partly evaluates the program l, based on the partial input d_s, and returns a specialised version of l that only depends on d_d.

[Barzdins & Bulyonkov 88] isolates another special case which still involves a concept of input division but where MIX_d is less trivial:

Definition 2 Advanced partial evaluation *is a pair of algorithms* $[APE_d, APE_p]$*, with*

$$\begin{array}{lcl} APE_p[l, [d_s, d_d]] & = & l' \\ APE_d[l, [d_s, d_d]] & = & [d_s', d_d] \\ \textit{so that} & & \\ L\, l\, [d_s, d_d] & = & L\, l'\, [d_s', d_d] \end{array}$$

and both l' *and* d_s' *are independent of* d_d □

2.3 Some Terminology

The input program l will be called the *source program*. The output program l' will be called the *residual program*.

If we divide the input into two parts $d = [d_s, d_d]$ of which d_s will be passed to MIX_p, but d_d will not, then d_s is called the *static* input and d_d is called the *dynamic* input.

Correspondingly all syntactic constructions in l whose evaluation depend on d_d are called dynamic, and those that do not are called static.

2.4 Binding Times and Binding Time Analysis

Consider the case where *MIX* is independent of some part of the input (the dynamic data), that is, it returns the dynamic data unchanged and the actions on the source program and static data do not depend on the dynamic data. If we want a non-trivial mixed computation for this case, we need to find out which parts of the computation specified by the source program are only dependent on the static data, so that *MIX* can perform them, and which parts depend on the dynamic data, so that *MIX* has to suspend them, that is, move them into the residual program or the residual data.

To express this, let us consider a simple imperative language. Then we can see the computation as a transformation on states consisting of pairs of a program point and a store. Then we want, for each state in the computation of the program, to find out whether the statement at that point in the program can be executed at transformation time, by *MIX*, or whether it depends directly or indirectly on the dynamic data, so that the state transformation must be postponed to run time. In the second case, the state must be transformed to residual output, either in the residual program or the residual data.

Such a division of states is not in general computable to find in a way that guarantees the mixed computation to terminate, since this amounts to solving the halting problem if the set of dynamic input is empty.

It is, however, computable to find a non-trivial, "safe" approximation, that is, a division of the states that causes the mixed computation to terminate at least as often as the source program for the same static data with all combinations of dynamic data. This can be done either "on the fly" or by pre-processing, for example by dividing the set of variables in the program into two sets, the set of static variables and the set of dynamic variables. A variable is static if it depends only on static variables — otherwise it is dynamic. The treatment of a state then depends on whether the statement refers to any dynamic variables.

This analysis is called *binding time analysis* [Jones et al. 89], since it determines the *binding time*s of the variables, *i.e.* whether the variable will be bound to a value at the time of transformation or when the residual program is run.

[Jones 88] analyses the concept of binding time analysis as a division of states and gives several algorithms detecting binding time information.

Thus the specialisation algorithms presented in this report will assume that their input have already been binding time analysed.

For a more detailed exposition, see also [Mogensen 89].

2.5 Mixed Computation as a General Computation Principle

Mixed computation was developed as a general principle that would cover all kinds of computation, particularly transformations of programs (since programs have to be compiled or interpreted in order to run this could be seen as all computations as well). Although it is now more than ten years old, this approach has only slowly gained recognition. The main applications have been in compiling, where it was originally developed. One problem might be that it is in fact too general.

One sign of this is that the compiler application is a special case, and many of the crucial concepts, such as the distinction of compile-time = transformation-time against run-time, and the notion of binding time of variables, concern the conditions separating this special case from the general case, rather than the properties of the general case.

In the following we will also only consider cases with a division of the input of the source program into static and dynamic, although in a more general way.

Chapter 3

Program Specialisation

3.1 Introduction

This chapter describes the principle of *program specialisation*, *i.e.*, the generation of a specialised (and preferably faster) target program from a source program and the values of some of the input of this program.

The first section describes the theoretical background in recursive function theory and the application of program specialisation to compiler generation.

The second section outlines some algorithms for program specialisation and some of the techniques used to get non-trivial results of self-application.

3.2 Theory

The goal of program specialisation is to take advantage of the case where we have a program and we know some of its input, but not the rest, or some of the input will be constant for several values of the rest of the input. The idea is to generate a *specialised* version of the program with respect to the known or constant input in such a case.

This concept is well-known from mathematics. If we have a function $\varphi(x, y)$ then φ_x denotes the x-indexed family of functions of one argument, y, so that $\varphi_x(y) = \varphi(x, y)$.

Kleene's S^m_n theorem shows that it is in fact possible, if we have an algorithm for φ, to find an algorithm for φ_x:

Theorem 1 *For all m, n there exists a function, S^m_n, of $m + 1$ arguments, so that for all terms $f(x_1, \ldots, x_{m+n})$:*

$$\varphi_f(x_1, \ldots, x_{m+n}) = \varphi_{S^m_n(f, x_1, \ldots, x_m)}(x_{m+1}, \ldots, x_{m+n})$$

Here f denotes a term and φ_f denotes the function computed by the term f.

The proof is constructive, so the theorem even gives us a program specialisation algorithm.

Program specialisation can also be seen as a special case of mixed computation with MIX_d being the function returning the dynamic data and MIX_p being the partial evaluator.

In [Jones et al. 89] interpreters and compilers are defined extensionally using the framework of programming languages as partial functions.

Definition 3 *An* **interpreter** *for a programming language L is a program int written in a programming language N, so that:*

$$N\,\mathit{int}\,[l,d] = L\;l\;d$$

□

Definition 4 *A* **compiler** *from a programming language L to a programming language M is a program comp written in a programming language N, so that:*

$$M\;(N\,\mathit{comp}\;l)\;d = L\;l\;d$$

□

In this framework we can also define a program specialising algorithm (usually called a partial evaluator) computing the S^m_n-function:

Definition 5 *A* **partial evaluator** *for a binding time pattern m, n, from a programming language L to a programming language M is a program PE written in a programming language N, so that:*

$$M\;(N\,\mathit{PE}\;[l, d_1, \ldots, d_m])[d_{m+1}, \ldots, d_{m+n}] = L\;l\;[d_1, \ldots, d_{m+n}]$$

□

If $L = N$, the partial evaluator is called *self-applicable* because it treats the same language it is written in, so it can be applied to its own text.

In [Futamura 71] it was shown that a self-applicable partial evaluator can be used to compile programs and even to generate compilers. The starting point is that if we have an L-program l, then an M-program that is generated from l and computes the same function as l, can be considered to be a compiled version of l. Now if we have a partial evaluator *PE* from N to M written in K and an L-interpreter written in N, then

$$\mathit{target} = K\,\mathit{PE}\;[\mathit{int}, l] \tag{3.1}$$

is a compiled version of l written in M, since by definition 5:

$$M\;\mathit{target}\;d = L\;l\;d$$

Next, we observe that any program that takes a source program written in L and generates an M program that computes the same function as the source program can be considered to be a compiler from L to M. Thus, if we have a self-applicable partial evaluator *PE* from N to M, written in N and an L-interpreter *int* written in N, then

$$\mathit{comp} = N\,\mathit{PE}\;[\mathit{PE}, \mathit{int}] \tag{3.2}$$

is (as the name suggests) a compiler from L to M, since

$$\begin{array}{lcl}
M\;(M\;(N\,\mathit{PE}\;[\mathit{PE}, \mathit{int}])\;l)\;d & = & \text{(by definition 5)} \\
M\;(N\,\mathit{PE}\;[\mathit{int}, l])\;d & = & \text{(do.)} \\
N\,\mathit{int}\;[l, d] & = & \text{(by definition 3)} \\
L\;l\;d & &
\end{array}$$

thus *comp* fulfills definition 4.

In [Turchin 74] it was then shown how to extend this to a compiler generator by further self-application:

$$cogen = N\, PE\, [PE, PE] \tag{3.3}$$

is a compiler generator, since $M\, cogen$ applied to an interpreter is a compiler:

$$M\, cogen\, int = N\, PE\, [PE, int] = comp$$

In fact *cogen* is more than a compiler generator, since it is possible to apply it to any kind of N-programs, not only interpreters.

Equations 3.1, 3.2 and 3.3 are the so-called *mix equations* or *Futamura projections*.

3.3 Algorithms

As for mixed computation, it is easy to see that there exist program specialisation algorithms. If we use a λ-notation for functions, let $PE_{trivial}$ be $\lambda\, x\, s.(\lambda\, d.\,(L\, x\, [s, d]))$. It is clear that $PE_{trivial}$ is a partial evaluator.

3.3.1 Polyvariant specialisation

Most program specialisers use the technique of polyvariant specialisation [Bulyonkov 84][1].

This technique aims to perform those parts of the computation that only depend on the static data. In order to do this, it is assumed that the dynamic variables have all possible values. Thus any condition that depends on dynamic data will be both true and false, *e.g.*, if the condition tests whether a dynamic expression is greater than five, this will be true, because the expression might evaluate to any number larger than five, but it will also be false, because the expression might *also* evaluate to any number smaller than five, or to five. This means that specialisation should continue on both branches of a conditional, in order to evaluate the static parts occurring there.

This way it is possible to execute those parts that depend only on static data, but how do we incorporate the results of this into the residual program, so that we do not have to repeat it at run-time?

An expression depending on static values will typically have many values at different times of the evaluation, so we cannot just replace it by its value. Similarly a static condition will be both true and false during a computation (*e.g.* if we descend recursively on a static list, the test for empty list will first be false and then true). So it is not possible to replace a conditional with the part that should be executed, since both branches have to be executed at some time.

The solution of polyvariant specialisation is to generate one instance of each program point for each set of values of the static variables. Then a static expression or condition is not only static, but constant, and can be evaluated and replaced by its value.

So the idea of polyvariant specialisation is to generate one program point in the residual program for each pair of program point and static values occurring during the symbolic evaluation of the source program on the partial data.

Let us for example consider the language Scheme [Rees & Clinger 86], with procedure definitions as program points.

As the example, we specialise the program `append` (given in figure 3.1) with respect to the list `(a b c)` as first argument and a dynamic second argument.

[1] The main exception is the Refal system [Turchin 74] [Turchin 81]

```
(define (append l1 l2)
    (if (null? l1)
        l2
        (cons (car l1) (append (cdr l1) l2))))
```

Figure 3.1: The source program `append`

During the symbolic evaluation we first call `append` with `(a b c)` and a dynamic `l2`. Since we want to generate a residual procedure definition for each set of static values the source procedure is called with, this means that we want to generate a new procedure corresponding to (`append`, `(a b c)`), let us call it `append-abc`. The body of this new procedure is the body of the source procedure, specialised to remove all occurrences of `l1`.

First we specialise the conditional. Since the test depends only on the static argument, we can determine that it is false and replace the conditional expression by its false-branch. The false-branch (a cons-expression) depends on the dynamic argument, and thus we have to leave it in the residual program. The first argument, however, depends only on `l1`, so we replace it by a representation of its value, `'a`.

The second argument to the cons-expression is a call to `append`. This call has the dynamic argument `l2` and its first argument is the static expression `(cdr l1)`. This can be evaluated to `(b c)`. Now we have a call to `append` with the first argument `(b c)` and the second argument `l2`, so we generate a new procedure corresponding to (`append`, `(b c)`), let us call it `append-bc`.

This is done as for the first residual procedure and leads to the generation of a third procedure, `append-c`, which again leads to `append-nil`. In `append-nil` the condition is true, so we specialise the true-branch, giving the expression `l2` as the body of this procedure. Our residual program then looks like this:

```
(define (append-abc l2)
    (cons 'a (append-bc l2)))

(define (append-bc l2)
    (cons 'b (append-c l2)))

(define (append-c l2)
    (cons 'c (append-nil l2)))

(define (append-nil l2)
    l2)
```

Examples of program specialisers using polyvariant specialisation are, *e.g.*, [Jones et al. 89], [Consel 89], [Bondorf & Danvy 89].

3.3.2 Unfolding

Polyvariant specialisation often gives trivial residual procedures containing only calls to other procedures, as seen in the example above. To a large extent this can be improved using *call unfolding*.

With this technique, the specialised body of a procedure is inserted in place of the call to the procedure.

In the append-example, the call to `append-nil` in `append-c` could simply be replaced by the body of the procedure, giving:

```
(define (append-c l2)
    (cons 'c l2))
```

Similarly the calls to `append-c` and `append-bc` could be unfolded, giving as the final result the much smaller residual program:

```
(define (append-abc l2)
    (cons 'a (cons 'b (cons 'c l2))))
```

Call unfolding can be performed either during specialisation (possibly based on annotations from a pre-processing phase) or by post-unfolding the residual program [Sestoft 88].

Although call unfolding will often reduce the size of the residual program, it may also increase both the size and the complexity because of *call duplication*. This is clearly seen in the classical example:

```
(define (foo x y)
    (bar (* x y)))
(define (bar z)
    (+ z z))
```

If the call to bar is unfolded here, we get:

```
(define (foo x y)
    (+ (* x y) (* x y)))
```

i.e., the multiplication of `x` and `y` will now be performed twice.

This problem can be avoided by an analysis of occurrences of parameters before unfolding, see, *e.g*, [Bondorf & Danvy 89].

Another problem is that of infinite unfolding, a problem related to the potential infinity inherent in recursive definitions. The problem arises when trying to unfold a recursive (or mutual recursive) call where the recursion is controlled by a dynamic test. Since the test cannot be computed at partial evaluation time, the partial evaluator will specialise both branches of the conditional, leading to an infinite unfolding of the recursive calls.

This can be avoided, *e.g.*, by not unfolding procedures containing dynamic conditionals [Bondorf & Danvy 89].

Chapter 4

Data Specialisation

4.1 Introduction

This chapter introduces the concept of *Data Specialisation* as defined in [Barzdins & Bulyonkov 88].

It is shown how partial evaluation can be seen as a special case of data specialisation and how data specialisation is again a special case of mixed computation.

Then some algorithms for data specialisation are discussed, both the algorithm from [Barzdins & Bulyonkov 88] (for a small imperative language) and some algorithms developed for this thesis (for a system of first-order recursive equations) as well as a series of special case algorithms, some trivial ones and polyvariant partial evaluation.

Finally a short section is dedicated to the consequences of changing the structure of the residual data.

4.2 Definition

Data specialisation can be seen as a special case of mixed computation where the input data is still divided into static and dynamic data, but now the program generator MIX_p is independent of the values of the static data, *i.e.*:

Definition 6 *A* data specialiser *for a programming language L is a pair of programs $[DS_d, DS_p]$ written in M, so that for all L-programs l with arity $n+m$ the following equation holds:*

$$L\, l\, [d_1, \ldots, d_{n+m}] = N\, (M\, DS_p\, l)\, [(M\, DS_d\, [l, d_1, \ldots, d_n]), d_{n+1}, \ldots d_{n+m}] \tag{4.1}$$

□

This is in fact also a special case of advanced partial evaluation as defined in [Barzdins & Bulyonkov 88], where $l' = M\, DS_p\, l$ is independent of d_s.

In the rest of the report "data generating algorithm" will be used for an algorithm computing DS_d and "program generating algorithm" for an algorithm computing DS_p.

4.3 Algorithms for Data Specialisation

4.3.1 Some trivial algorithms

As with program specialisation, it is possible to find an easy, but trivial, algorithm performing data specialisation. We just define

$$
\begin{array}{lcl}
M\,\mathit{DS}_d\,[l,d_1,\ldots,d_n] & = & [d_1,\ldots,d_n] \\
M\,\mathit{DS}_p\,l & = & \lambda\,[[d_1,\ldots,d_n],d_{n+1},\ldots,d_{n+m}].\,L\,l\,[d_1,\ldots,d_{n+m}]
\end{array}
$$

(using λ-notation for the program).

The trivial mixed computation algorithm, $\mathit{MIX}_{trivial}$ from section 2.2 is also a data specialisation algorithm.

4.3.2 The relation to partial evaluation

[Barzdins & Bulyonkov 88] observes that partial evaluation can be seen as a special case of data specialisation.

Let us assume that we have a partial evaluator *PE* generating programs written in a language K and an K-interpreter, *int* written in N. Then we can define:

$$
\begin{array}{lcl}
\mathit{DS}_d & = & \mathit{PE} \\
\mathit{DS}_p & = & \lambda\,l.\,\mathit{int}
\end{array}
$$

This clearly fulfills definition 6, since

$$
\begin{array}{lcl}
N\,(M\,\mathit{DS}_p\,l)\,[(M\,\mathit{DS}_d\,[l,d_1,\ldots,d_n]),d_{n+1},\ldots d_{n+m}] & = & \text{(as defined above)} \\
N\,\mathit{int}\,[(M\,\mathit{PE}\,[l,d_1,\ldots,d_n]),d_{n+1},\ldots d_{n+m}] & = & \text{(by definition 3)} \\
N\,(M\,\mathit{PE}\,[l,d_1,\ldots,d_n])\;d_{n+1},\ldots d_{n+m} & = & \text{(by definition 5)} \\
L\,l\,[d_1,\ldots,d_{n+m}] & &
\end{array}
$$

At first it might sound surprising that program specialisation is a special case of data specialisation rather than a dual case. But it seems reasonable, since programs are in fact not the dual of data but rather a special kind of data objects.

It is worth to notice that this is also a second way to see partial evaluation as a special case of mixed computation.

4.4 Barzdins's and Bulyonkov's algorithm

[Barzdins & Bulyonkov 88] presents an algorithm for data specialisation that is related to the polyvariant partial evaluation algorithms and proves its correctness, *i.e.*, that it fulfills definition 6.

The algorithm treats programs in a small imperative language with labeled statements, assignment, `goto`, and conditional `goto`. The programs are assumed to be *analyser programs* [Bulyonkov 84]. In relation to specialisation this means that there is only one static variable and that the computation is controlled by this variable.

The language has the following syntax

$$
\begin{array}{rcl}
\mathit{Program} & ::= & \texttt{((}\mathit{Definition}^*\texttt{)}\,\texttt{(}\mathit{Instruction}\texttt{))} \\
\mathit{Definition} & ::= & \texttt{(}\mathit{Identifier\ Expression}\texttt{)} \\
 & | & \texttt{(}\mathit{Identifier}\texttt{)} \\
\mathit{Instruction} & ::= & \texttt{(}\mathit{Label\ Command}\texttt{)} \\
\mathit{Label} & ::= & \mathit{Identifier} \\
\mathit{Command} & ::= & \mathit{Left\text{-}side}\texttt{:=}\mathit{Expression}
\end{array}
$$

```
                  |  goto Label
                  |  if Expression goto Label
                  |  read Left-side
                  |  print Expression
                  |  end
Expression  ::=  #t | #f | Numeral | String |
                  |  Identifier | Identifier [Expression]
                  |  (Mon-op Expression)
                  |  (Bi-op Expression Expression)
```

and is syntactically sugared with structured loops and `if-then-else` constructions.

The residual data generated by the data generating algorithm is a labeled graph. Each node in this graph represents a static store (*i.e.*, the value of the static variable). The value of the static variable is not saved in the node, however. Instead each node has an attribute for each static expression whose value is computed in the corresponding store. The attribute contains the value of the expression in that store. Each edge represents a static transition, *i.e.*, a change in the static store caused by a static assignment. The edges are labeled with the label of the command that contains the assignment.

The residual program resembles the source program. Its arguments are the dynamic arguments of the source program and two new arguments: the residual graph and a pointer to the initial node in the graph. Instead of static assignments, the residual program contains updatings of the pointer to the current node. Instead of static expressions it contains references to attributes of the current node.

The residual graph is generated by a symbolic evaluation of the source program based on the partly defined initial store. The generator performs a breadth-first traversal of the part of the computation tree that is reachable on the static data using a list of "active" stores and program points to save reachable, yet untreated, paths. If the computation was not limited by the static data in all branches, the generation might not terminate since there could be infinitely many different static stores.

More concretely the data generating algorithm takes a program and a value and iterates over a list of active states, starting with the first label in the program paired with the static input value. For each state it either generates a new node or finds the corresponding node in the graph using a table `node_of`.

If the statement denoted by the label is an assignment to the static variable the algorithm performs the assignment, looks up the node corresponding to the resulting state (or generates a new node if it has not been met before), and generates an edge to that node from the current one. The new state is then put in the list of active states.

If the statement is anything else, the generator computes the values of all maximal static expressions in the statement and adds an attribute to the current node in the graph for each static expression. The label of the following statement (or statements, if the current statement is a dynamic test) is then paired with the static value to give the new state, that is added to the list of active states.

The program generator is more simple. It runs through the program once, modifying the commands. All static expressions that are not subexpressions of larger static expressions are replaced by references to attributes in the current node in the graph. All static assignments are replaced by updatings of the pointer to the current node in the graph along the edge with the label of the assignment.

The graph generator and the program generator are shown in fig. 4.1.

Example

As an example, consider the small program given in fig. 4.2 multiplying two natural numbers. We can specialise this with a static `n`.

The data generating algorithm takes the program, some kind of binding time information telling it which variable is the static one, and an input stream containing the static value, let us say three.

First it creates the initial state from the first label, `read0:`, and the first value of `n`, undefined and generates an empty node. Then it initialises the list of active states with the state, the graph to the node and the correspondence `node_of` to match the initial state with the initial node and starts its main loop. The graph now looks like this:

$$\bigcirc$$

In this loop it takes a state from the list of active states, here [`read0:`, [undefined]]. Then it adds it to the list of passed states, finds the corresponding node using `node_of` and treats the state. Since the statement with the label `read0:` is static, it is executed, giving the new store [three]. This store does not have a node, so the algorithm generates a node for the state and adds an entry to `node_of`. Then it generates an edge from the current node to the new node. Now the graph looks like this

$$\bigcirc \xrightarrow{\text{read0:}} \bigcirc$$

Then the algorithm adds [`read1:`, three] to the list of active states and iterates.

This time it takes the state [`read1:`, three] from the list of active states, adds it to "passed" and finds its node (the one we just put in the graph). The statement corresponding to `read1:` is dynamic, so it is not executed. It does not contain any static expressions, so nothing is added to the current node. It passes control to the next statement in the program, so the state [`L1:`, three] is added to "active".

The statement corresponding to `L1:` is also dynamic and does not contain any static expressions, so it is treated the same way.

The next state is [`loop:`, three]. The corresponding statement is a static conditional, which is executed and found to be false. Then the attribute "`(= n 1)` is false" is added to the node corresponding to [three], giving the following graph:

$$\bigcirc \xrightarrow{\text{read0:}} \texttt{(= n 1)} \text{ is false}$$

Since the conditional was false, the label of next statement in the program is paired with the store to give the next state.

This state is a static assignment, so it is executed, giving the new store [two], so a new node is generated, giving the graph:

$$\bigcirc \xrightarrow{\text{read0:}} \texttt{(= n 1)} \text{ is false} \xrightarrow{\text{L2:}} \bigcirc$$

The next state is [`L3:`, [two]]. The corresponding statement is a dynamic assignment with no static expressions, so nothing is added to the graph.

The next statement is a goto, so the algorithm will use the given address and add [`loop:`, [two]] to the active states.

```
Graph generating algorithm:
arguments: source program, initial state $s_0$
   G := <{$v_0$},$v_0$,0>
   Active := {<$l_0$,$s_0$>}; Passed := 0
   set node_of($s_0$) = $v_0$
   while Active is not empty do
       take <l,s> from Active
       if <l,s> doesn't belong to Passed then
          add {<l,s>} to Passed
          v := node_of(s)
          if instruction labeled by l is static assignment then
             s' := eval((source of assignment in l),s)
             if node_of(s') is not defined then
                v' := new node
                add node v' to G
                set node_of(s') = v'
             else
                v' := node_of(s')
             fi
             add edge labeled l from v to v'
          else
          add attributes for the statement of l to v
          s' := s
          fi
       add { <l',s'> | l' is a possible successor of l on s} to Active
       fi
   od
result G
Program generating algorithm:
arguments: source program P indexed by PC
   P'[new-label-1] := "v:= $v_0$"
   PC :=  first label
   repeat
       if P[PC] is static assignment then
          P'[PC] := "v := v->PC"
       else Make new command where all static expressions are
            replaced by references to node v in the graph.
            Assign the new command to P[PC]
       fi
   until PC is the last label
```

Figure 4.1: The algorithm from [Barzdins & Bulyonkov 88]

```
(((n) (m) (res))
 (read0:   read n)
 (read1:   read m)
 (L1:      res := m)
 (loop:    if (= n 1) goto success:)
 (L2:      n := (sub1 n))
 (L3:      res := (+ m res))
 (L4:      goto loop:)
 (success: print res)
 (stop:    end))
```

Figure 4.2: Small imperative program computing the product of two natural numbers

This way it continues until the state [`loop:`, [one]] is treated. Then the condition is true, so the next state in active is [`success:`, [one]]. This corresponds to a dynamic statement with no static expressions, and the next state is [`stop:`, [one]] that does not add any states to "active".

At the next loop "active" is thus empty, and the algorithm returns the graph, which looks like this:

$$\bigcirc \xrightarrow{\text{read0:}} \texttt{(= n 1)} \text{ is false} \xrightarrow{\text{L2:}} \texttt{(= n 1)} \text{ is false} \xrightarrow{\text{L2:}} \texttt{(= n 1)} \text{ is true}$$

The program generating algorithm adds two read commands reading the graph and the initial node, replaces the updating of `n` (command `read0` and `L2:`) with updatings of the graph and references to `n` with references to the graph. This gives the following residual program:

```
(((g) (node) (m) (res))
 (g29 read g)
 (g30 read node)
 (read0: node := (graph-ref g node read0:))
 (read1: read m)
 (l1: res := m)
 (loop: if (graph-ref g node (= n 1)) goto success:)
 (l2: node := (graph-ref g node l2:))
 (l3: res := (+ m res))
 (l4: goto loop:)
 (success: print res)
 (stop: end))
```

`graph-ref` is a primitive accessing edges and attributes in graphs like the residual data.

This is of course a particularly simple example, since there are no dynamic conditionals at all, and thus no nodes with more than one child in the residual graph. The treatment of a dynamic conditional is the following: first the possible static expressions are computed and added as attributes to the node of the current state. Then the states corresponding to the next statement in the program *and* the statement corresponding to the jump are both added to the list of active states and the algorithm iterates.

[Barzdins et al. 89] presents a less trivial example; an interpreter of finite automata taking an automaton and a string over the alphabet {a, b} and returning accept or fail depending on whether the automaton recognises the string. The program is given in fig. 4.3. The automaton is represented

```
(((R) (c) (next-a) (next-b) (accept))
 (R1:    R := 0)
 (read1: read c)
 (l3:    if (equal? c 'end) goto l9:)
 (l4:    if (equal? c 'b) goto Rb:)
 (Ra:    R := next-a[R])
 (l5:    goto l7:)
 (Rb:    R := next-b[R])
 (l7:    read c)
 (l8:    goto l3:)
 (l9:    print accept[R])
 (stop:  end))
```

Figure 4.3: Small imperative program interpreting automata

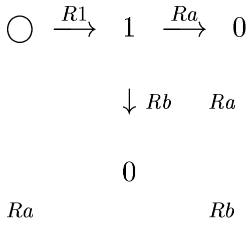


Figure 4.4: The residual graph when data specialising the program in fig. 4.3.

by three arrays, one for transitions on 'a', one for transitions on 'b' and one indicating accept or fail (0 or 1).

When this program is data specialised, the structure of the automaton is revealed in the residual graph. As an example, the result of data specialising with

```
accept  =  [1, 0, 0]
next-a  =  [1, 2, 2]
next-b  =  [2, 0, 2]
```

is shown in fig. 4.4.

This gives exactly the automaton, with accept or fail as an attribute in each node.

The residual program (shown in fig. 4.5) walks along the residual automaton. It is worth to notice, though, that it is not more efficient than the original program, it has the same complexity.

4.4.1 Evaluation of the algorithm

The algorithm is inspired by polyvariant specialisation but differs in the fact that a node in the residual graph does not correspond to a pair of program point and static store but only to a static store. This makes a difference, *e.g*, if we have a program with four labels, `l1`, `l2`, `l3`, `l4`, and one static variable `a` where execution of statement `l1` and `l3` on two different stores both cause `a` to become the same value, *e.g.*, one. Then the edges in the residual graph labeled `l1` and `l3` will both lead to the same node, even if `l1` transfers control to `l2` and `l3` transfers control to `l4`. In polyvariant specialisation,

```
(((g) (node) (c))
 (g96: read g)
 (g97: read node)
 (R1: node := (graph-ref g node R1:))
 (read1 read c)
 (l3: if (equal? c 'end) goto l9:)
 (l4: if (equal? c 'b) goto Rb:)
 (Ra: node := (graph-ref g node Ra:))
 (l5: goto l7:)
 (Rb: node := (graph-ref g node Rb:))
 (l7: read c)
 (l8: goto l3:)
 (l9: print (graph-ref g node accept[R]))
 (stop: end))
```

Figure 4.5: The residual program from the automaton recogniser.

the new program point are made from static stores and program points, so [l2, [one]] is distinguished from [l4, [one]].

Since the data generator algorithm resembles polyvariant specialisation, the structure of the residual graph resembles a program.

It can also be compared to a trace but one that records only the static computations and that folds back on itself when it encounters a store that it has seen before. Since it must represent traces corresponding to all dynamic input, the treatment of a static test in a particular static store causes the corresponding node to have more than one child, except when one of the branches ends the computation or returns to the same state without performing any static updatings. If one of the branches of computation leads to another dynamic test without performing static updatings the node may have three outgoing edges, *etc.*

The fact that static expressions are computed at specialisation time corresponds to constant folding [Aho-Ullman:77] but here it is done for each value of a static variable, *i.e.*, it is a kind of polyvariant constant folding.

The algorithm does not eliminate static loops. This is quite obvious from the way the residual program is constructed: there is exactly one statement for each statement in the source program.

The termination properties of the algorithm rely on the fact that the input is supposed to be an analyser program. It is simple to generalise the algorithm to arbitrary binding time patterns but then it will often happen that the graph generator does not terminate.

We have implemented a version of this algorithm for arbitrary binding time patterns (*i.e.*, more than one variable may be static). For reasons mentioned above, particularly since the algorithm does not eliminate static loops, the results are not very interesting. The residual programs have basically the same complexity as the source programs, except for the constant folding and the fact that the size of the static data now appears as a constant.

The program appear in appendix A.

4.5 Extending the algorithm of Barzdins and Bulyonkov

This section describes the transformation of the algorithm of [Barzdins & Bulyonkov 88] to treat systems of tail-recursive equations and some extensions of this algorithm.

The source languages are systems of recursive equations or *procedures*, with a Scheme syntax [Rees & Clinger 86].

4.5.1 Binding time analysis

In this section we assume that the source programs are already annotated with binding times. This is not unreasonable since the binding time analysing algorithms developed for program specialisation can be used without modification.

We choose a design where the parameters of the procedures are annotated with binding times. Thus the binding time pattern of calls to the same procedure is always the same, even though some of the parameters classified as dynamic may be static in some of the calls. In other words the binding time analysis is monovariant. It is possible, but more complicated, to make a polyvariant binding time analysis [Consel 89]

4.5.2 The algorithm for tail-recursive first-order recursive equations

This section describes how to change the algorithm from section 4.4 to treat programs written in a small language of tail-recursive first-order recursive equations.

Since it is possible to transform from a flow-chart language with jumps and conditionals to a system of tail-recursive equations and back [McCarthy 62], it is possible to make a data specialiser for a language of tail-recursive equations by adding a transformer before and (for DS_d) after the actual specialiser algorithms.

We have, however, re-designed the arbitrary binding-time pattern version of Barzdins's and Bulyonkov's algorithm to handle a tail-recursive language directly. Our tail-recursive language has the following syntax:

```
               Program  ::=  Definition Definition*
            Definition  ::=  (define (Fname Identifier*) Expression)
            Expression  ::=  (quote Expression) | Restricted-Expression | (Fname Expression*)
                          |  (if Restricted-Expression Expression Expression)
                 Fname  ::=  Identifier
Restricted-Expression  ::=  Identifier | Constant | (Prim-op Restricted-Expression*)
               Prim-op  ::=  car | cdr | cons | + | - | * | / | add1 | sub1 | null? | equal? | ...
```

We want to use the same algorithm as far as possible and generate the same kind of residual data. We still want the residual program to take the residual graph and the initial node as new arguments and we will replace static expressions by access to the current node and static updatings by updatings of the pointer to the current node.

Although the overall design of the algorithm stays the same, the new kind of language does require some changes, both in the structure of the algorithm and of a more technical nature, due to differences in the structure and concepts of the languages.

The first difference is that a such a language introduces a block structure, since each equation (or procedure) has its own set of variables. It is still limited though, because there are no nested definitions or local scope. Since the variables are no longer global, a static store has to contain an indication of its scope as well as the values of the static variables.

We will use the name that uniquely defines each equation.

The second difference is that control and data are updated simultaneously, at procedure calls, where the imperative language has separate assignment and `goto` statements. This means that the separate actions for static assignment, dynamic assignment, and control transfer performed in the imperative version now have to be performed simultaneously. When we meet a call we have to compute the values of the static arguments and generate a new node in the graph if the resulting static state has not been seen before.

Also the expression structure is more complicated than in the imperative case, since the conditional is an expression. This can be handled by descending recursively into the expression, computing the value of the completely static subexpressions and adding the corresponding attributes. This introduces one more loop in the algorithm, unless we want to use nested function calls.

Using nested calls, returning the set of additions to the graph and the list of active states, rather than tail-recursive calls returning the graph is, however, undesirable. First, it ruins any hope of self-application, since the tail-recursive language does not have nested calls. But this is a minor concern, since there are no immediate uses of self-applicable data-specialisers and anyway the algorithm will be extended to handle nested calls as well (section 4.5.3). Second it makes the algorithm less suited for program specialisation — this will be discussed in chapter 6.

The larger complexity of the expressions also makes it more advantageous to add binding time annotations to each expression by pre-processing rather than on the fly, since we then avoid descending into each expression to determine if it is static before possibly evaluating it.

Another problem is that there are no labels, so we have to find some other way to label the edges. This is in fact a symptom of the more fundamental problem that in the case of functional programs and recursive equations it is less clear what a program point is than in the case of imperative programs.

A program point is either an address to go to (such as an equation) or a place where control divides (the conditionals). In the imperative case, the distinction does not make any difference because the conditionals are also labeled statements. The Mix system [Jones et al. 89] uses the first definition. In the Similix system [Bondorf & Danvy 89], the problem is reduced to the imperative case by rewriting the source program to create a function definition for each dynamic conditional (the static conditionals do not make any difference, since they can be reduced at partial evaluation time, and thus control does not divide at that point).

Here we will say that the entries to the defined procedures are program points and label the edges with the text of the procedure call. This might not be unique, but if we have two calls in the same store with the same text, they will perform the same static updatings and thus lead to the same node, so we do not have to distinguish them.

Another problem is that it is not very efficient but for debugging during development it is convenient to have the actual text of the calls as labels in the output.

Using procedure entries as program points also has an interesting side-effect. Since we record the unique name of the procedure with the static store in order to remember the scope, we do in fact record a state (program point and store). This means that each node now represents a static state (comparable to polyvariant specialisation) rather than a static store (like in [Barzdins & Bulyonkov 88]'s algorithm).

Outline of the algorithm

The graph generating algorithm still loops over a list of active states. A state is now a procedure name and a list of values of the static arguments of the procedure. The first state is made from the first procedure and the static data and a node is generated for this state. The list of active states is initialised to the first state and the graph is initialised to its node.

For each state in "active" the algorithm finds the body of the procedure and the node of the state and specialises the expression that is the body. Expressions are specialised depending on their form and binding time. If, *e.g*, the expression is a dynamic conditional, the algorithm specialises both the test, the true-branch, and the false-branch.

If the expression is an application, the algorithm finds the values of the static parameters of the called procedure and use these and the name of the procedure to make a new state. If this state has a node in the graph, it has been seen before, so it is sufficient to make an edge from the current node to this node. If it does not have a node, it has not been seen before, so the algorithm makes a node for it as well as an edge to the node and updates the "node-of" correspondence. Since it has not been seen before, the state is also added to "active" to be specialised. Either way, the dynamic arguments are then specialised, too.

The other kinds of expressions and restricted expressions are specialised in the obvious way, *e.g.* for dynamic identifiers, nothing is done, for static identifiers an attribute is added to the current node.

The program generator is a simpler program. It takes a source program and transforms it by removing all static parameters from the procedures and adding the graph and the current node as new arguments. The bodies of the procedures are transformed by replacing all static expressions and restricted expressions by references to the current node in the graph.

4.5.3 An algorithm for first-order recursive equations

By extending the algorithm to handle procedure calls nested in calls to other procedures or primitives the source language becomes a standard language of first-order call-by-value recursive equations.

The language has the following syntax:

```
   Program  ::=  (Definition Definition*)
Definition  ::=  (define (Fname Identifier*) Expression)
Expression  ::=  Constant | Identifier | (if Expression Expression Expression)
              |  (Prim-op Expression*) | (Fname Expression*)
     Fname  ::=  Identifier
   Prim-op  ::=  car | cdr | cons | + | - | * | / | add1 | sub1 | null? | equal? | ...
```

Before we can design an algorithm treating this language, we have to find out what kind of residual graph is needed to reflected nested calls and how the residual program will act on it. Perhaps surprisingly we can use basically the same graph structure.

Let us assume that we have a residual program and a residual graph resembling the tail-recursive case. What does the residual program do in case of a nested call?

Since the call corresponds to updating the static store *and* since the static store is only represented in the residual graph, the residual program must call a residual procedure, updating the pointer to the current node in the graph. This means that the graph generator must generate nodes and edges for nested calls, just as it does for tail-recursive calls.

But a nested call returns a value to the place it is called. How does the residual program "reset" its pointer to the graph when a nested call returns its result? Do we have to add a new kind of edges to the graph to direct that?

This is in fact not necessary, because the pointer to the graph is just passed as an argument between the residual procedures, so when a nested call returns, computation continues in the scope of the calling procedure, with the graph-pointer of the calling procedure. So the path in the residual graph corresponding to a nested call simply ends in a leaf.

So in the algorithms we basically proceed as before when we meet a procedure call. The data generating algorithm computes the values of the static arguments and pair them with the name of the called procedure to make a state. If this gives a new state, it generates a new node; otherwise it makes an edge to the node of the existing state. The program generating algorithm generates a procedure call with the pointer to the current node updated to point to the node representing the state and with the graph and the dynamic arguments.

Since the language is statically scoped, the execution of the residual program returns the value of the procedure call and computation then continues with the proper value of the pointer to the current node.

The structure with nested calls, however, makes it more interesting to add another optimisation. If a call is static, that is, the procedure has only static parameters, it is possible to evaluate the procedure call at specialisation time and add the value as an attribute rather that generating a node in the graph. This way it is possible to eliminate some paths in the residual graphs, as we shall see in chapter 5.

The program is described in more detail in appendix C.

4.6 Evaluation of the Extensions

The algorithms for recursive equations generally gives smaller residual graphs than the imperative algorithm. This is because one call only gives one new node in the graph, even if several static values are updated. In the imperative case, each of the static updatings result in a node in the graph.

With the recursive equations it is possible to have even more compact results than with the tail-recursive equations, since completely static computations can be isolated in static procedures. Calls to such procedures will appear as attributes in the residual graph. In the tail-recursive case the dynamic values are usually passed as arguments to all procedures, even if they are only used to pass on in further calls. This causes nodes and edges in the residual graph, giving generally larger results.

If the source programs are written carefully this gives significant improvements. Unlike the imperative and tail-recursive cases it is now possible to eliminate statically determined control structures of the source program, not just compute the static expressions. In other words it is now *possible* to eliminate all computations depending on static data, so that the complexity of the residual program only depends on the dynamic data.

The algorithms for recursive equations still have some of the problems of the imperative algorithm, however.

4.6.1 Conditionals

There is at least one point about the algorithms described above, that does not seem satisfactory. This is the specialisation of static conditionals.

Unlike polyvariant program specialisation, it is not possible with this kind of data specialisation to eliminate a conditional if only the test is static. This is because the expression corresponding to the test in the residual program may correspond to one of several static stores, unlike program specialisation, where there will be one version of the expression for each static store in which the test occurs.

To avoid this problem, it would be necessary to design a different pair of data specialisation algorithms that generates a different kind of residual data.

4.6.2 The Structure of the Residual Data

All the algorithms described above generate the same kind of residual data. The only difference from one algorithm to the next is a gradual extension of the source language structures that can be treated and in the complexity of the results.

However there is nothing in the definition of data specialisation that limits the kind of data that can be generated.

Even generating the same structure it is possible to interpret it as another structure, depending on its representation. If, *e.g*, the graph is represented as a table of nodes, where each node contains a table of attributes and a table of pointers to other nodes (with the label of the edge as the key to the table), then the generated output could as well be conceived as a table of tables, rather than a graph. This might be reasonable for the kind of problems that are usually solved using tables.

4.7 Adding unfolding

The algorithms presented so far do not unfold function calls. This follows the spirit of the original algorithm from [Barzdins & Bulyonkov 88] which mainly focused on the generation of the data structure and made only minimal changes to the program.

If we want to execute all computations depending on the static data, however, we also want to remove the control structures depending only on the static data from the residual program. As already mentioned this can in some cases be done by moving them into a completely static procedure. If we add unfolding it is possible to have a similar effect even though some arguments are dynamic.

The idea of call unfolding is that when some of the data are known, it is not necessary to have as many residual program points (procedures) since some of the alternatives in the computation path have been removed (the static conditionals). In other words, there is no reason to generate nodes in the residual graph for the procedures that contain only static conditionals. In graph terms, there is no reason to generate chains where each node only has one child.

So although unfolding introduces the risk of call duplication and infinite unfolding (*cf.* section 3.3.2), it still seems that there are definite advantages to introducing call unfolding in a data specialiser with the structure described above. The size of the residual data depends (among other things) on the number of residual procedures and residual procedure calls and since an unfoldable call will not be a call in the residual program, there will not be generated nodes or edges at the unfoldable calls.

If *all* calls to a procedure are unfoldable its specialised version will never be called, and not even occur, in the residual program. On the other hand the remaining residual procedures may be larger, so there is not necessarily any improvement in the size of the residual program.

Changes to the algorithm

Let us for simplicity assume that the calls are annotated by preprocessing as unfoldable or residual. This annotation is furthermore assumed to be safe (no duplication, no infinite unfolding).

Call unfolding involves additions to both the data generator and the program generator.

The graph generator proceeds as before when it meets a residual call and when it meets an unfoldable call where all arguments are static.

When it meets an unfoldable call with dynamic arguments, it has to continue as if the body of the called procedure was in the place of the call. This means that it does not need to generate any new nodes or edges, nor any new state to add to the list of active states. It does have to generate the attributes corresponding to the body of the called procedure, however.

One way to do this is to fetch the body from the program when the call is encountered, perform the necessary β-conversion and add the resulting expression to the list of expressions to be treated in the current state.

The program generator has to do basically the same: insert the body of the called procedure instead of each unfoldable, non-static call.

Unfoldability analysis

With this strategy it is not immediately possible to use the unfoldability annotating algorithms developed for program specialisation.

The problem is to determine when it is safe to unfold, that is, when we can unfold a procedure call without risking infinite unfolding. In program specialisation it is safe to unfold if the procedure we unfold does not contain a dynamic conditional.

Unfortunately, this is not sufficient with the present kind of data specialisation. The problem is that if we have a static loop, such as the one in append with static first argument, then it is safe to unfold it in program specialisation because we know the actual value, and in particular the length, of the static string. In data specialisation, with the present division of information into the residual data and the residual graph, it would not be safe to unfold the recursive call to append, because the program generator does not know the length of the static string and so would try to unfold the call infinitely.

Instead, it is only safe to unfold calls that are part of a cyclic chain of calls where at least one of the calls will not be unfolded. *E.g.*, if we have two co-recursive procedures then it is safe to unfold one of the calls but if we have one recursive procedure it is not safe to unfold the recursive call, whether it contains dynamic conditionals or not.

This is related to the problem of conditionals with static tests where data specialisation is also less powerful than program specialisation because the residual program generated by data specialisation basically has the structure of the source program and thus each program point corresponds to several static stores.

The more restricted unfolding is thus less powerful than unfolding in program specialisation but still useful.

Because we have to use this restricted unfolding it is also possible to preprocess the source program, unfolding all unfoldable calls. Then we could use the previous version of the data specialisation algorithm on the unfolded source program.

Preprocessing might in fact be faster, since a large part of the unfolding will otherwise be performed repeatedly. Both the program generating algorithm and the data generating algorithm unfolds, the data generating algorithm even every time a call is encountered in the symbolic evaluation.

4.8 A depth-first algorithm

All the algorithms described above have traversed the computation tree in a breadth-first manner, *i.e.*, leaving the new state on a pending-list to be treated later when it encountered a procedure call. This seems to be the tradition in polyvariant specialisation [Jones et al. 89] [Consel 89] although other areas treating programs as data objects (*e.g.*, denotational – and other – semantics) usually descend in a depth-first manner.

It is indeed perfectly possible to write a depth-first data generating algorithm or a depth-first program specialiser. Such an algorithm treats the procedure call and instead stack the rest of the treatment of the current state either indirectly, by a nested call in the specialiser, or directly in a control (continuation) stack [Kehler & Danvy 89].

To keep track of already encountered states such an algorithm would, for example, have to pass a list of treated states along with the continuation. In our breadth-first data generating algorithm, this is recorded in the residual graph, that contains a node for each encountered state.

4.9 Towards a Formal Derivation of Data Specialisation Algorithms

This section sketches some very lose ideas on how to derive a data specialisation algorithm from a denotational semantics of a language.

Deriving a data generating algorithm *and* a program generating algorithm resembling the ones presented in this chapter from a denotational semantics has some inherent problems. These stem from the fact that a polyvariant data generating algorithm symbolically evaluates the expressions, *i.e.*, it walks down the computation tree. The corresponding program generating algorithm walks down the actual syntax tree of the source program, rewriting each expression. Thus the generation of program and data cannot be done by the same algorithm.

It is certainly possible to design a pair of algorithms that both go down the syntax tree. It seems likely that such a data specialiser will not be very interesting. It is, on the other hand, not a good idea to design a pair of algorithms that will both go down the computation tree, since the program generator must be independent of the actual values of the static data, and without those, the computation tree will almost always be infinite.

One way to get around this problem is to walk down the syntax tree and generate a residual program *and* a λ-term that, given the static data, generates the residual data. This λ-term is constructed from actions to be taken for each syntactic construction and performs the walk down the computation tree in order to generate the residual data.

A possible way to find such a generating semantics is to extend the standard semantics with graph generating terms, *i.e.*, to a kind of trace semantics that only traces static data.

The generating semantics could then be derived from this in a process resembling the actions of *cogen*, where actions are turned into other actions rather than being performed.

Assuming that such a generating semantics could be found, we could find a data specialisation algorithm by

- Finding a semantics of residual programs so that the meaning of a source program is the same as the meaning of the corresponding pair of residual program and data generating term (notice that they have the same functionality since we have not yet used the static data).

- Separate the equations in this semantics into two sets of equations, one that applies the graph generating terms to the static data and one that applies the residual program to the resulting graph. This is possible because only the graph generating parts uses the static data and only the residual program refers to the dynamic data. (This strongly resembles the technique used in [Jørring & Scherlis 86] to derive a compiler from a denotational specification.)

- Performing the data generating action corresponding to a syntactic construction directly when encountering the construction rather than generating the λ-term and using it (giving the data generating algorithm). The residual program is generated by the generating semantics.

Chapter 5

Application to String Matching

5.1 Introduction

This chapter describes the application of program specialisation and data specialisation to string matching algorithms.

Readers more interested in the principles may wish first to read chapter 6.

The goal is to obtain residual programs with a run-time that is linear in the length of the dynamic string, out of source programs with a quadratic run-time.

We first consider a naive string matching algorithm with quadratic run-time, which specialises to quadratic residual programs both for a static pattern string and for a static subject string.

Then we analyse why the results are quadratic and rewrite the algorithm so that it specialises better. This leads to a distinction between algorithms that will specialise well with respect to static pattern and dynamic subject and algorithms that will specialise well with respect to dynamic pattern and static subject.

It is seen that the rewriting of the naive algorithm follows rather straightforward principles and in relation to this various topics are discussed:

- how to automatize the rewriting,
- how to eliminate the need for rewriting by extending the partial evaluator,
- whether the rewriting techniques used for string matching are generally useful and sufficient to apply to other programs before specialising.

By comparing the results of program specialisation with the results of data specialisation, it is seen that the problems of static conditionals and of unfolding static loops mentioned in sections 4.6.1 and 4.7 cause the data specialisation algorithm to give less good results.

The program specialiser used is the Similix system [Bondorf & Danvy 89].

5.1.1 Notation

All programs shown in this section are written in a subset of Scheme [Rees & Clinger 86] that corresponds to the language of recursive equations given in section 4.5.3. The programs shown are the actual input to Similix, which includes a binding time analyser and a call unfolding analysis. The input to the data specialiser are versions of the shown programs that have been hand-annotated with information about binding times, unfoldability, and syntax. The programs have not been changed in any other way.

The output of the data specialiser has been hand-modified to shorten the labels of attributes and edges for readability, but not in any other way.

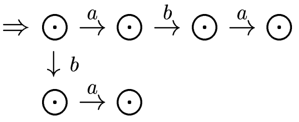
The target language of data specialisation is assumed to contain primitive procedures that access the graph. Presently these are in fact defined in a separate file. The program generator then generates a call to the Similix procedure `loadt` as the first part of the residual program. [1] This load statement is not shown.

In this chapter we will show the residual graphs as graphs where the attributes are written inside the nodes and the labels on edges shortened to clarifying names. The actual output appears in appendix D.

For space and clarity reasons we will also sometimes use a graphical expression of residual programs obtained by program specialisation where a successful test against the character `a` is noted as the transition $\xrightarrow{a}$, and where each node $\odot$ has two implicit transitions: a success one if the pattern ends there, and a failure if no character matches the explicit transitions. As an example the program

```
(define (match?-0 pat_0)
   (let ((m_3_0_ (length pat_0)))
      (or (= 0 m_3_0_)
          (cond
             ((equal? 'a (list-ref pat_0 0))
              (or (= 1 m_3_0_)
                  (and (equal? 'b (list-ref pat_0 1))
                       (or (= 2 m_3_0_)
                           (and (equal? 'a (list-ref pat_0 2))
                                (= 3 m_3_0_))))))
             ((equal? 'b (list-ref pat_0 0))
              (or (= 1 m_3_0_)
                  (and (equal? 'a (list-ref pat_0 1)) (= 2 m_3_0_))))
             (else ())))))
```

corresponds to



For the complexity considerations in this chapter, it will be assumed that references to attributes and following edges in the residual graph can be done in constant time. This is possible in principle, although it is not the case in the current implementation.

5.2 Definitions

We consider the boolean problem of whether one string, the pattern string, occurs as a substring of another string, the subject string.

We want to automatically generate specialised algorithms for the two cases where either the pattern string or the subject sting are known in advance.

[1] It would also be possible just to insert the definitions but then the procedures would be loaded several times if we want to run several residual programs, whereas `loadt` only loads files that are not already loaded.

These problems have been solved by hand, for a static pattern, *e.g.*, in [Knuth, Morris & Pratt 77] [Boyer & Moore 77] and for a dynamic pattern,*e.g.*, in [Weiner 73]. In both cases these solutions consist of having a preprocessing phase which takes the static string and linearly transforms it into an intermediate structure. This structure is then used by a matching algorithm which takes the dynamic string and, with a complexity linear in the length of the dynamic string, determines whether there is a match. In the case of a static pattern, it might also be feasible to integrate the intermediate structure into the matching algorithm to obtain faster matching. In the case of a static subject, this will typically be undesirable for space reasons.

We will not try to match these results, since the polyvariant specialisation of a quadratic matching algorithm symbolically evaluates the algorithm with respect to the static string. Thus this phase will not be linear.

We do, however, expect to obtain results, comparable in complexity to the hand-written algorithms, for the size of the intermediate data structures and the run-time of the residual matching programs.

Notation

We use ω to denote a string over the alphabet Σ, a, a' for symbols from Σ and we always assume $\omega = a_1 \dots a_n$ for the subject string and $\omega' = a'_1 \dots a'_m$ for the pattern string.

The concatenation of two strings, ω_1 and ω_2, is denoted $\omega_1\omega_2$.

If $a \in \Sigma$ we use a to denote both the character a and the one element string a.

The length of a string ω is denoted $lg(\omega)$. We will always assume the length of the pattern to be m and the length of the subject to be n. We will usually index the pattern by j and the subject by i. These conventions are employed in the shown programs as well.

We say that a string $\omega' = a'_1 \dots a'_m$ occurs in a string $\omega = a_1 \dots a_n$ at position i if $a_i = a'_1, \dots, a_{i+m} = a'_m$.

If $G = (V, E)$ is a labeled directed graph, with the set V of vertices and the set E of labeled edges, then an edge e in G labeled a out of v_1 into v_2 is also denoted (v_1, v_2, a).

5.2.1 The Knuth-Morris-Pratt matching algorithm for static pattern strings

The Knuth-Morris-Pratt matching algorithm [Knuth, Morris & Pratt 77] can be used in cases where the pattern is known in advance. The idea is to use regularities in the pattern string to determine from which position in the pattern to continue matching in case of a mismatch (operationally: "how far the pattern can be moved to the right"). If, for example, the first four characters in the pattern are all different, then we know that if we match those four characters against the subject and then have a mismatch on the fifth, we can immediately move the pattern four positions to the right and compare the mismatching character against the first character in the pattern, since the first character does not match any characters in between.

More concretely, the pattern is used to generate a *failure table* (or *failure function*), with the same length as the pattern, giving for each position j the largest position i in the pattern so that the prefix up to i is a suffix of the prefix up to j.

The failure table is then used by the actual matching algorithm to find the new value of the index into the pattern in case of a mismatch.

Definition 7 *The failure function f_ω of a string ω is a function defined for all positions in ω, so that*

$$f(j) = i \Leftrightarrow a_1 \dots a_i \textit{ is the longest proper suffix of } a_1 \dots a_j$$

□

In particular $f(1) = 0$.

As an example, the failure table for the string "ababc" is

a	b	a	b	c
0	0	1	2	0

If the matching algorithm tries to match, *e.g.*, against "abaab", it will succeed to match "aba" and fail on "b". Then it can see in the table that it can continue the match from position 2, since it knows that position 1 matches.

5.2.2 Definitions related to a static subject string

Definition 8 *For a string $\omega = a_1 \dots a_n$ over an alphabet Σ and a character $t \notin \Sigma$ the* **prefix identifier** *for position i, $p(i)$, is a substring $a_i \dots a_j$ of ωt so that $a_i sem - sec \dots a_j$ does not occur anywhere else in ωt, but $a_i \dots a_{j-1}$ occurs in at least one position different from i.* □

That is, the prefix identifier for a position i is the shortest substring of ωt which starts with a_i and does not occur anywhere else in ωt. To guarantee that such a substring exists, ω is extended with a termination character t, which is not an element of Σ.

Definition 9 *If $G = (V, E)$ is a tree with edges labeled with elements of an alphabet Σ, $n, n' \in V$ and there is a path $(n, n_1, \dots, n_i, n')$ from n to n', then* **the string from** n **to** n'*, denoted $T(n, n')$, is the string of the labels of (n, n_1), $(n_1, n_2) \dots (n_i, n')$.* □

Definition 10 *The* **prefix tree**, *$P(\omega) = (V, root, E)$ for a string ω is a rooted tree with edges labeled with elements of $\Sigma \cup \{t\}$ with:*

- *at most one edge labeled a for each $a \in \Sigma \cup \{t\}$ out of each node*
- *an edge labeled a out of a node n into a node n' iff $T(root, n)a$ is a prefix identifier of a position in ω.*

Furthermore every leaf, l, of the tree is labeled with the position i that $T(root, l)$ identifies. □

Definition 11 *The* **compacted prefix tree** *$P^c(\omega) = (V', root, E')$, for a string ω is a compression of the prefix tree, $(V, root, E)$ for ω given by:*

- *$v \in V'$ iff $v \in V \wedge (v$ has at least two sons in $P(\omega) \vee$ the father of v has at least two sons in $P(\omega))$*
- *$e = (v, v', a) \in E'$ iff $v, v' \in E' \wedge$ for the nodes $v_1, \dots, v_i$ in the path $(v, v_1, \dots, v_i, v')$ in $P(\omega)$, $v_1, \dots, v_i \notin V' \wedge$ the first character of $T(v, v')$ in $P(\omega)$ is a.* □

To use a compacted prefix tree for ω to determine whether a string ω' is a substring of ω we need a copy of ω and at each node, v, the information of the length of $T(root, v)$ in $P(\omega)$ and of the position that the edge into v is part of the prefix identifier for. (If there are several, any one will do.)

Conceptually this can be understood as extending the compacted prefix tree with the "rest" of ω at each leaf and with the missing pieces in the interior, thus making it a kind of "substring tree" (defined below).

The prefix tree and the compacted prefix tree comes from [Weiner 73] where they are used as intermediate data structures for a linear matching algorithm. In [Aho, Hopcroft & Ullman 74] a prefix tree is also called a position tree.

The advantage of the compacted prefix tree over the un-compacted one is that it is faster to generate. Both trees are generated in time $O(k)$, where k is the number of nodes, and the compacted trees have $O(n)$ nodes whereas the un-compacted have $O(n^2)$ nodes. The algorithms for matching the trees against the pattern string, however, take $O(m)$ time in both cases.

For the purpose of this report we define:

Definition 12 *The* **substring tree** *of a string ω, $S(\omega)$ is is a rooted tree with edges labeled with elements of Σ fulfilling:*

- *For each node n and for each $a \in \Sigma$ there is at most one edge labeled a out of n and*
- *for each node n there is an edge labeled a out of n into a node n' iff $T(root, n)a$ is a substring of ω.* □

Thus the substring tree contains full suffixes of the tree, whereas the prefix tree only contains prefix identifiers. This makes the substring trees larger, both prefix trees and substring trees have a worst case size of $O(n^2)$ nodes but prefix trees have an average size of $O(n)$ [Aho, Hopcroft & Ullman 74], whereas substring trees only have an $O(n)$ size in some special cases (*e.g.* a^n).

But the algorithm for substring matching is simpler for the substring tree, since it just has to walk down the tree according to ω'.

It can also be noticed that the substring tree does not contain any indication of the position of the substring, since here we only consider the boolean function "is ω' a substring of ω?", not at which position the match occurs.

This is also why we can omit the termination character which guarantees that all positions have a prefix identifier. If a position i does not have a prefix identifier it is because the suffix of ω starting at i occurs somewhere else in ω and if we are only interested in *if* there is a match, it does not matter at which occurrence we find the match, so we might as well use the first.

Fortunately the space complexity can be reduced by *folding* the substring tree to a substring graph: if two nodes k_1 and k_2 have isomorphic subtrees the nodes (and their subtrees) can be collapsed. The resulting (directed) graph still has the property that there is a path labeled $a_i \dots a_j$ in the graph if and only if $a_i \dots a_j$ is a substring of ω and that there is at most one edge labeled a out of any vertex for each $a \in \Sigma$.

Definition 13 *The* **folded substring graph** $S^f(\omega) = (V^f, root, E^f)$ *for a string ω is a compressed version of the substring graph $S(\omega) = (V, root, E)$, fulfilling:*

- *$v \in V^f$ iff $v \in V \wedge$ there are no other vertices in $S(\omega)$ with the same subtree as v.*
- *$e = (v, v', a) \in E^f$ iff $e \in E \wedge$ both v and v' fulfill the criteria above.*
- *$\forall e_1 = (v_1, v_1', a_1), \dots, e_n = (v_n, v_n', a_n)$ in $S(\omega)$ for which all v_i' have identical subtrees in $S(\omega)$ but none of v_i do, $S^f(\omega)$ have n edges, e_i^f, out of v_i into a new vertex, $(v_1', \dots, v_n')$, labeled a_i and $(v_1', \dots, v_n')$ have the same subtree as v_1'.*

This is another characterisation of the structure that in [Blumer et al. 85] is called a directed acyclic word graph.

The space complexity of a directed acyclic word graph is $O(n)$ both in nodes and edges. In fact there is an upper limit of $2n-1$ on the number of nodes and $3n-4$ edges [Blumer et al. 85].

It must be noticed, however, that the folding relies on the fact that there is no position information in the tree, since two nodes representing different positions may be collapsed, making it impossible to distinguish the positions.

5.3 Specialising the naive algorithm

5.3.1 A naive, quadratic, pattern matching algorithm

We start with a simple matching algorithm. The algorithm starts from the head of both strings and compares the characters one by one. If it finds a match it continues with the next position. If it finds a mismatch it backs up to the first position in the pattern, shift the pattern one position to the right, and continues.

This algorithm obviously has the worst case complexity $O(mn)$. An implementation in Scheme is shown below.

```
(define (match? sub pat)
    (find-match sub 0 (length sub) pat 0 (length pat)))

(define (find-match sub i n pat j m)
    (if (= j m)
        #t
        (find-match-aux sub i n pat j m)))

(define (find-match-aux sub i n pat j m)
    (if (= i n)
        #f
        (if (equal? (list-ref sub i) (list-ref pat j))
            (find-match sub (add1 i) n pat (add1 j) m)
            (find-match-aux sub (add1 i) n pat 0 m))))
```

When this program is specialised with respect to a static subject or pattern we obtain a residual program where the characters of the static string are integrated into the residual output (either program or data). Except for that, the residual program will perform exactly the same computations as the source program. In particular, it will still be quadratic, in the sense that it will be $O(mn)$. Since one of the strings is now fixed, this is in fact linear in n, but this does not correspond to an improvement, only to the fact that the length of the static string is fixed.

Particularly in the case of data specialisation, it is clear that the residual program performs the same algorithm as the source program.

As an example, the results of program specialisation and data specialisation of this program with respect to `(a b a)` are shown in appendix D, both for static pattern and static data. The graphical representations of the results of program specialisation is shown in fig. 5.1.

Since we know that it is possible to write linear (that is, $O(n+m)$) matching algorithms, let us try to find out why our specialisers cannot reproduce such results directly.

Static pattern:

$$\Rightarrow\ lg(pat) \neq 0 \xrightarrow{first} a, (j+1) \neq m \xrightarrow{j+1} b, (j+1) \neq m \xrightarrow{j+1} a, (j+1) = m$$

Static subject:

$$\Rightarrow \odot \xrightarrow{first} a,\ i \neq n \xrightarrow{j+1} \odot \longrightarrow b,\ i \neq n \xrightarrow{j+1} \odot \longrightarrow a,\ i \neq n \xrightarrow{j+1} \odot \longrightarrow i = n$$

Figure 5.1: The graphical representation of the result of program specialising the simple algorithm.

5.3.2 Analysis of the quadratic result

If we look at the result of the data specialisation with respect to a static pattern, we see that the graph is structured exactly as the algorithm proceeds. It has a node for each position in the pattern with two outgoing edges: one going to the next position, corresponding to a match, and one back to the initial node, corresponding to a mismatch.

In the case of static subject, the residual graph is again structured exactly as the algorithm proceeds on the static string. It has one node for each position in the subject and edges going down the subject updating `j`, corresponding to matches, and from each node to the next node without updating, corresponding to mismatches.

The residual program looks in both cases exactly like the source program, except that all references to the static string are replaced by references to the graph. Thus it is quite obvious that the complexity of the residual programs is still $O(mn)$ – assuming that a graph-reference takes constant time.

It is clear that the graphs are redundant. Some of the paths will never be followed and some will be followed with no result.

As an example of the first, consider the path in the static pattern graph going first right, corresponding to a match on the character `a`, then right, corresponding to a match on `b`, then back, corresponding to a mismatch and then right, corresponding to a match on `a`. This path will obviously never be taken, since the second character in the subject cannot both match `a` and `b`. So the test on equality with `a` is redundant in the cases where the node is reached after a successful match.

As an example of the second, consider the path in the static subject graph taking first the lower edge, corresponding to a mismatch on `a`, then again the lower edge, corresponding to a mismatch on `b`, and then again the lower edge, corresponding to a mismatch on `a`. The last test is redundant: since the first character in the pattern was not `a`, the first time it was tested, it will not be `a` the second time either.

In the case of program specialisation, it is harder to make a provably correct statement about the complexity of the residual programs in general, since each residual program depends on the static data. It could probably be proven by analysing $PE\,[PE, \textit{string matcher}]$ but, knowing that the Similix partial evaluator uses polyvariant specialisation, it is not hazardous to guess that it is always $O(mn)$, as in the shown example.

5.3.3 Suggestions for improvement

The fact that the residual graphs are redundant seems to imply that, in the generation of the graph, we ignore or forget some of the information that is present.

This is in fact the case and if we consider the matching algorithm carefully, we can see where the loss of information occurs.

Let us first observe that if we succeed to match some positions, then we could use that information. *E.g.*, if we have a match in the three first positions and then a mismatch, then we know the characters at those three positions in the dynamic string. If we use this information, it is possible to change the binding time of some of the tests in the algorithm from dynamic to static, so that they can be performed at specialisation time.

In term of specialisation, it can be seen like this.

If we compare the values of a static variable and a dynamic variable in a conditional, we do not know the value of the test, so we have to specialise both the true-branch and the false-branch. When we specialise the true-branch, however, we know that the dynamic variable has the value of the static variable in the test since we have assumed that the test is true. This holds until the dynamic variable is updated. If the updating is static (*e.g.*, `add1`), we might even evaluate it and record the new value.

This is not done in the present specialisers, and thus we lose some information. This can be improved either by changing the specialiser or by rewriting the algorithm to specialise well with respect to a particular binding time pattern. Here we will rewrite the algorithm and only outline (see section 5.7) the possible additions to a specialiser.

Let us then observe that even if we do not have a match, we do have some information about the current position in the dynamic string, *i.e.*, the information that it is *not* equal to the current position in the static string.

This observation is in fact dual to the observation above: if we specialise the false-branch in a dynamic conditional, we know that the condition is false. Particularly if the condition tests on equality between a static and a dynamic variable, we know that the dynamic variable does not have the value of the static variable in the test.

Since the observation is dual, it is also here possible to use it either by extending the specialiser or by rewriting the program.

This can also be said in another way.

The redundancy is caused by the fact that the specialisers do not memorize the assumed results of the dynamic test. *I.e.*, when the test `(equal? (car l1) (car l2))` is encountered, the specialiser cannot evaluate it because `l1` is dynamic. Instead it has to specialise both branches, since both paths could be taken in the actual computation. But if the static list has the same value occurring later, this will be encountered at the head later in the treatment of, *e.g.*, the true-branch. Then the result of the test is known, since it must be the same as last time. Thus there is no reason to specialise the false-branch this time. But the specialiser does not know this, since it does not save the assumed values of dynamic tests. It will also specialise the false-branch and thus generate redundant results.

One way to get around this is to rewrite the specialiser so that it memorises the assumed results of dynamic tests [Haraldsson 77] [Futamura & Nogi 88].

Another way is to rewrite the program so that it does not compare a static value against a dynamic value unless the static value has never been compared against that dynamic value before. This means that the memorization will be done in the source program.

Such a rewriting of the source program is not complicated to do but since it depends on the binding time pattern the resulting program will only yield good results when specialised with that particular binding time pattern.

```
(define (match? sub pat)
    (find-match sub 0 (length sub) pat 0 (length pat)))

(define (find-match sub i n pat j m)
    (if (= j m)
        #t
        (find-match-aux sub i n pat j m)))

(define (find-match-aux sub i n pat j m)
    (if (= i n)
        #f
        (if (equal? (list-ref sub i) (list-ref pat j))
            (find-match sub (add1 i) n pat (add1 j) m)
            (if (zero? j)
                (find-match-aux sub (add1 i) n pat 0 m)
                (find-match-aux sub i n pat (length-s-match pat 0 1 j) m)))))

(define (length-s-match pat k start j)
    (if (= (+ start k) j)
        k       ; pos start to start + k - 1 matches pos 0 to k-1
        (if (equal? (list-ref pat (+ start k)) (list-ref pat k))
            (length-s-match pat (add1 k) start j)
            (length-s-match pat 0 (add1 start) j))))
```

Figure 5.2: Matching program with static re-matching to specialise well with respect to a static pattern.

5.4 Obtaining the Knuth-Morris-Pratt Algorithm

In this section we will assume that the pattern is known in advance and we will see how to obtain good results when we rewrite the pattern matching algorithm to take advantage of this.

[Consel & Danvy 89] uses comparable techniques with a partial evaluator to obtain compiled failure tables.

5.4.1 Rewriting the algorithm

As mentioned above we want to rewrite the algorithm so that it makes use of the information obtained about the subject when it compares a position in the pattern against a position in the subject.

In order to use the positive information, *i.e.*, the information assumed when we take a true-branch in the equality test, we change the algorithm to match the pattern against itself, instead of the subject, when we have a mismatch after a match of positive length. This matching returns the length of the statically detected match (possibly zero) using a procedure `length-s-match`. That number is then used as the next index of the pattern. The resulting algorithm is shown in fig. 5.2.

Its complexity is still $O(mn)$.

We can see that it always moves forward on the subject and since the matching of the pattern against itself is completely static, this will be performed at specialisation time. This means that the residual program after data specialisation is linear. This can be confirmed by analysing the residual program of data specialisation, which is given in fig. 5.3.

It is worth to notice that the static matching is only performed if we have a match of positive

```
(define (match? current g sub)
    (if (find-att '(= 0 (length pat)) (fetch-atts (find-node current g)))
        #t
        (find-match-aux
         (find-edge '(find-match-aux sub 0 (length sub) pat 0 (length pat))
                    (fetch-edges (find-node current g)))
         g sub 0 (length sub))))

(define (find-match-aux current g sub i n)
    (if (= i n)
        #f
        (if (equal? (list-ref sub i)
                    (find-att '(list-ref pat j) (fetch-atts (find-node current g))))
            (if (find-att '(= (add1 j) m) (fetch-atts (find-node current g)))
                #t
                (find-match-aux
                 (find-edge '(find-match-aux sub (add1 i) n pat (add1 j) m)
                            (fetch-edges (find-node current g)))
                 g sub (add1 i) n))
            (if (find-att '(zero? j) (fetch-atts (find-node current g)))
                (find-match-aux
                 (find-edge '(find-match-aux sub (add1 i) n pat 0 m)
                            (fetch-edges (find-node current g)))
                 g sub (add1 i) n)
                (find-match-aux
                 (find-edge '(find-match-aux sub i n pat (length-s-match pat 0 1 j) m)
                            (fetch-edges (find-node current g)))
                 g sub i n)))))
```

Figure 5.3: The result of data specialising 5.2.

length. Before matching the pattern against itself, it is checked whether there is any pattern to match at all. The point is that in this case the index into the subject must be updated. If this was not done, the program would loop.

Now the residual graph is comparable to a failure table. For each position in the pattern, the residual graph has a node. Such a node has three attributes, one giving the character at that position and two signifying whether the current node is the last node and whether it is the first node, and two outgoing edges, one to the next position and one to the "failure position",*i.e.*, the position from where to continue the matching if the matching of position j has failed. A residual graph for `(a b a b c)` is shown in fig. 5.4.

The residual program (fig. 5.3) is also comparable to the Knuth-Morris-Pratt algorithm.

The test on whether the node is the first node serves to determine whether the index into the subject should be increased. The test on whether the node is the last one tests whether the match has succeeded.

[Knuth, Morris & Pratt 77] also mentions that for shorter patterns it might be worth to compile the pattern into the matching program. This is what is obtained by program specialisation, as shown in fig. 5.5.

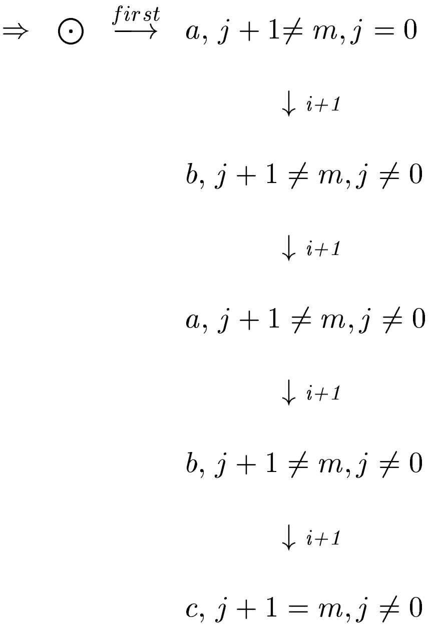

⇒ ⊙ $\xrightarrow{first}$ $a, j+1 \neq m, j=0$
$\downarrow$ i+1
$b, j+1 \neq m, j \neq 0$
$\downarrow$ i+1
$a, j+1 \neq m, j \neq 0$
$\downarrow$ i+1
$b, j+1 \neq m, j \neq 0$
$\downarrow$ i+1
$c, j+1 = m, j \neq 0$

Figure 5.4: The residual graph when specialising the program of fig. 5.2.

5.5 Obtaining Suffix Trees and Automata

In this section we assume that the subject is known in advance and try to apply the same techniques as in the previous section to obtain a linear algorithm.

It is seen that the result will still contain some redundancies. This is seen to be because we do not use the negative information in the conditionals. Although this can be remedied by rewriting the algorithm, this rewriting introduces new problems.

The reason of this is isolated and the problem is remedied using a less obvious rewriting technique.

5.5.1 Rewriting the algorithm

First we simply apply the same rewriting techniques as used in the previous section, but to the subject rather than the pattern. Thus we introduce a match of the subject against itself to determine the position where we can take up the matching against the pattern after a mismatch.

In order to do this, however, we need to know the length of the part to re-match, that is, the value of the variable `j` must be static. Since `j` is initialised to a constant, its value can be known at specialisation time. But this does not mean that we are free to declare it static. The problem is that it is only bounded by the length of the pattern, which is dynamic.

This is a case of the well-known specialisation problem of "static updating under dynamic control". According to the polyvariant specialisation principle we want to generate one residual procedure/node in the graph for each possible value of `j`. `j` takes all integer values from zero to the length of the pattern. Since the pattern is dynamic, its length is assumed to have all its possible values, *i.e.*, all integer values. Thus the specialiser will try to generate infinitely many residual procedures/nodes.

This can be avoided by classifying the static variable as dynamic instead or by placing it under static control, either by rewriting the program or by adding a facility to the specialiser to specify an

```
(define (match?-0 sub_0)
    (let ((i_3_0_ (generalize 0)))
        (let ((n_4_1_ (length sub_0)))
            (find-match-aux-1 sub_0 i_3_0_ n_4_1_))))
(define (find-match-aux-1 sub_0 i_1 n_2)
    (and (not (= i_1 n_2))
         (if (equal? (list-ref sub_0 i_1) 'a)
             (let ((i_6_0_ (add1 i_1)))
                 (find-match-aux-3 sub_0 i_6_0_ n_2))
             (find-match-aux-1 sub_0 (add1 i_1) n_2))))
(define (find-match-aux-3 sub_0 i_1 n_2)
    (and (not (= i_1 n_2))
         (if (equal? (list-ref sub_0 i_1) 'b)
             (let ((i_6_0_ (add1 i_1)))
                 (find-match-aux-5 sub_0 i_6_0_ n_2))
             (find-match-aux-1 sub_0 i_1 n_2))))
(define (find-match-aux-5 sub_0 i_1 n_2)
    (and (not (= i_1 n_2))
         (if (equal? (list-ref sub_0 i_1) 'a)
             (let ((i_6_0_ (add1 i_1)))
                 (and (not (= i_6_0_ n_2))
                      (if (equal? (list-ref sub_0 i_6_0_) 'b)
                          (let ((i_6_1_ (add1 i_6_0_)))
                              (and (not (= i_6_1_ n_2))
                                   (if (equal? (list-ref sub_0 i_6_1_) 'c)
                                       (let ((i_6_2_ (add1 i_6_1_))) #t)
                                       (find-match-aux-5 sub_0 i_6_1_ n_2))))
                          (find-match-aux-3 sub_0 i_6_0_ n_2))))
             (find-match-aux-1 sub_0 i_1 n_2))))
```

Figure 5.5: The residual program when program specialising the program of fig. 5.2 with respect to `(a b a b c)`.

```
(define (match? sub pat)
    (find-match sub 0 (length sub) pat 0 (length pat)))

(define (find-match sub i n pat j m)
    (if (= j m)
        #t
        (find-match-aux sub i n pat j m)))

(define (find-match-aux sub i n pat j m)
    (if (>= (+ i j) n)
        #f
        (find-match-aux-2 sub i n pat j m)))

(define (find-match-aux-2 sub i n pat j m)
    (if (equal? (list-ref sub (+ i j)) (list-ref pat j))
        (find-match sub i n pat (add1 j) m)
        (find-match-aux sub (pos-next-match sub (add1 i) n 0 i j) n pat j m)))

(define (pos-next-match sub i n j start msf)
    (if (= j msf)
        i
        (pos-next-match-aux sub i n j start msf)))

(define (pos-next-match-aux sub i n j start msf)
    (if (= (+ i j) n)
        i
        (if (equal? (list-ref sub (+ i j)) (list-ref sub (+ start j)))
            (pos-next-match sub i n (add1 j) start msf)
            (pos-next-match-aux sub (add1 i) n 0 start msf))))
```

Figure 5.6: Rewriting the algorithm to specialise with respect to the subject.

upper bound on the number of unfoldings of any call in the program [Consel 89].

In the present case it is simple to rewrite the program to bring j under static control. We change the function of the variable i from denoting the current position in the subject to denoting the start of the current match. Then we compare position (+ i j) in the subject with position j in the pattern and both i and j are bounded by the condition (>= (+ i j) n).

5.5.2 Results of specialisation

The resulting algorithm is shown in fig. 5.6 and the result of specialising with respect to (a b a) is shown in fig. 5.7

Although the program is now linear, the residual graph is still redundant. This is because we still do not use the negative information that is assumed when taking a false-branch in the equality test as described in section 5.3.2.

In order to do this, we should save the characters of the subject that have been compared against

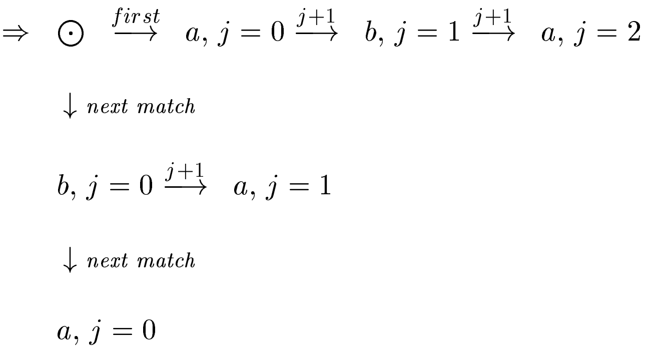


Figure 5.7: The residual graph when the program is specialised with respect to `(a b a)`.

a particular position in the pattern. For this purpose we introduce the variable `e*` that will hold the characters that did not match the current position in the pattern. Since we always move forward on the pattern, it is not necessary to save `e*` for the positions in the pattern that have been passed.[2]

This will avoid repeated matching of the same value against a dynamic variable – instead it will be matched against static values. This means that the comparison can be done at specialisation time.

This program is shown in fig. 5.8.

When this program is program specialised, it gives residual programs with the structure of a substring tree. The result for `(a b a b c)` is shown in fig. 5.9.

With data specialisation, however, the results are still not satisfactory. The residual program from data specialisation of 5.8 is shown in fig. 5.10.

Another problem common to program and data specialisation is that a substring tree is a rather large structure, that in fact contains many redundancies. So although the result of program specialisation is linear in time, its space complexity is less satisfactory.

5.5.3 How can we get small, linear residual results?

Our problem is that the partial evaluator generates an instance of a procedure for each static value, in this case the values of `i` and `j` that indicate the match so far, even though the generated procedures actually performs the same actions (*i.e.*, implement the same function).

If we want to generate programs corresponding to the folded graph instead, we must thus substitute the actual values of the match-so-far for some dummy values, which are identical for calls to procedures that we want to have only one residual occurrence of.

This is in principle a simple solution but it raises two questions:

- Can we do it? *I.e.*, is it possible to find some dummy values for the match-so-far that gives the right folding but do not interfere with the original purpose of the variable.
- How do we do it. *I.e.*, how do we recognize, in the quadratic algorithm, having only the subject string to work on, that two calls to a procedure should have their match-so-far parameters

[2] It is possible to write a program that saves compared characters for each position in a list isomorphic to the subject. Such a program would not specialise well, since this list is built under dynamic control so unless it is classified dynamic, the specialisation will loop, trying to build an infinite list. If the list is classified dynamic, however, the memorising effect is lost.

```
(define (match? sub pat)
    (find-match sub 0 (length sub) pat 0 (length pat)))
(define (find-match sub i n pat j m)
    (if (= j m)
        #t
        (find-match-aux sub i n pat j m ())))
(define (find-match-aux sub i n pat j m e*)
    (if (>= (+ i j) n)
        #f
        (find-match-aux-2 sub i n pat j m e*)))
(define (find-match-aux-2 sub i n pat j m e*)
    (if (member (list-ref sub (+ i j)) e*)
        (find-match-aux sub (pos-next-match sub (add1 i) n 0 i j) n pat 0 m e*)
        (if (equal? (list-ref sub (+ i j)) (list-ref pat j))
            (find-match sub i n pat (add1 j) m)
            (find-match-aux sub (pos-next-match sub (add1 i) n 0 i j) n
                                pat j m (cons (list-ref sub (+ i j)) e*)))))
(define (pos-next-match sub i n j start msf)
    (if (= j msf)
        i
        (pos-next-match-aux sub i n j start msf)))
(define (pos-next-match-aux sub i n j start msf)
    (if (= (+ i j) n)
        i
        (if (equal? (list-ref sub (+ i j)) (list-ref sub (+ start j)))
            (pos-next-match sub i n (add1 j) start msf)
            (pos-next-match-aux sub (add1 i) n 0 start msf))))
```

Figure 5.8: A string matching program saving both positive and negative information.

Figure 5.9: The result of program specialisation of 5.8 with respect to `(a b a b c)`.

```
(define (match? current g pat)
    (if (>= 0 (length pat))
        #t
        (if (find-att '(>= (+ 0 0) (length sub)) (fetch-atts (find-node current g)))
            #f
            (find-match-aux-2
             (find-edge '(find-match-aux-2 sub 0 (length sub) pat 0 (length pat) ())
                        (fetch-edges (find-node current g)))
             g pat (length pat)))))
(define (find-match-aux-2 current g pat m)
    (if (equal? (find-att '(list-ref sub (+ i j)) (fetch-atts (find-node current g)))
                (list-ref pat (find-att 'j (fetch-atts (find-node current g)))))
        (if (>= (find-att '(add1 j) (fetch-atts (find-node current g)))
                m)
            #t
            (if (find-att '(>= (+ i (add1 j)) n) (fetch-atts (find-node current g)))
                #f
                (find-match-aux-2
                 (find-edge '(find-match-aux-2 sub i n pat (add1 j) m ())
                            (fetch-edges (find-node current g)))
                 g pat m)))
        (if (find-att '(>= (+ (pos-next-match sub (add1 i) n 0 i j
                                              (cons (list-ref sub (+ i j)) e*))
                              j)
                           n)
                      (fetch-atts (find-node current g)))
            #f
            (find-match-aux-2
             (find-edge '(find-match-aux-2
                          sub
                          (pos-next-match sub (add1 i) n 0 i j (cons (list-ref sub (+ i j)) e*))
                          n pat j m (cons (list-ref sub (+ i j)) e*))
                        (fetch-edges (find-node current g)))
             g pat m))))
```

Figure 5.10: The result of data specialisation after the static tests have been moved to `pos-next-match`.

Figure 5.11: The residual graph when specialising 5.8 with respect to the static subject `(a b a b c)`.

substituted for dummies, that is, that the calls should specialize to calls to the same residual procedure.

Fortunately both questions can be answered favorably.

In order to see *how* to do it we give a criteria equivalent to the folding criteria of section 5.2.2 but expressed in terms of the string rather than the graph, *i.e.*, a criteria which can be used to generate the folded graph directly from the string rather than by generating the substring tree and then fold it.

Theorem 2 *The vertices v_1 and v_2 in the substring tree for a string ω can be folded iff $T(root, v_1)$ is a suffix of $T(root, v_2)$ or $T(root, v_2)$ is a suffix of $T(root, v_1)$ and the shortest of the strings does not occur in ω except as a suffix of the longest.*

Proof

Let $\omega_1 = T(root, v_1)$ and $\omega_2 = T(root, v_2)$. It is easy to see that if ω_1 only occurs as a suffix of ω_2, then v_1 and v_2 can be folded. Conversely, if v_1 and v_2 are folded, then there are two paths leading to the folded vertex (v_1, v_2). Since any path from (v_1, v_2) to sink (the node with no sons) corresponds to a suffix of ω and the concatenation of any of v_1 and v_2 is also a suffix of ω, it follows that one must be a suffix of the other. Since there are no other paths out of root corresponding to ω_1 and ω_2 than the ones leading to (v_1, v_2), ω_1 and ω_2 must even occur at the same places in ω. □

The string condition for folding corresponds to the relation called *end-equivalence on substrings* in [Blumer et al. 85].

In order to generate folded graphs by program specialisation, we have to find, before every call to every function that takes the dynamic argument, the longest extension of the current match-so-far which extends it at all the occurrences of the match-so-far. *E.g.*, if the match-so-far is equal to or longer than the prefix identifier of its first position it can be extended with the head of ω up to its first position.

Since the extension occurs exactly as often as the actual match-so-far this will not change the semantics of the algorithm. If there is a match starting with the match-so-far, there will also be a match starting with the extended match-so-far, since the match-so-far only occurs as a suffix of the extended match-so-far. If there is no match, there will not be any match using the extended match-so-far, since it is a superstring of the match-so-far.

With the new algorithm procedure calls corresponding to edges into the same vertex in the folded graph will have the same parameters so, assuming that the partial evaluator only generates one residual procedure for each static argument of a procedure, we will generate folded residual programs corresponding to the folded graph.

This algorithm has been implemented in a version that descends on the strings with `car` and `cdr` rather than using indices into the strings. The program is shown in fig. 5.12.

It is worth to notice, though, that although the residual program generated from this algorithm is compact and linear, the algorithm itself is rather inefficient and repeats many computations, so that it is even slower that the initial quadratic algorithm when run directly. This means that the specialisation of it is likely to be very slow, too.

5.6 How to "Specialise Well"

The rewriting techniques used in this chapter are in fact surprisingly simple when the underlying principles are taken into account. They are centered around the idea of moving computation from

```
(define (substring? pat sub)
    (find-match pat sub () () sub))

(define (find-match pat sub msf e* osub)
    (if (null? pat)
        #t
        (find-match-to pat sub msf e* osub)))

(define (find-match-to pat sub msf e* osub)
    (if (null? sub)
        #f
        (if (member (car sub) e*)
            (if (null? msf)
                (find-match-to pat (cdr sub) msf e* osub)
                (find-match-s msf pat (cdr (append msf sub)) () e* osub))
            (if (equal? (car pat) (car sub))
                (find-match (cdr pat) (cdr sub)
                            (iso-ext (snoc (car sub) msf)
                                     (reverse (remove-suffix osub (append msf sub)))
                                     (cdr (append msf sub))
                                     ())
                            () osub)
                (if (null? msf)
                    (find-match-to pat (cdr sub) msf (cons (car sub) e*) osub)
                    (find-match-s msf pat (append (cdr msf) sub)
                                  () (cons (car sub) e*) osub))))))

(define (find-match-s s-pat pat sub msf e* osub)
    (if (null? s-pat)
        (determine-rest-match pat sub msf e* osub)
        (if (null? sub)
            #f
            (if (equal? (car s-pat) (car sub))
                (find-match-s (cdr s-pat) pat (cdr sub) (snoc (car s-pat) msf) e* osub)
                (find-match-s (append msf s-pat) pat (cdr (append msf sub)) () e* osub)))))

(define (determine-rest-match pat sub msf e* osub)
    (if (null? sub)
        #f
        (if (member (car sub) e*)
            (find-match-s msf pat (cdr (append msf sub)) () e* osub)
            (if (equal? (car pat) (car sub))
                (find-match (cdr pat) (cdr sub)
                            (iso-ext (snoc (car sub) msf)
                                     (reverse (remove-suffix osub (append msf sub)))
                                     (cdr (append msf sub))
                                     ())
                            () osub)
                (find-match-s msf pat (append (cdr msf) sub) () (cons (car sub) e*) osub)))))
```

Figure 5.12: A matching program extending the match-so-far.

```
(define (snoc x l) (append l (cons x ())))

(define (remove-suffix x y)
    (if (equal? x y)
        ()
        (cons (car x) (remove-suffix (cdr x) y))))

(define (remove-suffix string suffix)
    (rs-loop string suffix ()))

(define (rs-loop string suffix a)
    (if (equal? string suffix)
        (reverse a)
        (rs-loop (cdr string) suffix (cons (car string) a))))

(define (iso-ext msf pref-r suff akk)
    (if (null? pref-r)
        (append akk msf)
        (let ((ext-akk (cons (car pref-r) akk)))
            (if (isomorphic? ext-akk msf suff)
                (iso-ext msf (cdr pref-r) suff ext-akk)
                (append akk msf)))))

(define (isomorphic? s-pref s-suff string)
    (if (null? string)
        #t
        (let* ((suffix (matching-suffix s-suff string))
               (prefix (remove-suffix string suffix)))
            (or (null? suffix)
                (and (suffix? s-pref prefix)
                     (isomorphic? s-pref s-suff (cdr suffix)))))))

; s not empty
(define (suffix? s string)
    (and (not (null? string))
         (or (equal? s string)
             (suffix? s (cdr string)))))

(define (prefix? s string)
    (or (null? s)
        (and (not (null? string))
             (equal? (car s) (car string))
             (prefix? (cdr s) (cdr string)))))

(define (matching-suffix pat string)
    (if (or (null? string) (prefix? pat string))
        string
        (matching-suffix pat (cdr string))))
```

Figure 5.13: Auxiliaries for the matching program extending the match-so-far

Figure 5.14: Program specialising the extending program 5.12 with respect to `(a b a b c)`.

run-time to specialisation time. This can also be expressed as moving computation from dynamic arguments to static arguments. In the present context, this amounts to two kinds of transformations:

- Introducing loops equivalent to the original main loop, but comparing the static string against itself.
- Saving the unmatched values of the static string for the current position in the dynamic string.

Although it is not hard to generalise these rewriting as principles for other tests than equivalence, we have seen that the second technique is not unproblematic, since it introduces a new static parameter that will typically take many different values. This is very likely to cause an explosion in the size of the result.

5.7 Extending the Specialiser

As mentioned in section 5.3.3, it would also be possible to obtain results like the present by using a different program specialiser. The idea would be to save the information obtained at the conditionals and use it in the later computation. Particularly in the case of dynamic pattern, it seems that this would be useful, since it would not introduce any new static variables thus avoiding the problem of distinguishing otherwise indistinguishable program points.

A system handling this kind of information has been designed in [Haraldsson 77].

It is expectable, however, that the management of such a system easily becomes quite complicated, as [Haraldsson 77] also points out.

Such a system was also advocated in [Futamura & Nogi 88].

5.8 Conclusion

The results of specialising the rewritten algorithms are surprisingly close to known algorithms and data-structures such as [Knuth, Morris & Pratt 77]. This is probably not accidental but rather a sign that the best known hand-written algorithms have been written by taking the maximal advantage of the fact that some data are known at an earlier state, in order to minimise the later run-time. This is of course exactly what we would like our specialisers to do.

The ability to obtain well-known algorithms by automatic specialising is interesting because of the possibilities it opens for development of new algorithms.

It is, however, uncertain whether it will have any practical use, because although the resulting algorithms are reasonably good, the generation of residual programs and data (*i.e.* the polyvariant specialisation) is rather slow compared to the hand-written algorithms. There has not been given any complexity for the various specialisation algorithms but is seems obvious that it must at least be dependent on the complexity of the source program on the static data. This is very far from the linear bounds on the generation of prefix trees or failure tables found in the literature.

Another interest of the automatic generation is that it shows a relation between the naive algorithms and the well-known optimised algorithms. Instead of being random examples of particular cleverness it is seen that the structures used as intermediate data are in fact natural consequences of the problem. This view can also be obtained by a mathematical access to the relations between strings, but here it is done without any particular assumptions about the strings, only from an analysis of the binding time information.

Chapter 6

Sequential Decomposition

6.1 Introduction

This chapter defines the concept of sequential decomposition [Barzdins & Bulyonkov 88] and shows how it can be used for compilation and how to obtain a sequential decomposition program from a data specialisation program by partial evaluation.

6.2 Definitions

Definition 14 *A* **sequential decomposition** *for a programming language L is a pair of algorithms $[SD_1, SD_2]$ so that for any program l in L taking a pair $[d_1, d_2]$ as input* [1]

$$L\ (M\ SD_2\, l)\ [(L\ (M\ SD_1\, l)\ d_1), d_2] = L\, l\ [d_1, d_2] \tag{6.1}$$

This is perhaps better illustrated graphically:

There is a trivial sequential decomposition algorithm: $SD_1 = \lambda\, x.\, x$, $SD_2 = \lambda\, x.\, x$.

The immediate motivation for doing non-trivial sequential decomposition is to separate those parts of the computation that only depend on d_1 from those parts that depend on both data. This is of interest in cases where the data arrive at different times or the computation will be repeated with a constant value of d_1 and varying d_2. These are the same interests as for specialisation.

The difference from specialisation is that where specialisation aims to *perform* the actions that only depend on the static data, sequential decomposition aims to generate a program that will perform those actions when it is run.

[1] Notice that in this chapter we assume that the source language is the same as the target language and that the source program have exactly two arguments. This is merely for simplicity, it would be quite straightforward to generalise.

By repeated applications, the technique can be used for data with more binding times, *e.g*, for three arguments with different binding times:

6.3 Sequential Decomposition by Data Specialisation

[Barzdins & Bulyonkov 88] shows how to obtain a sequential decomposition algorithm from a data specialisation algorithm.

It is clear that there are similarities between the two kinds of transformations. Particularly it is easy to see that DS_p could be used as SD_2, since it generates a program that expects a pre-processed representation of the static data and the dynamic data in order to produce the original result.

But in order to use DS_p as SD_2 we need to find an algorithm, that takes a source program and produces a residual program that, when it is given the static data, will produce the same result as DS_d produces, given both the program and the static data.

Such a residual program can be obtained by we program specialising DS_d with respect to the source program. But we do not want to the algorithm to take DS_d as an argument, only the source program. But this is exactly what we get if we self-apply the program specialiser with DS_d as the second argument.

Thus $[(M\ PE\ [PE, DS_d]), DS_p]$ is a sequential decomposition algorithm.

To express this in terms of the equations, we see that if we want DS_p to act as SD_2, then we want to find SD_1, so that

$$L\ (M\ DS_p\ l)\ [(L\ (M\ SD_1\ l)\ d_1), d_2] = L\ l\ [d_1, d_2]$$

Since $[DS_d, DS_p]$ is a data specialisation algorithm, we know that

$$L\ l\ [d_1, d_2] = M\ (M\ DS_p\ l)\ [(M\ DS_d\ [l, d_1]), d_2]$$

So we can use a SD_1 which fulfills

$$(L\ (M\ SD_1\ l)\ d_1) = (M\ DS_d\ [l, d_1])$$

This is fulfilled by $(M\ PE\ [PE, DS_d])$, since

$$\begin{aligned} L\ (M\ (M\ PE\ [PE, DS_d])\ l)\ d_1 &= \text{(by definition 5)} \\ L\ (M\ PE, [DS_d, l])\ d_1 &= \text{(do.)} \\ L\ DS_d\ [l\ d_1] & \end{aligned}$$

This is not surprising, given the charaterisation of the different goal of sequential decomposition and specialisation from above. If we say that the sequential algorithms aims to generate an algorithm that takes the static data and does what a specialiser would do on the program and the static data, this is exactly the definition of what program specialisation gives.

6.4 Sequential Decomposition for Multi-Phase Compiling

In the generation of compilers from interpreters or interpretive specifications, one of the main obstacles to obtaining realistic results seems to be that these automatic compilers are one-pass compilers, whereas modern hand-written compilers are multi-pass compilers.

Sequential decomposition seems to offer a possible solution to this design problem.

Abstractly, the structure of a multi-pass compiler can be described in the following way.

The compiler consists of a sequence n algorithms $[comp_1, \ldots, comp_n]$. Two adjacent algorithms $comp_i$ and $comp_{i+1}$ communicate via an intermediate language, C_i, that is the source language of $comp_{i+1}$ and the target language of $comp_i$. In particular C_0 is the source language of the whole compiler and C_n the target language. $comp_i$ then takes C_{i-1} programs and compile then into C_i programs.

E.g., if the first pass is a lexical analyser, C_0 could be a language of strings of characters and C_1 could be a language of strings of tokens.

We want to generate an n-pass compiler automatically from an interpreter int^{C_0} for the source language C_0. The target language C_{target} is given. We know that by program specialisation we are able to generate a compiler from C_0 to C_{target} from int^{C_0}. What we want is rather to generate a sequence of simpler (or dedicated) compilers between a series of intermediate languages that are not given.

This can be done if we can find a sequence of algorithms $A_1, \ldots, A_n$ so that

- Each A_i for $i = 1, \ldots, n-1$ takes an interpreter $int^{C_{i-1}}$ for a language C_{i-1} written in L and returns a compiler from C_{i-1} to another language C_i written in L *and* an interpreter int^{C_i} for the target language of the compiler also written in L.

- A_n takes an interpreter $int^{C_{n-1}}$ and returns a compiler from C_{n-1} to a fixed target language $C_n = C_{target}$.

Notice that the intermediate languages are *not* fixed but generated on the way. In fact the interpreters are only generated to specify the intermediate language, *i.e.*, to be given to the next A_i.

It should also be noticed that the As do not have to be different.

We immediately see that A_n can be *cogen* (*cf.* 3.3).

For the other As, we see that sequential decomposition fulfills the criteria. Assume that we have a sequential decomposition algorithm $[SD_1, SD_2]$ and an interpreter int^{C_i}. Then $comp = M\ SD_1\ int^{C_i}$ is a compiler from C_i to the language defined by the interpreter $int = M\ SD_2\ int^{C_i}$ since

$$L\ int\ [(L\ comp\ p), d] = C_i\ p\ d$$

which is exactly the definition of a compiler (chapter 3, definition 4), except that here the target language is only given by an interpreter[2].

[2] If we assume that we have a curry operator, however, $Curry\ (L\ int)$ is in fact the language defined by int

It is interesting to notice that $A_i = [SD_1, SD_2] = [(M\,PE\,[PE, DS_d]), DS_p]$ and $A_n = cogen = M\,PE\,[PE, PE]$ corresponds with the view of program specialisation as a special case of data specialisation with DS_p generating a constant interpreter and DS_d being the program specialiser.

In this application, the repeated use of sequential decomposition does not correspond to the arrival of more and more information. The effect that is sought is rather to decompose a compound action on the same data into independent phases. To actually obtain this it is necessary to be able to distinguish independent actions in a composed action, which is a non-trivial problem.

Chapter 7

Applications of Sequential Decomposition

7.1 Introduction

After the previous theoretical chapter, this chapter discusses some of the problems of obtaining non-trivial results when program specialising programs based on the algorithm from section 4.5.3. The problems are basically the same as those obtained when self-applying a partial evaluator.

Then one of the string matching algorithms from chapter 5 is used for a small example of sequential decomposition and the possibility of using the algorithm from section 4.5.3 for compiler generation is discussed.

The importance of the design of the residual data structures is pointed out and some data specialisation algorithms designed specially for compiler generation are proposed.

7.2 Partially Evaluating the Data Specialisation Algorithm

Although it is theoretically possible to obtain a sequential decomposition algorithm from a partial evaluator and a data specialisation algorithm, this result does not say anything about the quality of the algorithm, from a computational point of view.

7.2.1 Problems

When specialising the graph generator we encounter the same problems as when self-applying a partial evaluator, since the graph specialiser is structured the same way as a polyvariant partial evaluator.

Thus it is also possible to use the same techniques to solve the problems.

One of the main problems in obtaining non-trivial results when specialising interpreters or self-applying partial evaluators is that if a procedure takes both static and dynamic arguments, then its result will be classified dynamic by the binding time analysis. This happens even if some part of the result or even the whole result is based on static data. To be able to use such information at specialisation time we do not return from such procedures, but continue the computation with tail-recursive calls where the "result" of the procedure is passed as a static argument in the call.

This is the reason why, *e.g.*, the lookup-loops finding procedure definitions in the program are tail-recursive in the programs shown in appendix C.

The problem can also be treated by introducing partially static structures rather than the coarse division into either static or dynamic [Mogensen 89].

It is worth to notice that since the pattern matching algorithms shown in chapter 5 are already tail-recursive, this problem does not arise in that example.

Another serious problem of specialising a polyvariant specialiser is that the list of active states ("pending" in the Mix terminology) is dynamic, because the static values are dynamic at specialisation of the specialiser. This means that the name of the next procedure to specialise is dynamic, so the lookup finding the next expression to specialise returns a dynamic result, *i.e.*, the expression to be specialised is dynamic. This is unfortunate, since the program to be specialised is the static input of the specialiser. If it is not possible to access this program statically, very little can be done at specialisation time.

It is not sufficient to separate the list of active states into two isomorphic lists, the procedure names and the values. The problem here is that the list of active states, and thus also a corresponding list of procedure names, is unbounded at specialisation time. It is generated under dynamic control; its generation depends on the static values of the source program, that are dynamic at the specialisation of the specialiser. Making a static list of procedure names isomorphic to the "active" list would thus result in infinite specialisation.

Instead this problem is circumvented by not using the name fetched from the list of active states to look up the body of the procedure in the source program. Instead this name is compared against a static list containing all the procedure names. Then the name from that list is used in the lookup (this technique is from the Mix system. For a more detailed description see [Gomard & Jones 88]). When program specialising the specialiser, this gives a dispatch on procedure names in the residual program, with the treatment corresponding to the procedure body at each entry.

As an extra optimisation, only the procedures that are called residually in the source program are put into the list of procedure names. This way there will only be code to generate those procedures that actually occur in the residual program.

7.3 Sequential Decomposition of a String Matching Algorithm

For a more straightforward application, let us first consider one of the string matching algorithms from chapter 5, *e.g.*, the one for static pattern strings given in fig. 5.2.

By using the Similix partial evaluator on our data generating program from section 4.5.3, we obtain a sequential decomposition algorithm (this is more carefully discussed in chapter 7), that we can apply to our matching algorithm. We then obtain a program that takes a pattern string and generates a failure table. This is exactly the same program as we would obtain by specialising the data generating program with respect to the string matching algorithm, since:

$$\begin{array}{rcl} M\,SD_1\,p_{match} & = & \text{(with our method)} \\ M\,(M\,PE\,[PE, DS_d])\,p_{match} & = & \text{(by definition 5)} \\ M\,PE\,[DS_d, p_{match}] & & \end{array}$$

The resulting, failure-table generating, program is rather large, so we will not show it. Instead, let us briefly outline its structure.

It is in fact a specialised version of our data generator, so it has the same general structure as this program. It consists of a main loop over a list of active states. Each state consists of a label, that can be either `match?` or `find-match-aux`, and a list of values, either the static string or the static string, its length and the current position. Only the initial state has the label `match?`, all the states that are generated in the loop are given the label `find-match-aux`. Of the static values, only the position varies, and this is bounded by zero and the length of the string. Thus there are only $m + 2$ different states.

Since the failure table generator generates a new node in the graph every time it encounters a new state it will generate $m + 2$ nodes. When it generates an already encountered state, it will generate an edge to this state, but *not* add it to the list of active states, just like the data generator. For the first state it generates one node and one edge. For the last state, it has no actions. For each state in the list of active states, except the first and last, it generates two new states, one by adding one to the position and one by computing the value of the failure function on the position. For the first state it generates one new state and for the last none. Thus it will generate $2m + 1$ edges.

The run-time of the failure table generator, however, is *not* bounded by the length of the pattern. The problem is that the failure function is computed using the quadratic algorithm inherited from the string matching program, where it is used to match the static string against itself. Thus the failure table generator as a whole is not linear in the pattern, although the main loop is.

The resulting failure tables, however, can be used for linear matching, since they will be used with the residual program obtained by the program generating data specialisation algorithm. We have already seen that this program is linear.

7.4 Program Specialising the Data Specialisation Program

The actual sequential decomposition algorithm obtained from the graph generating algorithm is also a quite large program. It is structured like the Similix partial evaluator and thus harder to comprehend. Like the data specialisation algorithms, Similix loops over a list of active states (here called *pending*) and this loop can be recognized in the sequential decomposition algorithm. The states have names from the data specialisation algorithm: `mc-loop`, `f&t-loop`, `treat-cond` (2 versions), `f&ms-loop`, `ds-eval`. For each of these names there is a specialised version of Similix's actions on the corresponding procedure body. Thus the treatment of a state will generate a specialised version of the data specialisation procedure with respect to the list of static values in the state. Furthermore it will add new states to "pending" corresponding to the residual procedure calls in this source procedure.

The target programs of this sequential decomposition algorithm will basically look like the string matching program from the preceding section. They will loop over "active" and generate nodes corresponding to the residual program points and edges corresponding to the calls.

7.5 Sequential Decomposition for Compiler Generation

In order to generate a realistic compiler, we need to design a series of data specialisers that generate, as residual data, the series of intermediary data that a realistic compiler uses. The graph generated by Barzdins's and Bulyonkov's algorithm does not immediately fit this description. Although the residual graph of an interpreter and a program is likely to resemble the structure of the program (depending on the construction of the interpreter) it does not resemble, *e.g.*, a syntax tree, not even in the structure since it contains cycles.

The problem seems to be that this data specialiser is in fact not particularly written to generate compilers. This shows in the fact that the residual data is not designed to be a typical compiler data structure – it does not "look like" target code.

Designing a series of data specialisers for compiler generation is an open problem that we will discuss informally in the next section.

7.6 Towards a Compiler-Based Design of Residual Data Structures

For compiler generation we want a series of data specialisers generating residual data resembling the internal structures of a compiler. For our language of recursive equations, this could for example be syntax trees, a kind of tree-structured code and a linearised code as the final result.

Let us consider the possibility of generating syntax trees as residual data.

Generating syntax-trees

We consider the generation of a syntax tree from a string of tokens as an example of a compiler pass.

In order to obtain such a pass we have to design a data specialisation algorithm that, given an source interpreter and a string, generates a syntax tree and a corresponding target interpreter. The source interpreter takes strings and input and interprets the string as a program to be run on the input. The target interpreter should not contain the actions used in the source interpreter to recognise syntax. Instead it should contain dispatches on syntax non-terminals. What the data specialiser does for any other kind of program is not important.

How can we isolate the syntax recognising actions in the source interpreter? It seems reasonable to say that we cannot in general. Somehow we will have to manage with a simpler criterion that is likely to include the actions we look for and which gives the desired kind of output.

How can we get the desired output, *i.e.*, syntax trees and corresponding target interpreters? For the residual program it is simple. We want this program to be an interpreter dispatching on syntax labels, so we simply move closer to the program specialisation case. Instead of basing the residual program on the source program we always generate an interpreter dispatching on syntax. The residual actions (*i.e.*, those depending on dynamic data) of the source program are then fitted into this interpreter skeleton. Now we just have to decide how to make a suitable syntax.

This is in fact the tricky part. An obvious decision is that the values of the constant expressions should appear as terminals. Correspondingly a symbol corresponding to each static expression should appear as a non-terminal. *E.g.*, in specialising one of the string matching algorithms with respect to the static pattern `(a b a)` we would have the production:

$$\mathit{symbol}_{\texttt{(list-ref pat j)}} ::= \texttt{a} \mid \texttt{b}$$

This is not enough, however, we need productions with non-terminals at the right-hand side to get one syntax tree as residual data. The possible sources of such non-terminals are the non-terminals of the source language (the language the source programs are written in) and the program points of the source program. Using the non-terminals of the source language will reproduce pieces of the source program in the residual data. Since we rather want such pieces in the residual program the second option seems more promising.

If we make non-terminals out of source program points we can see the recursive equations that make up our source program as part of our syntax instead. We re-interpret all the terms to a syntax format. To this we add the static expression non-terminals as mentioned above.

The simpler dynamic expressions can be made into non-terminals if they involve static subexpressions and terminals if they do not. The code interpreting them is then put in the target interpreter under the treatment of this syntactic form.

The conditional expressions with static test but dynamic branches correspond to alternatives on the right-hand side (*true-branch* | *false-branch*). The dynamic conditionals are simply conditionals, *i.e.*, a sequence of three non-terminals.

This gives a reasonable syntax. Unfortunately we cannot generate residual data unless all calls in the source program are unfoldable. If a call is residual, the procedure might be called infinitely often, so the syntax tree is infinite.

Instead we can leave the residual procedures (the procedures that are called residually) in the target program as auxiliaries to the interpreter loop (such as "apply" in a hand-written interpreter). The production of the non-terminal corresponding to a residual call has the list of arguments as right-hand side rather than the syntax corresponding to the procedure body. The residual syntax tree contains one subtree for each residual program point (resembling the declaration of residual procedures in program specialisation). The procedure in the residual interpreter calls the interpreter to evaluate the arguments, searches the residual syntax tree for the subtree with the right "name" (pattern of static arguments) and calls the interpreter loop with this tree.

This algorithm has not been implemented but on paper it looks like a step towards our goal of compiler generation. It does seem too coarse, though, the generated syntax will involve the whole interpreter, not just the syntax recognizing parts. It seems that the ability to have good results with this algorithm still relies on the ability to write source programs that "specialise well" and have the right kind of division of tasks into procedures.

To generate a kind of target code we might let us inspire by the hand transformation techniques used to derive compilers from interpreters (see chapter 8). The point here is to try to identify residual operations in the source program and generate, as residual data, a graph where the nodes are corresponds to residual operations and the edges are pointers to the next residual operation in the computation tree (*i.e.*, state transitions).

A likely design would be to generate a simple, but tree-structured, code and then linearise it. Since the last phase is program specialisation based to end at a fixed target language we should probably linearise in this phase. Otherwise there is no particular reason that the linear code we generate does not have to be permuted to translate into the final target language, thus reversing the computations we just made. This will require closer examinations, both theoretical and practical.

To actually combine these algorithms to give a compiler might still give rise to unexpected problems. It is possible to foresee at least one problem from the composition: we can hand-tune a handwritten program to specialise well but if we want realistic compilers out of this method, the successive transformations must preserve that property in the residual program.

To design and implement a full set of data specialisation algorithms seems a large task and it might be worth to consider whether this is actually simpler than using the existing compiler technology. The crucial point here is whether the same intermediary structures are adequate in compilers for many different languages and whether the same source language is adequate for writing our source interpreters for many different languages.

Chapter 8

Related Work

The related works in mixed computation have been mentioned in chapters 2, 3 and 4.

The string matching algorithms in chapter 5 were developed to resemble the algorithms from [Knuth, Morris & Pratt 77], [Weiner 73] and [Blumer et al. 85]. This continues the work of [Consel & Danvy 89] and has partly been developed in cooperation with these authors. To our knowledge automatic specialisation has not before been used to generate basic pattern matching algorithms. [Emanuelson 80], [Emanuelson 82] has used program specialisation for pattern matching in more complicated matchers.

8.1 Hand Transformations from Interpreter to Compiler

Many authors discuss techniques for transforming language specifications into executable compilers by hand. These techniques can be seen as integrations into one phase of specialisation and the techniques necessary to "specialise well".

[Jørring & Scherlis 86] stress the concept of binding time (though under a different name) and point out that when the arguments of programs have different binding times, many optimisations are made by shifting computations to earlier binding times. This concept is called *staging transformation*. The main example throughout the article is compiler generation. Two approaches are discussed: pass separation and meta-transformation.

In the first case, the specification of the interpreter is divided into two parts by starting with the trivial case where the compiler is just identity and the moving as many actions as possible into the compiler by observing which actions are independent of the unavailable data. The two parts are then separated and the remaining actions then comprise the target interpreter.

In the case of meta-transformations the interpreter is transformed by "freezing" parts of the actions and returning them as result rather than executing them. In this case the target language is the language of the original specification.

It is interesting to notice that these different approaches can both be viewed as data specialisation. Meta-transformation clearly resembles the actions of a partial evaluator on its source programs. The transformations of source programs to make such actions efficient correspond to the transformations employed in partial evaluation to make programs "specialise well".

Pass separation, on the other hand, resembles sequential decomposition. The goal is, given a function $f(x,y)$, to find to functions f_1, f_2, so that $f(x,y) = f_2(f_1(x), y)$ by starting with the identity function as f_1. In fact the derivations show how to derive a non-trivial f_1 from an interpreter by hand.

[Jørring & Scherlis 86] also stress the importance of choosing the right data structure as "object code" between the two phases.

[Wand 82], [Wand 82'] [Wand 86] treats the derivation of compilers from interpretive specifications in the form of Scheme interpreters written in a style resembling denotational semantics. The techniques are gradually simplified and improved to give better code.

The idea is to re-organise the semantics to use a small number of compound actions that can then be abstracted as combinators. A compiler is then obtained by generating combinator sequences rather than executing the action. The target machine interprets the sequences of combinator that has been derived from the semantics.

[Montenyohl & Wand 88] describes a technique for obtaining a compiler with type-checking. The idea is to rewrite the semantics to express the type-checking in terms of the compile time information and shift the type-checking actions to be performed before the run time actions. The resulting definition can then be submitted to the transformation techniques of [Wand 83]. This results in a compiler performing type-checking at compile time. All transformations are proven to be semantics preserving.

In fact, it seems that everyone makes the same transformations. In automatic transformation we have the binding time annotations to guide us. In hand transformations, many of the transformations seem to arrive out of thin air (although they are still guided by binding time information).

Chapter 9

Further Work

Although the data specialisation algorithm discussed here is less powerful than the polyvariant program specialisation algorithm, the benefit of a more compact representation of the residual output still justifies further work on this approach.

One obvious line of work would be to design an algorithm able to handle unfolding and conditionals more efficiently than ours. The syntax-tree generating algorithm of chapter 7 might solve this problem and it would probably be worth to polish this to an implementable description and implement a version of the algorithm.

A more ambitious plan would be also to design and implement the remaining algorithms for a compiler.

Another direction is to extend our algorithm to handle other language features, such as higher-order procedures or even side-effects, in order to treat, *e.g.* full Scheme, or structured loops and block structures for imperative languages.

Before the present work could be used for any practical purposes it would also be necessary to implement a more efficient treatment of the residual data.

There is also much to be done about a semantic foundation for data specialisation.

In order to be able to separate static and dynamic computation it might be possible to use some of the techniques employed when parallelising sequential programs. This would require a clarification of the relation between independence of computation as we want it for specialising well and independence of computation as it is used to parallelise sequential programs.

Chapter 10

Conclusion

The string matching examples show that it is possible to obtain realistic results by data specialisation.

The various rewritings of the original naive algorithm emphasize that the straightforward algorithms usually refer to their arguments in an "interleaved" way. If we want to obtain good results from specialisation we have to re-organise the source algorithm, so that the static data are accessed first and the static conditionals are completely eliminated. Furthermore the implicit information about dynamic data that can be derived from dynamic conditionals should be exploited.

The string matching examples use mainly the last technique, and these examples suggest to extend the specialiser to take advantage of the implicit information that can be derived during a symbolic evaluation.

Since this is a complicated task with inherent threats of combinatory explosion, it might be more realistic to rewrite the source program in a preprocessing phase. This is, however, not always an ideal solution. This is seen in the string matching with static subject string, where the rewriting introduces a new static variable that causes the specialiser to distinguish program points that we want to be collapsed.

It is possible to separate static and dynamic computation by hand rewriting, by mechanical rewriting or on the fly during specialisation. But we see that although the technique of polyvariant specialisation captures the concept of specialisation very precisely, it is not capable of sorting the threads of an interleaved treatment of static and dynamic data.

Presently this sorting must be done by the programmer, who needs to understand polyvariant specialisation and binding times to have good results.

But it is not necessary to have any understanding of which parts of the program would naturally fit the purpose of the residual program and thus should be abstracted as residual program points. This will arise by itself from the binding time directed rewriting, since such primitives are exactly characterised by being completely dynamic. This is opposed to the hand-transformation techniques, where isolating such parts often appear as the "magic" point in the transformation.

In the case of data specialisation, however, there do not seem to be any similar principles determining the "optimal" structure of the residual data. It should incorporate those parts of the structure and contents of the static data that will be compared against the dynamic data and those parts of the static control that modify dynamic data. Furthermore it should *not* contain those parts of the static data that only determine static control but only the resulting control structure and it should *not* contain those parts of the static control that only modify static data. These are, however, rather general conditions and do not tell us which structure best fulfills them.

Furthermore, since multipass transformations are frequently used in realistic applications, such a structure might best be constructed in several phases. This is possible using sequential decomposition but again we have no directions about the "optimal" (or even adequate) structure after each pass.

Bibliography

[Aho, Hopcroft & Ullman 74] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman: *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974)

[Aho-Ullman:77] Alfred V. Aho & Jeffrey D. Ullman: *Principles of Compiler Design*, Addison-Wesley (1977)

[Barzdins & Bulyonkov 88] G. J. Barzdins & M. A. Bulyonkov: *Mixed Computation and Translation: Linearisation and Decomposition of Compilers.* Preprint 791 from Computing Centre of Sibirian division of USSR Academy of Sciences, p.32, Novosibirsk (1988)

[Barzdins et al. 89] G. J. Barzdins, M. A. Bulyonkov, K. Malmkjær: *Data Specialisation — A Mixed Computation Principle*, working paper (presents material from [Barzdins & Bulyonkov 88] in English) (1989)

[Blumer et al. 85] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, J. Seiferas: *The Smallest Automaton Recognizing the Subwords of a Text*, Theoretical Computer Science, no 40, pp 31-55, North-Holland (1985)

[Bondorf & Danvy 89] Anders Bondorf & Olivier Danvy: *Automatic Autoprojection for Recursive Equations with Global Variables and Abstract Data Types*, working paper, DIKU, Computer Science Department, University of Copenhagen (July 1989)

[Boyer & Moore 77] R. S. Boyer & J. S. Moore: *A Fast String Searching Algorithm*, Communications of the ACM, Vol. 20, pp 762-772 (1977)

[Bulyonkov 84] M. A. Bulyonkov: *Polyvariant mixed computation for analyzer programs* Acta Informatica, Vol. 21, Fasc. 5, pp 473-484 (1984)

[Consel 89] Charles Consel: *Evaluation Partielle, Analyse de Programmes, et Génération de Compilateurs*, PhD thesis, University of Paris VI, Paris, France (June 1989)

[Consel & Danvy 89] Charles Consel, Olivier Danvy: *Partial Evaluation of Pattern Matching in Strings*, Information Processing Letters, Vol. 30, No 1 (January 1989)

[Dybvig 87] R. Kent Dybvig: *The Scheme Programming Language*, Prentice-Hall (1987)

[Emanuelson 80] P. Emanuelson: *Performance Enhancement in a Well-Structured Pattern Matcher Through Partial Evaluation*, Linköping Studies in Science and Technology Dissertations, No 55, Linköping University, Sweden (1980)

[Emanuelson 82] P. Emanuelson: *From Abstract Model to Efficient Compilation of Patterns*, from International Symposium on Programming, 5th Colloquium, Turin, Italy, Lecture Notes in Computer Science, M. Dezani-Ciancaglini and U. Montanari (eds.), Vol. 137, pp 91-104, Springer-Verlag (1982)

[Ershov 77] Andrei P. Ershov: *On the Partial Computation Principle*, Information Processing Letters, Vol. 6, No 2 pp 38-41 (April 1977)

[Ershov 78] Andrei P. Ershov: *On the Essence of Compilation*, in *Formal Description of Programming Concepts* pp 391-420, E. J. Neuhold (ed.), North Holland (1978)

[Ershov 82] Andrei P. Ershov: *Mixed Computation: Potential Applications and Problems for Study*, Theoretical Computer Science No 18 pp 41-67 (1982)

[Futamura 71] Yoshihiko Futamura: *Partial Evaluation of Computation Process – an Approach to a Compiler-Compiler*, Systems, Computers & Control Vol. 2, No 5 pp 45-50 (1971)

[Futamura & Nogi 88] Yoshihiko Futamura & Kenroku Nogi: *Generalized Partial Computation*, from Partial Evaluation and Mixed Computation, Dines Bjørner, Andrei P. Ershov, Neil D. Jones (eds.) Elsevier Science Publishers B.V. (North-Holland) (1988)

[Gomard & Jones 88] Carsten K. Gomard & Neil D. Jones: *Compiler Genertion by Partial Evaluation: A Case Study*, DIKU Report, No 88/24, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark (1988)

[Haraldsson 77] A. Haraldsson: *A Program Manipulation System Based on Partial Evaluation*, Dissertation, Linköping University, Sweden (1977)

[Jones et al. 1985] Neil D. Jones, Peter Sestoft and Harald Søndergaard: *An Experiment in Partial Evaluation: The Generation of a Compiler Generator*, Rewriting Techniques and Applications, Dijon, France, Lecture Notes in Computer Science, J.-P. Jouannaud (ed.), Vol. 202, pp 124-140, Springer-Verlag (1985)

[Kehler & Danvy 89] N. Carsten Kehler Holst & Olivier Danvy: *Partial Evaluation Without a Partial Evaluator*, working paper, DIKU, Computer Science Department, University of Copenhagen (July 1989)

[Jones 88] Neil D. Jones: *Automatic Program Specialization: A Re-examination from Basic Principles*, from Partial Evaluation and Mixed Computation, Dines Bjørner, Andrei P. Ershov, Neil D. Jones (eds.) Elsevier Science Publishers B.V. (North-Holland) (1988)

[Jones et al. 89] N.D. Jones & P. Sestoft & H. Søndergaard: *Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation*, Lisp and Symbolic Computation, Vol. 2, No 1, pp 9-50 (1989)

[Jørring & Scherlis 86] Ulrik Jørring & William L. Scherlis: *Compilers and Staging Transformations*, Proceedings of POPL 86 (1986)

[Knuth, Morris & Pratt 77] Donald E. Knuth, James H. Morris, Vaughan R. Pratt: *Fast Pattern Matching in Strings*, SIAM Journal on Computing, Vol. 6, No 2, pp 323-350 (June 1977)

[Lombardi & Raphael 64] Lionello A. Lombardi, Bertram Raphael: *LISP as the Language for an Incremental Computer*, from *The Programming Language LISP: Its Operation and Applications*, E.C. Berkeley & D.G. Bobrow (eds.), pp 204-219, Information international, Inc., The M.I.T. Press, Cambridge, Massachusetts (1964)

[Lombardi 67] L. A. Lombardi: *Incremental computation*, *Advances in Computers*, Vol. 8, pp 247-333, F. L. Alt & M. Rubinoff (eds.), Academic Press (1967)

[McCarthy 62] John McCarthy: *Towards a Mathematical Science of Computation*, Information Processing 62 (Popplewell, ed), pp 21-28, North-Holland (1962)

[Mogensen 89] Torben Mogensen: *Binding Time Aspects of Partial Evaluation*, PhD thesis, University of Copenhagen, Copenhagen, Denmark (March 1989)

[Montenyohl & Wand 88] Margaret Montenyohl & Mitchell Wand: *Correct Flow Analysis in Continuation Semantics*, Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California (January 1988)

[Rees & Clinger 86] Jonathan Rees, William Clinger (eds.): *Revised^3 Report on the Algorithmic Language Scheme*, SIGPLAN Notices, Vol. 21, No 12, pp 37–79 (December 1986)

[Sestoft 88] Peter Sestoft: *Automatic Call Unfolding in a Partial Evaluator*, from Partial Evaluation and Mixed Computation, Dines Bjørner, Andrei P. Ershov, Neil D. Jones (eds.) Elsevier Science Publishers B.V. (North-Holland) (1988)

[Turchin 74] V. F. Turchin: *Equivalent Transformations of Refal Programs*, from Automatizirovannaya Sistema Upravlenya Stroitelstvom, Vol. VI, pp 36-68, TsNIPIASS, Moscow (1974) (in Russian)

[Turchin 81] Valentin F. Turchin: *Semantic definitions in REFAL and the automatic production of compilers*, Semantics-Directed Compiler Generation, Aarhus, Denmark, Lecture Notes in Computer Science, Neil D. Jones (ed.), Vol. 94, pp 441-474, Springer-Verlag (1981)

[Wand 82] Mitchell Wand: *Deriving Target Code as a Representation of Continuation Semantics*, ACM Transactions on Programming Languages and Systems, Vol. 4, No 3, pp 496-517 (July 1982)

[Wand 82'] Mitchell Wand: *Semantics-Directed Compiler Generation*, Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, (January 1982)

[Wand 83] Mitchell Wand: *Loops in Combinator-Based Compilers*, Information and Control 57, pp 148-164 (May/June 1983)

[Wand 86] Mitchell Wand: *From Interpreter to Compiler: A Representational Derivation*, from Programs as Data Objects, H. Ganzinger & N. D. Jones (eds.), Lecture Notes in Computer Science, Vol. 217, Springer-Verlag (1986)

[Weiner 73] Peter Weiner: *Linear Pattern Matching Algorithms*, IEEE Symposium on Switching and Automata Theory, Vol. 14 pp 1-11, IEEE, New York (1973)

Appendix A

Implementation of the Imperative Algorithm

BNF for the imperative language

```
Pr-ann  ::=  ((De-ann*) In*)
De-ann  ::=  (Id Bt Ex) | (Id Bt)
    Bt  ::=  d | s
    In  ::=  (La Co)
    Co  ::=  LS := Ex | goto La | if Ex goto La | read LS | print Ex | end
    La  ::=  Id
    Ex  ::=  (O1 Ex) | (O2 Ex Ex) | #t | #f | Id | (vector-ref Id Ex) | Number | String
    O1  ::=  add1 | sub1 | length | not
    O2  ::=  + | - | / | * | member
    LS  ::=  Id | (vector-ref Id Ex)
```

A.1 Main parts of the graph generating program

The initialisation procedure takes an annotated program and a list of input to the program. It initialises the graph and state and calls the main loop.

```
(define (mc-1 p-ann i)          ; initialisation
    (let* ((node-0 (new-node))
           (defs (fetch-defs-p p-ann))
           (p (fetch-p p-ann))
           (state (mk-init-state (first-lbl p) defs)))
        (mc-loop (init-graph node-0) (list state) init-passed p
                 (lambda (s)
                     (if (equal? s (fetch-store state))
                         (node-name node-0) 'no-node))
                 i)))
```

The main loop over the list of active states ac:

```
(define (mc-loop G ac pa p node-of i)   ; loop over ac
    (if (null? ac)
        G
        (let ((state (car ac)))
            (if (member state pa)
                (mc-loop G (cdr ac) pa p node-of i)
                (treat-state state G (cdr ac) (cons state pa) p node-of i)))))
```

The treatment of each state have been separated into the procedure `treat-state`. This procedure takes a state and adds the values of all static expressions as attributes to the corresponding node in the graph `G`. Each subsequent state(s) is added to ac and in case of assignment the edge *to* and the node *of* the corresponding new store is added to G. The node corresponding to a store is found using the procedure `node-of`.

`treat-state` dispatches on the type of command and most of the treatment is done by tail-recursive auxiliary procedures.

```
(define (treat-state state G ac pa p node-of i)
    (let* ((lbl (fetch-lbl state))
           (store (fetch-store state))
           (comm (fetch-comm lbl p)))
        (record-case comm
            [goto (to-lbl)
                  (mc-loop G (cons (cons to-lbl store) ac) pa p node-of i)]
            [if (expr goto to-lbl)
                (treat-if expr to-lbl lbl state (node-of store) G ac pa p node-of i)]
            [read (ls)
                (treat-read ls lbl (fetch-next-lbl lbl p) state (node-of store)
                            G ac pa p node-of i)]
            [print (expr)
                (let ((next-lbl (fetch-next-lbl lbl p)))
                    (treat-expr state (node-of store)
                                G (cons (mk-state next-lbl store) ac) pa p node-of i (list expr)))]
            [end ()
                (mc-loop G ac pa p node-of i)]
            [else (if (equal? (cadr comm) ':=)
                      (let ((nxt-lbl (fetch-next-lbl lbl p)))
                          (treat-assign lbl (car comm) (caddr comm) nxt-lbl
                                        state (node-of store)
                                        G ac pa p node-of i))
                      (error 'treat-state "unknown command: ~s." comm))])))
```

`treat-expr` descends into expressions to find and evaluate static sub-expressions. The results are then added as attributes in the graph.

`treat-assign` determines whether the right-hand side is static or dynamic and then either evaluates it and calls `treat-static-update` to perform all the updatings corresponding to a new store or calls `treat-expr` to determine the static subparts of the right-hand side of the assignment.

```
(define (treat-assign lbl l-side r-side nxt-lbl state v G ac pa p node-of i)
    (if (static-expr-s? r-side state)
        (treat-static-update lbl
                             (update-state nxt-lbl l-side (eval-expr r-side state) state)
                             v G ac pa p node-of i)
        (treat-expr state v
                    G (cons (mk-state nxt-lbl (fetch-store state)) ac)
                    pa p node-of i
                    (cons r-side (if (vec-ref? l-side)
                                     (list (fetch-pos l-side))
                                     '())))))
```

treat-if determines whether the test is static or dynamic. For static tests it evaluates the test and adds the corresponding next state to ac. For dynamic tests it adds both subsequent states to active and calls treat-expr to determine the static subparts of the expr.

The rest of the procedures are fairly obvious. Since the imperative version is not written to be partially evaluated it is of a standard design except for the fact that it descends breath-first into the evaluation, using a list of active states, rather than depth-first, with a control stack.

A.2 The program generating program

The program generator simply removes the static variables from the variable list, add the graph and the current node and traverses the list of commands, modifying each as described in section 4.4.

The main procedure:

```
(define (mc-2 p-ann)
    (let ((p (fetch-p p-ann))
          (defs (fetch-defs-p p-ann)))
        (make-program (make-res-defs defs) (make-res-p-init p defs))))
```

The command modifying procedure:

```
(define (make-res-p p defs)
    (if (null? p)
        '()
        (let* ((first (car p))
               (comm (cdr first))
               (lbl (car first))
               (rest (cdr p)))
            (record-case comm
                [goto (to-lbl)
                    (cons first (make-res-p rest defs))]
                [if (expr goto to-lbl)
                    (cons (make-res-if lbl (make-res-expr expr defs) to-lbl)
                          (make-res-p rest defs))]
                [read (ls)
                    (if (static-ls-d? ls defs)
                        (cons (make-res-assign lbl 'node (list 'graph-ref 'G 'node lbl))
                              (make-res-p rest defs))
                        (cons (make-res-read lbl (make-res-ls ls defs))
                              (make-res-p rest defs)))]
                [print (expr)
                    (cons (make-res-print lbl (make-res-expr expr defs))
                          (make-res-p rest defs))]
                [end ()
                    (cons first (make-res-p rest defs))]
                [else (if (equal? (cadr comm) ':=)
                          (if (static-expr-d? (car comm) defs)
        (cons (make-res-assign lbl 'node (list 'graph-ref 'G 'node lbl))
              (make-res-p rest defs))
        (cons (make-res-assign lbl
                               (make-res-ls (car comm) defs)
                               (make-res-expr (caddr comm) defs))
              (make-res-p rest defs)))
    (error 'make-res-p "unknown command: ~s." comm))])))
```

Appendix B

Semantics of the First-Order Recursive Equations

This appendix contains a denotational semantics of the language treated by the data specialiser described in section 4.5.3.

Abstract Syntax

$$
\begin{array}{lcl}
P & ::= & \texttt{(}D\ D^*\texttt{)} \\
D & ::= & \texttt{(define (}F\ I_1\ \ldots\ I_n\texttt{)}\ E\texttt{)} \\
E & ::= & C \mid I \mid \texttt{(if}\ E_1\ E_2\ E_3\texttt{)} \mid \texttt{(}O\ E_1\ \ldots\ E_n\texttt{)} \mid \texttt{(}F\ E_1\ \ldots\ E_n\texttt{)} \\
F & ::= & I \\
O & ::= & \texttt{car} \mid \texttt{cdr} \mid \texttt{cons} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{add1} \mid \texttt{sub1} \mid \texttt{null?} \mid \texttt{equal?} \mid \ldots
\end{array}
$$

Semantic Domains

Num, String, Ide, List, Bool (definition omitted)
Denotable values:
$v \in Val = Num + String + List + Bool$
Environments:
$\rho \in Env = Ide \rightarrow Val$
Operations: $lookup\text{-}\rho\colon Ide \rightarrow Env \rightarrow Val$
$lookup\text{-}\rho = \lambda\, i\, \rho.\, \rho\, i$
Primitive procedures: $\pi \in Prims = Val^* \rightarrow Val$
Operations: $lookup\text{-}\pi\colon O \rightarrow Val{*} \rightarrow Val$
(definition omitted)
Procedures:
$f \in Proc = Val^* \rightarrow Cont \rightarrow Ans$
$\phi \in Procs = Ide \rightarrow Proc$
Operations: $lookup\text{-}\phi\colon F \rightarrow Procs \rightarrow Val^* \rightarrow Cont \rightarrow Ans$
$lookup\text{-}\phi = \lambda [\![F]\!]\, \phi.\, (\phi[\![F]\!])$
$extend\text{-}\phi\colon Ide \times Proc \rightarrow Procs \rightarrow Procs$
Continuations:
$\kappa \in Cont = Val \rightarrow Ans$
Answers:
$v \in Ans = Val_\perp$

Valuation Functions

$\mathcal{P} : P \to \mathit{Env} \to \mathit{Procs}$
$\mathcal{D} : D \to \mathit{Env} \to \mathit{Prims} \to \mathit{Procs} \to \mathit{Ide} \times \mathit{Proc}$
$\mathcal{E} : E \to \mathit{Prims} \to \mathit{Procs} \to \mathit{Expr\text{-}Cont}$
$\mathcal{C} : C \to \mathit{Val}$
$\mathcal{I} : I \to \mathit{Ide}$

$\mathcal{P}[\![\texttt{(}D \;\; D^*\texttt{)}]\!]\rho$
$= \mathit{fix}\,\lambda\,\phi.\,(\mathit{extend\text{-}\phi}\;\;(\mathcal{D}[\![D]\!]\rho\,\pi_{init}\,\phi)\,\mathcal{P}[\![D^*]\!]\rho)$

$\mathcal{D}[\![\texttt{(define (}F \;\; I_1 \;\; \ldots \;\; I_n\texttt{)} \;\; E\texttt{)}]\!]\,\rho\,\pi\,\phi$
$= [[\![F]\!], \lambda\,[v_1 \;\ldots\; v_n].\,\mathcal{E}[\![E_1]\!](\mathit{extend\text{-}\rho}\,[\mathcal{I}[\![I_1]\!], \ldots, \mathcal{I}[\![I_n]\!]]\,[v_1, \ldots, v_n]\,\rho)\,\pi\,\phi]$

$\mathcal{E}[\![C]\!]\rho\,\pi\,\phi\,\kappa = \kappa\,\mathcal{C}[\![C]\!]$
$\mathcal{E}[\![I]\!]\rho\,\pi\,\phi\,\kappa = \kappa\,(\mathit{lookup\text{-}\rho}\,\mathcal{I}[\![I]\!]\,\rho)$
$\mathcal{E}[\![\texttt{(if}\;\; E_1 \;\; E_2 \;\; E_3\texttt{)}]\!]\rho\,\pi\,\phi\,\kappa$
$= \mathcal{E}[\![E_1]\!]\rho\,\pi\,\phi\,(\lambda\,v.\,(\mathit{let}\; v = \;\mathit{Bool}(b)\;\; \mathit{in}\; b \to \mathcal{E}[\![E_2]\!]\rho\,\pi\,\phi\,\kappa \,\|\, \mathcal{E}[\![E_3]\!]\rho\,\pi\,\phi\,\kappa))$
$\mathcal{E}[\![\texttt{(}OE_1 \ldots E_n\texttt{)}]\!]\rho\,\pi\,\phi\,\kappa$
$= \mathcal{E}[\![E_1]\!]\rho\,\pi\,\phi\,(\lambda\,v_1.\;\ldots\;\mathcal{E}[\![E_n]\!]\rho\,\pi\,\phi\,(\lambda\,v_n.\,\kappa\,((\mathit{lookup\text{-}\pi}\;[\![O]\!]\pi)\;[v_1, \ldots, v_n])))$
$\mathcal{E}[\![\texttt{(}FE_1 \ldots E_n\texttt{)}]\!]\rho\,\pi\,\phi\,\kappa$
$= \mathcal{E}[\![E_1]\!]\rho\,\pi\,\phi\,(\lambda\,v_1.\;\ldots\mathcal{E}[\![E_n]\!]\rho\,\pi\,\phi\,(\lambda\,v_n.\,\mathit{lookup\text{-}\phi}\;[\![F]\!]\phi\,[v_1, \ldots, v_n]\,\kappa))$

Appendix C

Implementation of the Algorithm for First-order Recursive Equations

This appendix describes two programs, written for the Similix system [Bondorf & Danvy 89], implementing the data specialisation algorithm from section 4.5.3.

The algorithm has been transformed to the source language of Similix in order to be partially evaluated for chapters 6 and 7.

In the Similix system, it is possible for the user to extend the set of procedures that Similix understands as primitives. This facility has been used here to hide, *e.g* the treatment of the graph. A graph is an abstract data type with operations add-node, add-edge, add-att, *etc.*

The program can be run in Scheme if these procedures are instead defined as Scheme procedures.

Certain Scheme procedures are not available in the current version of Similix. An example is `list`, which is replaced by `list-1`, `list-2`, *etc.* with fixed arities.

C.1 The graph generator

C.1.1 Data structures

The input is an annotated program, the name of the procedure to be specialised and a list of the static data.

The output is a graph, represented as a list of nodes, where each node contains a name, a list of attributes and a list of labeled edges.

The main internal data structures are a list of active state, `ac`, a correspondence between states and node-names, `node-of`, and a list of expressions within the current state, `exprs`. A state is a list of the name of a procedure and the list of static arguments.

C.1.2 The initialisation

The introductory procedure initialises the graph and the list of active states. The variable `f*` is a list of the names of the procedures that will occur in the residual program. It is used to look up the bodies of the procedures in the program.

```
(define (mc-1 p-ann fname s-vals)
    (let ((node-0 (new-node))
          (f* (make-f* p-ann fname))
          (state (list-2 fname s-vals)))
        (mc-loop (init-graph node-0) (list-1 state) p-ann f*
                 (make-node-tbl state (node-name node-0)))))
```

C.1.3 The main loop

The main loop loops over the list of active states. For each state it adds the values of all static expressions except static arguments of procedures to the node in the graph corresponding to that state. It also computes the new states arising from calls to other procedures. If a state has not been seen before, a node is created for it and it is added to the list of active states. Anyway the edge from the node corresponding to the current state to the node corresponding to the new state is generated.

```
(define (mc-loop G ac p f* node-tbl)
    (if (null? ac)
        G
        (let ((state (car ac)))
            (let ((v (node_of state node-tbl))
                  (fname (fetch-fname state))
                  (s-vals (fetch-vals state)))
                (fetch&treat-body fname s-vals v G (cdr ac) p f* node-tbl)))))

; Finds the body of the goal procedure and calls treat-expr to treat it
(define (fetch&treat-body fname s-vals v G ac p f* node-tbl)
    (f&t-loop 0 f* fname s-vals v G ac p f* node-tbl))

(define (f&t-loop index loop-list fname s-vals v G ac p f* node-tbl)
    (if (null? loop-list)
        (error 'fetch&treat-body "unknown procedure: ~s" fname)
        (let ((tag-name (car loop-list)))
            (if (residual-f? tag-name)
                (if (equal? fname (fetch-fname-tag tag-name))
                    (let ((def (fetch-def p index)))
                        (let ((body (fetch-def-body def))
                              (ann-pars (fetch-def-param def)))
                        (let ((s-pars (fetch-s-pars ann-pars)))
                            (treat-expr s-pars s-vals v G ac p f* node-tbl (list-1 body)))))
                    (f&t-loop (add1 index) (cdr loop-list) fname s-vals v G ac p f* node-tbl))
                (f&t-loop (add1 index) (cdr loop-list) fname s-vals v G ac p f* node-tbl)))))
```

The body of the procedure that is to be specialised is found using a loop over the list of possible procedure names. This is done in order to specialise well. Since we want to specialise the program with `ac` dynamic, we cannot use the actual name of the state, since the lookup would then be dynamic.

C.1.4 Treatment of expressions

The procedure `treat-expr` loops over a list of expressions. For each expression it adds attributes for all static subexpressions. If the expression contains a call it evaluates the static arguments and makes the new state. If the new state has not been seen before, it adds it to `ac` and makes a new node and an edge to it in `G`. If it has been seen before, it just makes an edge from the current node to the node of the new state. Most of this is done in auxiliary procedures, the procedure `treat-expr` itself mainly dispatch on the syntactic form of the expression.

```
(define (treat-expr s-pars s-vals v G ac p f* node-tbl exprs)
    (if (null? exprs)
        (mc-loop G ac p f* node-tbl)
        (let ((expr (car exprs)))
            (let ((nt (car expr)))
                (cond
                 ((equal? nt 'quot)      ; (quot (quote s-expr))
                  (treat-expr s-pars s-vals v G ac p f* node-tbl (cdr exprs)))
; corresponds to putting the constant in the graph --
; (add-att (list q s-expr) s-expr v-state G) corresponds to putting the
; constants in the residual program
                 ((equal? nt 'ide)       ; (ide i)
                  (let ((ident (cadr expr)))
                      (if (static-ident? ident s-pars)
                          (let ((att (lookup-s ident s-pars s-vals)))
                              (treat-expr s-pars s-vals v (add-att expr att v G)
                                          ac p f* node-tbl (cdr exprs)))
                          (treat-expr s-pars s-vals v G ac p f* node-tbl (cdr exprs)))))
                 ((equal? nt 'cst)
                  (treat-expr s-pars s-vals v G ac p f* node-tbl (cdr exprs)))
                 ((equal? nt 'prim-op)  ; (prim-op (ann op) args)
                  (let ((op (cadr expr))
                        (args (cddr expr)))
                      (if (static-prim? op)
                          (treat-stat-prim-op expr s-pars s-vals v G ac p f* node-tbl (cdr exprs))
                          (treat-expr s-pars s-vals v G ac p f* node-tbl
                                      (append args (cdr exprs))))))
                 ((equal? nt 'appl); (appl (tag ann fname) args)
                  (let ((fname (cadr expr))
                        (args (cddr expr)))
                      (treat-app expr (car fname) (cadr fname) (caddr fname) args
                                 s-pars s-vals v G ac p f* node-tbl (cdr exprs))))
                 ((equal? nt 'cond) ; (cond if test exp-t exp-f)
                  (let ((test (caddr expr))
                        (exp-t (cadddr expr))
                        (exp-f (car (cddddr expr))))
                      (treat-cond test exp-t exp-f s-pars s-vals v G ac p f* node-tbl
                                  (cdr exprs))))
                 (else (error 'treat-expr "unknown expr: ~s" expr)))))))
```

C.1.5 Treatment of procedure application

The most interesting of the auxiliaries to `treat-expr` is probably `treat-app` that treats the procedure calls.

The actions of this procedure were briefly outlined above. It first test on the annotations to determine whether the call is unfoldable and static, unfoldable and dynamic, or residual. In the case of a static call, it is simply evaluated and the value is added as an attribute to the current node `v`. In the case of an unfoldable dynamic call, the body of the called procedure is found and inserted using the auxiliary `unfold-and-treat`. In the case of a residual call the parameter list of the called procedure is fetched to determine which arguments are static and they are then evaluated to make a new state.

```
(define (treat-app expr tag ann fname args s-pars s-vals v G ac p f* node-tbl exprs)
    (if (equal? tag 'u)                                  ; unfoldable
        (if (equal? ann 's)                              ; even static
            (let ((att (ds-eval expr s-pars s-vals p f*)))
                (treat-expr s-pars s-vals v (add-att expr att v G) ac p f* node-tbl exprs))
            (unfold-and-treat fname args s-pars s-vals v ; just unfoldable
                              G ac p f* node-tbl exprs))
        (fetch&make-state fname args s-pars s-vals v ; residual - make edge
                              G ac p f* node-tbl exprs)))
```

The new state is made in `f&ms-loop`, the loop that also finds the parameter list of the called procedure. It is then tested whether there is a node corresponding to the state. If this is the case, it has been seen before, and `f&ms-loop` calls `old-state-app`, that adds an edge to `G` from the current node to the node of the new state. If it is not the case the state is new and `f&ms-loop` calls `new-state-app`, that adds the state to `ac`, makes a new node, and adds the node and an edge to it to `G`.

```
(define (f&ms-loop index loop-list fname args s-pars s-vals v G ac p f* node-tbl exprs)
    (if (null? loop-list)
        (error 'fetch&make-state "unknown procedure: ~s" fname)
        (let ((tag-name (car loop-list)))
            (if (residual-f? tag-name)
                (if (equal? fname (fetch-fname-tag tag-name))
                    (let* ((def (fetch-def p index))
                           (ann-pars (fetch-def-param def))
                           (dyn-args (d-args args ann-pars))
                           (new-s-vals (make-s-vals ann-pars args s-pars s-vals p f*))
                           (new-state (list-2 fname new-s-vals))
                           (label (cons fname args))
                           (new-exprs (append dyn-args exprs)))
                        (if (no-node? new-state node-tbl)
                            (new-state-app label s-pars s-vals v new-state
                                           G ac p f* node-tbl new-exprs)
                            (old-state-app label s-pars s-vals v new-state
                                           G ac p f* node-tbl new-exprs)))
                    (f&ms-loop (add1 index) (cdr loop-list) fname
                               args s-pars s-vals v G ac p f* node-tbl exprs))
                (f&ms-loop (add1 index) (cdr loop-list) fname
                           args s-pars s-vals v G ac p f* node-tbl exprs)))))

(define (new-state-app label s-pars s-vals v new-state G ac p f* node-tbl exprs)
    (let* ((v-new-state (new-node))
           (new-v-name (node-name v-new-state))
           (new-node-tbl (add-to-node-tbl new-state new-v-name node-tbl))
           (new-edge (make-edge label new-v-name)))
        (treat-expr s-pars s-vals v
                    (add-node v-new-state (add-edge new-edge v G))
                    (cons new-state ac) ; only place ac is increased!
                    p f* new-node-tbl exprs)))

(define (old-state-app label s-pars s-vals v new-state G ac p f* node-tbl exprs)
    (let ((new-edge (make-edge label (node_of new-state node-tbl))))
        ; don't put new-state in ac, since it is already there or treated
        (treat-expr s-pars s-vals v (add-edge new-edge v G) ac p f* node-tbl exprs)))
```

C.2 The Program Generator

The program generator is again relatively simpler, it simply traverses the list of procedures in the program and modifies each one. The static parameters are removed from the parameter list, and the current node and the graph are added.

```
(define (mc-2-loop defs p)
    (if (null? defs)
        '()
        (let ((def (car defs)))
            (make-pgm (make-def (fetch-def-name def)
                                (fetch-dpars def)
                                (make-expr (fetch-def-body def) (fetch-spars def) p))
                      (mc-2-loop (cdr defs) p)))))
```

Then the body is modified using the procedure `make-expr` that dispatch on the syntactic constructions and calls itself to build the new expression. The only slightly tricky point is when it meets an unfoldable procedure call. Then it has to insert the body of the procedure. For this it uses the procedure `beta-reduce` to replace the occurrences of the actual parameters by the arguments and the resulting expression is then transformed.

```
(define (make-expr expr svars p)
    (record-case expr
        (quot (s-expr) (make-quot s-expr))
        (ide (i)       (if (static-ident? i svars)
                           (make-graph-ref expr)
                           (make-i i)))
        (cst (c)       (make-c c))
        (prim-op (op . args)
            (if (static-prim? op)
                (make-graph-ref expr)
                (make-prim-op (cadr op) (make-p-args args svars p))))
        (appl (fname . args)
            (if (res-fname? fname)
                (make-app fname (make-f-args fname args svars p) args)
                (if (static-f? fname)
                    (make-graph-ref expr)
                    (make-expr (beta-reduce (fetch-def-by-name (fetch-name fname) p)
                                            args)
                               svars p))))
        (cond (name test e-t e-f)
            (if (static-cond? test e-t e-f svars)
                (make-graph-ref expr)
                (make-if (make-expr test svars p)
                         (make-expr e-t svars p)
                         (make-expr e-f svars p))))
        (else (error 'make-expr "unknown expression: ~s" expr))))
```

`make-expr` uses `record-case` that is a syntactic extension to Scheme [Dybvig 87].

Appendix D

Specialising String Matching Algorithms

This appendix contains various results of specialising string matching programs. The results are described in chapter 5.

Program specialising the quadratic algorithm with respect to the static pattern `(a b a)` using Similix:

```
(define (match?-0 sub_0)
   (let ((i_3_0_ (generalize 0)))
     (let ((n_4_1_ (length sub_0)))
       (find-match-aux-1 sub_0 i_3_0_ n_4_1_))))
(define (find-match-aux-1 sub_0 i_1 n_2)
  (and (not (= i_1 n_2))
       (if (equal? (list-ref sub_0 i_1) 'a)
           (let ((i_6_0_ (add1 i_1)))
             (and (not (= i_6_0_ n_2))
                  (if (equal? (list-ref sub_0 i_6_0_) 'b)
                      (let ((i_6_1_ (add1 i_6_0_)))
                        (and (not (= i_6_1_ n_2))
                             (if (equal? (list-ref sub_0 i_6_1_) 'a)
                                 (let ((i_6_2_ (add1 i_6_1_))) #t)
                                 (find-match-aux-1
                                  sub_0
                                  (add1 i_6_1_)
                                  n_2))))
                      (find-match-aux-1 sub_0 (add1 i_6_0_) n_2))))
           (find-match-aux-1 sub_0 (add1 i_1) n_2))))
```

Data specialisation of the quadratic algorithm with the same static data:

```
((g93 (((find-match-aux sub (add1 i) n pat 0 m)              g91))
      (((= (add1 j) m) #t)
       ((list-ref pat j) a)))
 (g92 (((find-match-aux sub (add1 i) n pat 0 m)              g91)
       ((find-match-aux sub (add1 i) n pat (add1 j) m) g93))
      (((= (add1 j) m) ())
       ((list-ref pat j) b)))
 (g91 (((find-match-aux sub (add1 i) n pat 0 m)              g91)
       ((find-match-aux sub (add1 i) n pat (add1 j) m) g92))
      (((= (add1 j) m) ())
       ((list-ref pat j) a)))
 (g90 (((find-match-aux sub 0 (length sub) pat 0 (length pat)) g91))
      (((= 0 (length pat)) ()))))
```

And the residual program of data specialisation:

```
(define (match? current g sub)
  (if (find-att '(= 0 (length pat)) (fetch-atts (find-node current g)))
      #t
      (find-match-aux
       (find-edge '(find-match-aux sub 0 (length sub) pat 0 (length pat))
                  (fetch-edges (find-node current g)))
       g sub 0 (length sub))))
(define (find-match current g sub i n)
  (if (find-att '(= j m) (fetch-atts (find-node current g)))
      #t
      (find-match-aux
       (find-edge '(find-match-aux sub i n pat j m)
                  (fetch-edges (find-node current g)))
       g sub i n)))
(define (find-match-aux current g sub i n)
  (if (= i n)
      ()
      (if (equal? (list-ref sub i)
                  (find-att '(list-ref pat j) (fetch-atts (find-node current g))))
          (if (find-att '(= (add1 j) m)
                        (fetch-atts (find-node current g)))
              #t
              (find-match-aux
               (find-edge
                '(find-match-aux sub (add1 i) n pat (add1 j) m)
                (fetch-edges (find-node current g)))
               g sub (add1 i) n))
          (find-match-aux
           (find-edge '(find-match-aux sub (add1 i) n pat 0 m)
                      (fetch-edges (find-node current g)))
           g sub (add1 i) n))))
```

Program specialisation of the quadratic program with respect to the static subject string `(a b a)`.

```
(define (match?-0 pat_0)
   (let ((j_4_0_ (generalize 0)))
      (let ((m_5_1_ (length pat_0)))
         (or (= j_4_0_ m_5_1_)
             (if (equal? 'a (list-ref pat_0 j_4_0_))
                 (let ((j_4_2_ (add1 j_4_0_)))
                    (or (= j_4_2_ m_5_1_)
                        (find-match-aux--0--4 j_4_2_ pat_0 m_5_1_)))
                 (find-match-aux--0--4 0 pat_0 m_5_1_))))))
(define (find-match-aux--0--4 j_0 pat_1 m_2)
   (if (equal? 'b (list-ref pat_1 j_0))
       (let ((j_4_0_ (add1 j_0)))
          (or (= j_4_0_ m_2) (find-match-aux--0--6 j_4_0_ pat_1 m_2)))
       (find-match-aux--0--6 0 pat_1 m_2)))
(define (find-match-aux--0--6 j_0 pat_1 m_2)
   (and (equal? 'a (list-ref pat_1 j_0))
        (let ((j_7_0_ (add1 j_0))) (= j_7_0_ m_2))))
```

And the results of data specialisation for the static subject `(a b a)`.

```
((g108 () (((= i n) #t)))
 (g107 (((find-match-aux sub i n pat j m)           g108))
       ())
 (g106 (((find-match-aux sub (add1 i) n pat 0 m)          g108)
        ((find-match sub (add1 i) n pat (add1 j) m) g107))
       (((list-ref sub i) a)
        ((= i n) ()))))
 (g105 (((find-match-aux sub i n pat j m)           g106))
       ())
 (g104 (((find-match-aux sub (add1 i) n pat 0 m)          g106)
        ((find-match sub (add1 i) n pat (add1 j) m) g105))
       (((list-ref sub i) b)
        ((= i n) ()))))
 (g103 (((find-match-aux sub i n pat j m)           g104))
       ())
 (g102 (((find-match-aux sub (add1 i) n pat 0 m)          g104)
        ((find-match sub (add1 i) n pat (add1 j) m) g103))
       (((list-ref sub i) a)
        ((= i n) ()))))
 (g101 (((find-match-aux sub 0 (length sub) pat 0 (length pat)) g102))
       ()))

(define (match? current g pat)
    (if (= 0 (length pat))
        #t
        (find-match-aux (find-edge '(find-match-aux sub 0 (length sub) pat 0 (length pat))
                                   (fetch-edges (find-node current g)))
                        g pat 0 (length pat))))
(define (find-match current g pat j m)
    (if (= j m)
        #t
        (find-match-aux (find-edge '(find-match-aux sub i n pat j m)
                                   (fetch-edges (find-node current g)))
                        g pat j m)))
(define (find-match-aux current g pat j m)
    (if (find-att '(= i n) (fetch-atts (find-node current g)))
        ()
        (if (equal? (find-att '(list-ref sub i) (fetch-atts (find-node current g)))
                    (list-ref pat j))
            (find-match (find-edge '(find-match sub (add1 i) n pat (add1 j) m)
                                   (fetch-edges (find-node current g)))
                        g pat (add1 j) m)
            (find-match-aux (find-edge '(find-match-aux sub (add1 i) n pat 0 m)
                                       (fetch-edges (find-node current g)))
                            g pat 0 m))))
```

The following pages contains some figures with the actual output of the graph generator. The corresponding graphs are shown in chapter 5.

Figure D.1 gives the residual graph when specialising the program of fig. 5.2 with respect to the pattern string “ababc”.

```
((g420 (((find-match-aux sub i n pat (length-s-match pat 0 1 j) m)        g418))
       (((zero? j) #f)
        ((= (add1 j) m) #t)
        ((list-ref pat j) c)))
 (g419 (((find-match-aux sub i n pat (length-s-match pat 0 1 j) m)        g417)
        ((find-match-aux sub (add1 i) n pat (add1 j) m)                   g420))
       (((zero? j) #f)
        ((= (add1 j) m) #f)
        ((list-ref pat j) b)))
 (g418 (((find-match-aux sub i n pat (length-s-match pat 0 1 j) m)        g416)
        ((find-match-aux sub (add1 i) n pat (add1 j) m)                   g419))
       (((zero? j) #f)
        ((= (add1 j) m) #f)
        ((list-ref pat j) a)))
 (g417 (((find-match-aux sub i n pat (length-s-match pat 0 1 j) m)        g416)
        ((find-match-aux sub (add1 i) n pat (add1 j) m)                   g418))
       (((zero? j) #f)
        ((= (add1 j) m) #f)
        ((list-ref pat j) b)))
 (g416 (((find-match-aux sub (add1 i) n pat 0 m)                          g416)
        ((find-match-aux sub (add1 i) n pat (add1 j) m)                   g417))
       (((zero? j) #t)
        ((= (add1 j) m) #f)
        ((list-ref pat j) a)))
 (g415 (((find-match-aux sub 0 (length sub) pat 0 (length pat))           g416))
       (((= 0 (length pat)) #f))))
```

Figure D.1: The actual output corresponding to figure 5.4.

Figure D.2 gives the residual graph when specialising the program of figure 5.6 with respect to the static subject string “aba”.

Figure D.3 gives the residual program of data specialising the program from figure 5.8 with respect to the static subject string “ababc”.

Figure D.4 gives the residual graph when specialising the program of figure 5.8 with respect to the static subject string “ababc”.

Figure D.5 gives the residual program of program specialising 5.12.

```
((g101 ()
       (((>= (+ (pos-next-match sub (add1 i) n 0 i j) j) n) #t)
        ((>= (+ i (add1 j)) n) #t)
        ((add1 j) 3)
        (j 2)
        ((list-ref sub (+ i j)) a)))
 (g100 ()
       (((>= (+ (pos-next-match sub (add1 i) n 0 i j) j) n) #t)
        ((>= (+ i (add1 j)) n) #t)
        ((add1 j) 1)
        (j 0)
        ((list-ref sub (+ i j)) a)))
 (g99 ()
      (((>= (+ (pos-next-match sub (add1 i) n 0 i j) j) n) #t)
       ((>= (+ i (add1 j)) n) #t)
       ((add1 j) 2)
       (j 1)
       ((list-ref sub (+ i j)) a)))
 (g98 (((find-match-aux-2 sub (pos-next-match sub (add1 i) n 0 i j) n pat j m) g100)
       ((find-match-aux-2 sub i n pat (add1 j) m) g99))
      (((>= (+ (pos-next-match sub (add1 i) n 0 i j) j) n) ())
       ((>= (+ i (add1 j)) n) ())
       ((add1 j) 1)
       (j 0)
       ((list-ref sub (+ i j)) b)))
 (g97 (((find-match-aux-2 sub i n pat (add1 j) m) g101))
      (((>= (+ (pos-next-match sub (add1 i) n 0 i j) j) n) #t)
       ((>= (+ i (add1 j)) n) ())
       ((add1 j) 2)
       (j 1)
       ((list-ref sub (+ i j)) b)))
 (g96 (((find-match-aux-2 sub (pos-next-match sub (add1 i) n 0 i j) n pat j m) g98)
       ((find-match-aux-2 sub i n pat (add1 j) m) g97))
      (((>= (+ (pos-next-match sub (add1 i) n 0 i j) j) n) ())
       ((>= (+ i (add1 j)) n) ())
       ((add1 j) 1)
       (j 0)
       ((list-ref sub (+ i j)) a)))
 (g95 (((find-match-aux-2 sub 0 (length sub) pat 0 (length pat)) g96))
      (((>= (+ 0 0) (length sub)) ()))))
```

Figure D.2: The actual output corresponding to figure 5.7.

```
(define (match?-0 pat_0)
   (let ((m_3_0_ (length pat_0)))
      (or (= 0 m_3_0_)
          (cond
             ((equal? ’a (list-ref pat_0 0))
              (or (= 1 m_3_0_)
                  (and (equal? ’b (list-ref pat_0 1))
                       (or (= 2 m_3_0_)
                           (cond
                              ((equal? ’a (list-ref pat_0 2))
                               (or (= 3 m_3_0_)
                                   (and (equal? ’b (list-ref pat_0 3))
                                        (or (= 4 m_3_0_)
                                            (and (equal?
                                                    ’c
                                                    (list-ref pat_0 4))
                                                 (= 5 m_3_0_))))))
                              ((equal? ’c (list-ref pat_0 2))
                               (= 3 m_3_0_))
                              (else #f))))))
             ((equal? ’b (list-ref pat_0 0))
              (or (= 1 m_3_0_)
                  (cond
                     ((equal? ’a (list-ref pat_0 1))
                      (or (= 2 m_3_0_)
                          (and (equal? ’b (list-ref pat_0 2))
                               (or (= 3 m_3_0_)
                                   (and (equal? ’c (list-ref pat_0 3))
                                        (= 4 m_3_0_))))))
                     ((equal? ’c (list-ref pat_0 1)) (= 2 m_3_0_))
                     (else #f))))
             ((equal? ’c (list-ref pat_0 0)) (= 1 m_3_0_))
             (else #f)))))
```

Figure D.3: The actual result of program specialisation of 5.8 with respect to `(a b a b c)`.

```
((g105 ()
       (((>= (+ (pos-next-match sub (add1 i) n 0 i j (cons (list-ref sub (+ i j)) e*)) j) n) #t)
        ((>= (+ i (add1 j)) n) #t) ((add1 j) 5) (j 4) ((list-ref sub (+ i j)) c)))
 (g104 ()
       (((>= (+ (pos-next-match sub (add1 i) n 0 i j (cons (list-ref sub (+ i j)) e*)) j) n) #t)
        ((>= (+ i (add1 j)) n) #t) ((add1 j) 3) (j 2) ((list-ref sub (+ i j)) c)))
 (g103 (((find-match-aux-2 sub i n (add1 j) m ())                               g105))
       (((>= (+ (pos-next-match sub (add1 i) n 0 i j (cons (list-ref sub (+ i j)) e*)) j) n) #t)
        ((>= (+ i (add1 j)) n) ()) ((add1 j) 4) (j 3) ((list-ref sub (+ i j)) b)))
 (g102 (((find-match-aux-2 sub (pos-next-match sub (add1 i) n 0 i j
                                               (cons (list-ref sub (+ i j)) e*))
                           n pat j m (cons (list-ref sub (+ i j)) e*))         g104)
        ((find-match-aux-2 sub i n pat (add1 j) m ())                          g103))
       (((>= (+ (pos-next-match sub (add1 i) n 0 i j (cons (list-ref sub (+ i j)) e*)) j) n) ())
        ((>= (+ i (add1 j)) n) ()) ((add1 j) 3) (j 2) ((list-ref sub (+ i j)) a)))
 (g101 ()
       (((>= (+ (pos-next-match sub (add1 i) n 0 i j (cons (list-ref sub (+ i j)) e*)) j) n) #t)
        ((>= (+ i (add1 j)) n) #t) ((add1 j) 4) (j 3) ((list-ref sub (+ i j)) c)))
 (g100 ()
       (((>= (+ (pos-next-match sub (add1 i) n 0 i j (cons (list-ref sub (+ i j)) e*)) j) n) #t)
        ((>= (+ i (add1 j)) n) #t) ((add1 j) 2) (j 1) ((list-ref sub (+ i j)) c)))
 (g99 (((find-match-aux-2 sub i n pat (add1 j) m ())                           g101))
      (((>= (+ (pos-next-match sub (add1 i) n 0 i j (cons (list-ref sub (+ i j) e*))) j) n) #t)
       ((>= (+ i (add1 j)) n) ()) ((add1 j) 3) (j 2) ((list-ref sub (+ i j)) b)))
 (g98 (((find-match-aux-2 sub (pos-next-match sub (add1 i) n 0 i j
                                              (cons (list-ref sub (+ i j)) e*))
                          n pat j m (cons (list-ref sub (+ i j)) e*))          g100)
       ((find-match-aux-2 sub i n pat (add1 j) m ())                           g99))
      (((>= (+ (pos-next-match sub (add1 i) n 0 i j (cons (list-ref sub (+ i j)) e*)) j) n) ())
       ((>= (+ i (add1 j)) n) ()) ((add1 j) 2) (j 1) ((list-ref sub (+ i j)) a)))
 (g97 (((find-match-aux-2 sub i n pat (add1 j) m ())                           g98))
      (((>= (+ (pos-next-match sub (add1 i) n 0 i j (cons (list-ref sub (+ i j)) e*)) j) n) #t)
       ((>= (+ i (add1 j)) n) ()) ((add1 j) 1) (j 0) ((list-ref sub (+ i j)) b)))
 (g96 (((find-match-aux-2 sub i n pat (add1 j) m ())                           g102))
      (((>= (+ (pos-next-match sub (add1 i) n 0 i j (cons (list-ref sub (+ i j)) e*)) j) n) #t)
       ((>= (+ i (add1 j)) n) ()) ((add1 j) 2) (j 1)       ((list-ref sub (+ i j)) b)))
 (g95 (((find-match-aux-2 sub (pos-next-match sub (add1 i) n 0 i j
                                              (cons (list-ref sub (+ i j)) e*))
  n pat j m (cons (list-ref sub (+ i j)) e*))          g97)
       ((find-match-aux-2 sub i n pat (add1 j) m ())                           g96))
      (((>= (+ (pos-next-match sub (add1 i) n 0 i j (cons (list-ref sub (+ i j)) e*)) j) n) ())
       ((>= (+ i (add1 j)) n) ()) ((add1 j) 1) (j 0)       ((list-ref sub (+ i j)) a)))
 (g94 (((find-match-aux-2 sub 0 (length sub) pat 0 (length pat) ())            g95))
      (((>= (+ 0 0) (length sub)) ()))))
```

Figure D.4: The residual graph corresponding to figure 5.11.

```
(define (substring?-0 pat_0)
    (or (null? pat_0)
        (cond
         ((equal? (car pat_0) 'a)
          (let ((pat_1_0_ (cdr pat_0)))
              (or (null? pat_1_0_)
                  (and (equal? (car pat_1_0_) 'b)
                       (find-match-6 (cdr pat_1_0_))))))
         ((equal? (car pat_0) 'b) (find-match-6 (cdr pat_0)))
         ((equal? (car pat_0) 'c) (find-match-9 (cdr pat_0)))
         (else #f))))
(define (find-match-6 pat_0)
    (or (null? pat_0)
        (cond
         ((equal? (car pat_0) 'a)
          (let ((pat_3_0_ (cdr pat_0)))
              (or (null? pat_3_0_)
                  (and (equal? (car pat_3_0_) 'b)
                       (let ((pat_8_1_ (cdr pat_3_0_)))
                           (or (null? pat_8_1_)
                               (and (equal? (car pat_8_1_) 'c)
                                    (find-match-9 (cdr pat_8_1_))))))))))
         ((equal? (car pat_0) 'c) (find-match-9 (cdr pat_0)))
         (else #f))))
(define (find-match-9 pat_0) (null? pat_0))
```

Figure D.5: The residual program when program specialising the extending program 5.12 with respect to (a b a b c).